The Parallelization of the Mesh Refinement Algorithm in the Finite Difference Element Method Program Package

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Dipl.-Math. techn. Torsten Dirk Adolph
aus Marbach am Neckar

Tag der mündlichen Prüfung: 24. November 2005

Erster Gutachter: Prof. Dr. Roland Vollmar

Zweiter Gutachter: Prof. Dr. Willi Schönauer

Abstract

The Finite Difference Element Method program package is a robust and efficient black-box solver for the solution of arbitrary nonlinear systems of elliptic and parabolic partial differential equations under arbitrary nonlinear boundary conditions on arbitrary domains. As we can generate difference formulas of arbitrary consistency order, we get from the difference of formulas of different order an easy access to the discretization error. By the knowledge of this error we can refine the mesh locally in order to reduce the error to a prescribed relative tolerance.

In this work, we first introduce the Finite Difference Element Method and explain how we generate the difference and error formulas for the solution of the systems of partial differential equations and for the error estimate. Further, the determination of the individual consistency order for each node and the local mesh refinement is described. The domain may be composed from different subdomains with different partial differential equations and different non-matching grids. The subdomains may even slide relatively to each other. We also explain the concept of the parallelization of the Finite Difference Element Method on distributed memory parallel computers where we make use of the basic principle of the separation of the selection and the of processing of the data to save communication time.

We introduce the mesh refinement algorithm on a single processor where we explain the basic principles in 2-D and 3-D. The particuliarities of computations with dividing lines or sliding dividing lines that separate the subdomains are noted separately. The storage scheme with a maximum of three nodes on an edge of the elements induces a "refinement cascade". Afterwards we explain how we overcome the difficulties in parallelizing the mesh refinement algorithm on distributed memory parallel computers where the processors have only local data and the refinement must be synchronized by the message passing paradigm. By the 1-D domain decomposition of the grid we are able to put up clear rules that fix the owning of the nodes and elements by the processors, and from these basic rules we derive further rules to control the refinement process. The passing of messages to neighbour processors is always split up into two parts: We first only pass to the neighbour processors the message lengths for the data we really intend to send afterwards. The result of this preparatory communication step is that each processor has the knowledge which data it will receive from which processor, and therefore a time- and data-optimized communication is guaranteed for the following data exchange.

Some examples illustrate the application of the refinement and demonstrate the scalability of the algorithm.

Danksagung

Diese Arbeit entstand im Wesentlichen während meiner Tätigkeit in der Numerikforschungsgruppe am Rechenzentrum der Universität Karlsruhe (TH).

Prof. Dr. Roland Vollmar danke ich für die freundliche Übernahme des Referats sowie für zahlreiche Vorschläge zum Inhalt und zur Darstellung meiner Dissertation.

Zu großem Dank bin ich Prof. Dr. Willi Schönauer verpflichtet, der nicht nur das Thema anregte, sondern auch das Fortschreiten der Dissertation mit großem Interesse verfolgte und das Korreferat übernahm. Die zahllosen mit Prof. Dr. Schönauer geführten Diskussionen und seine vielen Anregungen waren mir während der gesamten Zeit der Anfertigung der Dissertation eine große Hilfe.
Durch die langjährige fruchtbare Zusammenarbeit, während der er mich mit seiner Begeisterung für die numerische Simulation angesteckt hat, und durch das Vertrauen, das er in mich gesetzt hat, wurde diese Arbeit erst möglich. Ich danke ihm auch für die Nachsicht, mit der er meine Launen erduldete.

Besonderen Dank schulde ich auch und vor allem meiner Ehefrau Anja Meisel, die auch in schwierigen Phasen sehr geduldig war und mich mit all ihrer Kraft großartig unterstützt hat, sowie meinem Sohn Julian Adolph, dem ich jetzt endlich wieder die Aufmerksamkeit schenken kann, auf die er viel zu lange verzichten musste.

Contents

1 Introduction **1**

2 Nomenclature **5**

3 The FDEM program package **13**

3.1 The error equation 13
3.2 The discretization error estimate 19
3.3 The generation of difference and error formulas 21
3.4 The selfadaptation process 31
3.4.1 The selfadaptation in time direction 32
3.4.2 The selfadaptation of the consistency order q 34
3.4.3 Mesh refinement 35
3.5 Coupled domains 37
3.5.1 Dividing lines 37
3.5.2 Sliding dividing lines 40
3.6 Parallelization 43
3.7 Remarks to LINSOL 48

4 The algorithm of the mesh refinement **51**

4.1 Context 51
4.1.1 Implementation language 51
4.1.2 The Portable Message Passing Interface 52
4.1.3 External procedures 54
4.1.4 Communication patterns 54
4.2 Refinement nodes and elements 61
4.3 Refinement cascade 64
4.3.1 Refinement cascade on a single processor 64
4.3.2 Refinement cascade on a distributed memory parallel computer 75
4.4 Mesh refinement on a single processor 86
4.4.1 Mesh refinement in 2-D 86
4.4.2 Extension to 3-D 103
4.4.3 Mesh refinement with dividing lines in 2-D and 3-D 111
4.5 Mesh refinement on a distributed memory parallel computer 123
4.5.1 Mesh refinement in 2-D 123
4.5.2 Extension to 3-D 160
4.5.3 Mesh refinement with dividing lines in 2-D and 3-D 172
4.6 Mesh coarsening 175
4.7 What we have learned from the mesh refinement algorithm 176

5 **Numerical examples** **185**
5.1 The test PDE . . . 185
5.2 The test of the Jacobian matrices . . . 187
5.3 System of PDEs for the 2-D test examples . . . 188
5.4 Approximating a polynomial by pure mesh refinement . . . 190
5.5 Experiments with rectangular grids . . . 196
5.5.1 Investigating all steps of FDEM . . . 196
5.5.2 Scaling of the mesh refinement only . . . 201
5.6 Example for 2-D dividing line . . . 203
5.7 Example for 2-D sliding dividing line . . . 209
5.8 A 3-D example . . . 216

6 **Summary and Outlook** **221**

References **225**

List of Algorithms **227**

List of Figures **227**

List of Listings **230**

List of Rules **231**

List of Tables **231**

1 Introduction

Many problems in physics, chemistry and engineering can be formulated by systems of partial differential equations. By the discretization of the equations we are able to get a numerical solution of these systems.

From recent textbooks to the numerical solution of partial differential equations (PDEs), see [1], [2], we learn that there are basically three main methods for the numerical solution of PDEs: The first method is the finite difference method (FDM) that dominated the early development of numerical analysis of PDEs. We search an approximate solution at the points of a uniform mesh of points, and we approximate the differential equation by replacing the derivatives by difference quotients at the mesh-points which reduces the differential equation problem to a finite linear system of algebraic equations. The PDEs and boundary conditions (BCs) can be entered directly, therefore the FDM is best suited for a black-box solver where the user enters “his” PDEs and BCs explicitly. Usually, the FDM is restricted to rectangular grids and consistency order 2. The essential drawback of the FDM is the lack of geometrical flexibility and that the solution of problems with coupled domains with different PDEs is not possible.
In the 1960s the finite element method (FEM) has been introduced by engineers, and over the last decades this method has become the most used numerical method for PDEs. This method is based on the variational formulations of the differential equations and approximates the exact solution by piecewise polynomial functions on some partition of the domain under consideration (minimization of a functional). It is more easily adapted to the geometry of the underlying domain than the FDM, i.e. we get full geometrical flexibility by unstructured meshes. The theory of the FEM is very complicated and the access to the error is very difficult, therefore it is not suitable for a black-box solver.
The third method is the finite volume method that is between the FDM and the FEM and that calculates the values of the variables averaged across the volume. It does not require a structured mesh, therefore it is powerful on coarse nonuniform grids and in computational fluid mechanics, above all for compressible flow with singularities (shocks).

Our aim is to design a robust and efficient black-box solver for the solution of all types of nonlinear elliptic and parabolic systems of partial differential equations with all types of nonlinear boundary conditions. We want a reliable error estimate so that we can refine the mesh locally and we can give each node an individual consistency order. The mesh may be unstructured on an arbitrary domain.

As we use a generalized finite difference method with arbitrary consistency order p in time direction (for parabolic partial differential equations) and arbitrary consistency order q in space direction on a typical unstructured finite element method mesh, this method is called Finite Difference Element Method (FDEM). The FEM mesh is only used to get knowledge

of the structure of the space but not for the solution of the partial differential equations.

The finite difference element method is an unprecedented generalization of the FDM which is the reason why there are no related other references. We have full geometrical flexibility, it is usable also for coupled domains with different PDEs and we can use an arbitrary consistency order. The PDEs and BCs can be entered explicitly, therefore the method is well-suited for a black-box. We have a direct explicit error estimate and by the knowledge of the error we have the possibility to optimize all the many computational parameters. FDEM is the endpoint of a 25 years' development. Additionally, there is a code that implements the method and this code is efficiently parallelized for distributed memory parallel computers. The next step is to use FDEM together with industrial partners for problems for which there does not exist standard software. Presently FDEM is used to simulate numerically PEM and SOFC fuel cells. Up to now nobody has solved the corresponding nonlinear systems of PDEs with error estimate.

By different consistency orders we get an easy and transparent access to the discretization error and therefore a reliable error estimate. It is also used for an automatic consistency order control and for mesh refinement.

To be able to solve problems on domains that are composed from different subdomains with different partial differential equations we introduced dividing lines/surfaces over which one cannot differentiate and we couple the subdomains by coupling conditions. The subdomains may even slide relatively to each other and may have non-matching grids.

The FDEM program package is efficiently parallelized by the means of MPI. The nodes are distributed in equal parts onto the processors serving load balancing and also reducing the bandwidth of the large linear system of equations by this reordering.

In order to obtain a prescribed tolerance, we refine the given mesh locally depending on the estimated relative error in the nodes. From the nodes we get the refinement elements that are refined by halving the edges. On a distributed memory parallel computer there are many difficulties to overcome such as refinement elements that are not owned by the same processor as the refinement node or refinement edges that are not owned by the same processor as the refinement element. If we have elements of different refinement stages we have to carry out several refinement steps. Here the problem is to distinguish between new nodes that are owned by a processor and nodes that are in the overlap, i.e. belong to a different processor. Before the next refinement step we have to update the node and element information at the processor borders.

This paper is organized in the following way:
In chapter 2 we summarize the notation used throughout this work.

We describe the Finite Difference Element Method in chapter 3. First we introduce the error equation in 2-D and then explain the means to get there. Afterwards we mention the differences to 3-D. Then follow the special features of FDEM. These are the selfadaptation of the consistency order and mesh refinement as well as dividing lines or sliding dividing lines respectively. We shortly explain the parallelization of FDEM on distributed memory parallel computers and say some words about the linear solver LINSOL by which we solve the large sparse linear system of equations arising from the discretization. A basic paper on FDEM is [3], a progress report is [4]. A detailed report is in preparation, see remark at the end of the References.

The parallelization algorithm for the mesh refinement is a challenging task. We introduce this algorithm in chapter 4 where we first explain the necessary steps to refine the mesh, then we discuss the concept of data exchange between the processors. After defining the refinement elements and explaining one of our main achievements, the refinement cascade, we finally consider the real mesh refinement. We begin with the mesh refinement in 2-D on a single processor, continue with the extensions to 3-D and the problems with dividing lines or sliding dividing lines, and finally extend the algorithm of the mesh refinement to distributed memory parallel computers. After some remarks to mesh coarsening we close this chapter with a short recapitulation about what we have learned from the mesh refinement algorithm.

Results of applying the mesh refinement algorithm to problems in 2-D and 3-D, also problems with dividing lines and sliding dividing lines will be given in chapter 5.

2 Nomenclature

In this chapter we introduce the notation that is used thoughout this paper.

Formula Numbers
Equations in section $(X.Y)$ are numbered $(X.Y.1)$, $(X.Y.2)$ etc.

Numbering of figures and tables
Figures and tables in section $(X.Y)$ are numbered $(X.Y.1)$, $(X.Y.2)$ etc.

Numbering of rules and listings
Rules and listings only occur in chapter 4 and therefore are numbered continuously, independent from the section number.

Numbering of algorithms
Algorithms are denoted by capital Latin letters. As algorithms also only occur in chapter 4 they are numbered continuously.

Array and message lengths
Lengths of arrays are given by the number of entries an array has got, message lengths are given in bytes.

Summaries and recapitulations
In chapter 4 we give for difficult passages a summary preceding the passage, introduced by SUMMARY, and a recapitulation following the passage, introduced by RECAPITULATION. For further separation we also change the font style here.

Special symbols
We distinguish between symbols used in chapter 3 and names of scalars and arrays used in chapter 4. In the following table, we do not list each space derivative of a variable but only the first derivative in x-direction, the derivatives for the y- and z-direction and the second derivatives are named analogously. We only present those variables that are mentioned more than once in this work. This goes for all the tables in this chapter.

Nomenclature of chapter 3.

Name	Space	Description
$\partial Pu/\partial u_{...}$	$\mathbb{R}^{l\times l}$	$l \times l$ Jacobian matrices
α	$\mathbb{R}$	pivot boundary factor
Δq	$\mathbb{N}$	surplus order
Δt	$\mathbb{R}$	time step size
Δu	$\mathbb{R}^l$	Newton correction function

Continued on next page

Name	Space	Description
Δu_d	$I\!R^l$	overall error vector
Δu_{Pu}	$I\!R^l$	Newton correction vector
Δu_x	$I\!R^l$	space derivative of Δu in x-direction
$\Delta u_{x,d}$	$I\!R^l$	difference formula for Δu_x
ε_{pivot}	$I\!R$	pivot threshold
ν	$I\!N$	Newton iteration index
ω	$I\!R$	underrelaxation factor for Newton iteration
A	$I\!R^{m\times m}$	matrix for coefficients of influence polynomials
$a_0,\ldots,a_{m-1}$	$I\!R$	coefficients of P_q
D_x	$I\!R^l$	space discretization error term, D_y and D_{xy} analogously
d_x	$I\!R^l$	discretization error estimate for $u_{x,d}$
$\overline{d}_{x,q}$	$I\!R^l$	exact discretization error for $u_{x,d,q}$
dim	$I\!N$	dimension of the computational domain, $dim \in \{2,3\}$
$epslin$	$I\!R$	factor for stopping of LINSOL iteration
$fstring$	$I\!N^{non_{max}}$	array for the nearest neighbour ring
$idstar$	$I\!N^{nle_3}$	array for the collected nodes around the central node
in_{max}	$I\!N$	maximum number of elements a node belongs to
l	$I\!N$	number of equations/components
M	$I\!R^{(m+r)\times m}$	matrix for the coefficients for the determination of the influence polynomials
m	$I\!N$	number of nodes in difference star
$m(q)$	$I\!N$	number of nodes in difference star of order q
n	$I\!N$	total number of nodes
nek	$I\!N^{nolnod}$	array for node numbers of the elements
$nekinv$	$I\!N^{in_{max}}$	array for element numbers a node belongs to (inverted nek list)
nle_3	$I\!N$	maximum number of collected nodes around a central node
$nolnod$	$I\!N$	maximum number of nodes per element
p	$I\!N$	time consistency order
P_q	$I\!R$	interpolation polynomial of order q
$P_{q,i}$	$I\!R$	influence polynomial of order q for point i
$\lvert pivot\rvert_{max}$	$I\!R$	maximum element in pivot column
$\lvert pivot\rvert_{mean}$	$I\!R$	mean value in pivot column
Pu	$I\!R^l$	general PDE operator
$(Pu)_d$	$I\!R^l$	Newton residual vector
q	$I\!N$	space consistency order
Q_d	$I\!R^l$	large sparse matrix of the linear system of equations
r	$I\!N$	number of surplus nodes for Gauss Jordan algorithm

Continued on next page

Name	Space	Description
s_{grid}	$I\!R$	factor for mesh refinement
t	$I\!R$	time variable
tol	$I\!R$	requested relative tolerance on the level of solution
$tolg$	$I\!R$	tolerance on the level of equation
u	$I\!R^l$	solution for the system of PDEs
u_d	$I\!R^l$	discretized solution vector
u_t	$I\!R^l$	time derivative of u
u_x	$I\!R^l$	x-derivative of u
$u_{x,d}$	$I\!R^l$	difference formula for u_x
$u_{x,d,q}$	$I\!R^l$	difference formula for u_x of consistency order q
x	$I\!R$	x-coordinate of a node, y and z analogously

For the symbols in chapter 4 we distinguish between scalar values and arrays.

Nomenclature of chapter 4 (scalars).

Name	Type	Description
bd_{max}	I	maximum number of external boundaries a node belongs to
el_{new}	I	number of additional elements that are generated from a refinement element
$elpt$	I	number of corner nodes in an element
in_{max}	I	maximum number of elements a node belongs to
ip	I	number of the regarded processor
m_{cnt}	I	number of own refinement edges (minus edges from the overlap of other processors)
n_{cnt}	I	number of new overlap nodes
n_{cyc}	I	number of the current computation cycle
n_{edge}	I	number of edges in an element
n_{exb}	I	number of external boundaries
n_{inb}	I	number of dividing lines
n_l	I	number of nodes owned by a processor
n_n	I	number of nodes on a processor incl. overlap
n_{max}	I	limit for the number of nodes
n_{rs}	I	number of nodes before current ref. step on a processor
n_{sect}	I	number of subdomains
nb_{max}	I	limit for the number of boundary nodes
nbm_6	I	number of entries for a refinement edge in *rtl*
ne	I	total number of elements

Continued on next page

Name	Type	Description
ne_l	I	number of elements owned by a processor
ne_{ll}	I	number of overlap elements on the left side
$ne_{ll,max}$	I	maximum number of overlap elements on the left side: $ne_{ll,max} = \max_{ip=1,...,np} ne_{ll,ip}$
ne_{lr}	I	number of overlap elements on the right side
$ne_{lr,max}$	I	maximum number of overlap elements on the right side: $ne_{lr,max} = \max_{ip=1,...,np} ne_{lr,ip}$
ne_{max}	I	limit for the number of elements
ne_n	I	number of elements on a processor incl. overlap
$ne_{n,max}$	I	maximum number of elements on a processor incl. overlap: $ne_{n,max} = \max_{ip=1,...,np} ne_{n,ip}$
ne_{re}	I	number of refinement elements in current ref. step on a processor
ne_{rs}	I	number of elements before current ref. step on a processor
neb	I	total number of external boundary nodes
neb_l	I	number of external boundary nodes owned by a processor
neb_n	I	number of external boundary nodes on a processor incl. overlap
nib_l	I	number of dividing line nodes owned by a processor
nib_n	I	number of dividing line nodes on a processor incl. overlap
nnb_{max}	I	maximum number of neighbour elements that share an edge
$nolnod$	I	maximum number of nodes per element
non_i	I	number of neighbour nodes of node i
non_{max}	I	maximum number of neighbour nodes of a node: $non_{max} = \max_{i=1,...,n_l} non_i$
$notp$	I	number of refinement edges of own elements in the overlap
np	I	number of processors
np_{max}	I	maximum of overlap processors (both left and right): $np_{max} = \max(np_{max,l}, np_{max,r})$
$np_{max,l}$	I	maximum of overlap processors on the left side: $np_{max,l} = \max_{ip=1,...,np} np_{sl,ip}$
$np_{max,r}$	I	maximum of overlap processors on the right side: $np_{max,r} = \max_{ip=1,...,np} np_{sr,ip}$
np_{sl}	I	number of overlap processors on the left side
np_{sr}	I	number of overlap processors on the right side
nsb_l	I	number of sliding dividing line nodes owned by a processor
r_{cnt}	I	counter for the edges to be refined by the processor itself
rs_{max}	I	highest refinement stage
sd_{max}	I	maximum number of sliding dividing lines a node belongs to

I: integer, DP: double precision

For the arrays we do not only give the type (integer, double precision or logical) and a short description but also—in the last row for each array—the dimension of the array that is computed with the values of the preceding table. The dimensions are valid both for 2-D and 3-D unless otherwise noted. If the dimensions change for computations wih dividing lines/sliding dividing lines this is also noted.

Nomenclature of chapter 4 (arrays).

Name	Type	Description Dimension
bnod	I	array for external boundary nodes $neb_l \times 2$
bdnr	I	array for computing new external boundary nodes $neb_l \times (bd_{max} + 1)$
dl	I	array for DL numbers for the DL nodes $nib_n \times (2 \cdot n_{inb} + 1)$
dlote	I	array for starting addresses of DL elements in *dloteadr* l_{dlote}[1]
dloteadr	I	array for the twin node information of DL edges ne_n
e_{cnt}	I	counter for the edges in *etl* $np_{max} \times 2$
etl	I	array for refinement edges in non-refinement elements $((nnb_{max} + 6) \cdot notp) \times np_{max} \times 2$
ia	I	array for starting addresses of influence polynomials in *infpol* for end points of the refinement edges $2 \cdot r_{cnt}$
iglob	I	array for starting addresses of own refinement edges in *rtl* r_{cnt}
indrel	I	refinement element list ne_l
infarr	I	array for sending update information l_{infarr}[2]
infpol	DP	array for the evaluated influence polynomials l_{infpol}[3]
ipadd	I	array for starting addresses in *infpol* n_l
lbnd	L	array for computing 3-D external boundary nodes n_n

Continued on next page

Name	Type	Description Dimension
lglob	L	array for own refinement edges r_{cnt}
lnpl	L	array for edge identification $n_n \times non_{max}$
lnpl$_2$	L	array for edge identification $n_n \times non_{max}$
logarr	L	array for computation of new DL elements in stage rs $ne_{re} \times n_{edge}$
lp	I	array for numbers of edges to receive np_{max}
lprocs	L	array for the processors *etl* has to be sent to $np \times 2$
lsent	L	auxiliary array for refinement cascade $ne_{ll} + ne_{lr}$
narpl	I	array for number of refinement elements per stage and starting addresses in *indrel* $(rs_{max} + 1) \times 2$
nb	I	array for neighbour element numbers in stage rs 2-D: $ne_{re} \times n_{edge}$ 3-D: $ne_{re} \times (n_{edge} \cdot in_{max})$
nbadd	I	array for starting addresses in *nb* in stage rs without DL: $narpl(rs + 1, 2) \times (n_{edge} + 1)$ with DL: $narpl(rs + 1, 2) \times (2 \cdot n_{edge} + 1)$
nbrs	I	array for numbers of nodes, elements, ext. boundary nodes, DL and SDL nodes (own and overlap) $(rs_{max} + 2) \times 5$
nek	I	array to store node numbers for each element $ne_n \times nolnod$
nekinv	I	array to store element numbers each node belongs to $n_n \times in_{max}$
nenr	I	array for global element numbers, subdomain number, DL element property and own processor for each local element $ne_n \times 4$
nenrs	I	array to store local element number for a global element number $ne_n \times 2$
newbn	I	array for new external boundary nodes $r_{cnt} \times 2$

Continued on next page

Name	Type	Description Dimension
newsn	I	array for new sliding dividing line nodes $r_{cnt} \times 3$
ngrid	I	array for difference stars of the end points of the refinement edges 2-D: $6 \times (2 \cdot r_{cnt})$ 3-D: $10 \times (2 \cdot r_{cnt})$
nnr	I	array for global node number, subdomain number, number of coupling domains and own processor for each local node $n_n \times 4$
nnrs	I	array to store local node number for a global node number $n_n \times 2$
p_{cnt}	I	counter array for the edges in *ptl* np_{sr}
ptl	I	array for the information of refinement edges owned by overlap processors $((nnb_{max} + 6) \cdot notp) \times np_{sr}$
rcvl	I	array for updating element information in left overlap $ne_l \times 3$
rcvr	I	array for updating element information in right overlap $ne_l \times 3$
refel	L	array for refinement elements during the refinement cascade $ne_{n,max} \times rs_{max}$
refpt	I	array for the refinement nodes n_l
rtl	I	array for the information of own refinement edges of processor ip $ne_n \cdot n_{edge} \cdot (nnb_{max} + 6)$
sect	I	auxiliary array for computation of new DL nodes n_{sect}
sndlct	I	counter array for *sndlto* $np_{sl} \times 2$
sndlto	I	array for sending elements to the left during refinement cascade $ne_{ll,max} \times np_{sl}$
sndrct	I	counter array for *sndrto* np_{sr}
sndrto	I	array for sending elements to the right during refinement cascade $ne_{lr,max} \times np_{sr}$
snod	I	array for sliding dividing line nodes $nsb_l \times (n_{sect} + 3)$

Continued on next page

Name	Type	Description Dimension
sdnr	I	array for computing new SDL nodes $nsb_l \times (sd_{max} + 1)$
tids	I	array for the physical processor numbers np
tnod	I	array for dividing line nodes $nib_l \times (n_{sect} + 3)$

I: integer, DP: double precision, L: logical

[1] see equation (4.3.1)

[2] see equation (4.5.38)

[3] see equations (4.4.8), (4.4.27)

3 The FDEM program package

In this chapter we introduce the Finite Difference Element Method we developed in our research group. We begin with the central linear system of equations that we get from our discretization method. Afterwards we explain our access to the discretization error and the way we compute the coefficients of the difference and error formulas. Then we deal with some features of FDEM such as the selfadaptation of time and space and dividing and sliding dividing lines. Finally we shortly look at the most important aspects of the parallelization and give a short summary of the linear solver LINSOL.

3.1 The error equation

The error equation is the basis of our solution method. It has been developed in its basic form for the FIDISOL program package (see [5]). The most general operator that we admit for a system of l partial differential equations and boundary conditions is in 3-D for a solution $u(t, x, y, z)$:

$$Pu \equiv P(t, x, y, z, u_t, u_x, u_y, u_z, u_{xx}, u_{yy}, u_{zz}, u_{xy}, u_{xz}, u_{yz}) = 0, \tag{3.1.1}$$

where u and Pu are vectors with l components:

$$u = \begin{pmatrix} u_1 \\ \vdots \\ u_l \end{pmatrix}, \quad Pu = \begin{pmatrix} P_1 u \\ \vdots \\ P_l u \end{pmatrix} \tag{3.1.2}$$

If we include t and u_t the system is parabolic, otherwise it is an elliptic system. Basically it is also possible to solve hyperbolic equations if they do not have discontinuities, but we do not have experiences with this type of partial differential equations. We now explain the solution method for 2-D and discuss the extension to 3-D later. The 2-D operator for the partial differential equations and boundary conditions is obtained by dropping z in (3.1.1):

$$P_u \equiv P(t, x, y, u_t, u_x, u_y, u_{xx}, u_{xy}, u_{yy}) = 0. \tag{3.1.3}$$

Pu is an arbitrary nonlinear function of its arguments. Therefore we use the Newton-Raphson method by the approach

$$u \Leftarrow u^{(\nu+1)} = u^{(\nu)} + \Delta u^{(\nu)} \tag{3.1.4}$$

where we immediately drop the iteration index ν and we linearize (3.1.3) in the Newton correction function Δu, which means that we get also the corresponding derivatives of Δu, e.g. Δu_{xx}. So we get a linear partial differential equation for the Newton correction function Δu:

$$\begin{aligned} Q\Delta u \quad &\equiv \quad -\frac{\partial P_u}{\partial u}\Delta u - \frac{\partial P_u}{\partial u_t}\Delta u_t - \frac{\partial P_u}{\partial u_x}\Delta u_x - \cdots - \frac{\partial P_u}{\partial u_{yy}}\Delta u_{yy} = \\ &= \quad P(t, x, y, u, u_t, u_x, u_y, u_{xx}, u_{xy}, u_{yy}) \end{aligned} \tag{3.1.5}$$

where $P(\ldots) \equiv Pu \equiv Pu^{(\nu)}$ is the Newton defect or Newton residual for $u^{(\nu)}$. If $u^{(\nu)}$ was the exact solution u we would have $Pu = 0$.

The $\partial Pu/\partial u_{\ldots}$ are the $l \times l$ Jacobian matrices. For a scalar partial differential equation with only one unknown variable, it is a scalar value. If we have a system of l partial differential equations where u and Pu have l components (3.1.2), the Jacobian matrices look like this:

$$\frac{\partial Pu}{\partial u} = \begin{pmatrix} \frac{\partial P_1 u}{\partial u_1} & \cdots & \frac{\partial P_1 u}{\partial u_l} \\ \vdots & & \vdots \\ \frac{\partial P_l u}{\partial u_1} & \cdots & \frac{\partial P_l u}{\partial u_l} \end{pmatrix}, \quad \frac{\partial Pu}{\partial u_x} = \begin{pmatrix} \frac{\partial P_1 u}{\partial u_{1,x}} & \cdots & \frac{\partial P_1 u}{\partial u_{l,x}} \\ \vdots & & \vdots \\ \frac{\partial P_l u}{\partial u_{1,x}} & \cdots & \frac{\partial P_l u}{\partial u_{l,x}} \end{pmatrix}, \quad \ldots \tag{3.1.6}$$

Now we discretize (index d) the linear Newton-PDE (3.1.5) by replacing function values by their value on the grid and derivatives by difference formulas:

$$\Delta u \Leftarrow \Delta u_d, \quad \Delta u_t \Leftarrow \Delta u_{t,d}, \quad \Delta u_x \Leftarrow \Delta u_{x,d}, \ldots, \tag{3.1.7}$$

where e.g. $\Delta u_{x,d}$ is the difference formula for Δu_x. However, the derivatives of the function $u = u^{(\nu)}$ in $P(\ldots)$ are replaced by difference formulas plus their error estimates:

$$u \Leftarrow u_d, \quad u_t \Leftarrow u_{t,d} + d_t, \quad u_x \Leftarrow u_{x,d} + d_x, \quad \ldots, \quad u_{xy} \Leftarrow u_{xy,d} + d_{xy}, \tag{3.1.8}$$

where e.g. d_x is the discretization error estimate for the difference formula $u_{x,d}$ (see section 3.3). For the derivatives of u we do not use error estimates because these would be errors of errors that go to zero with the Newton correction.

This discretization generates a large sparse matrix Q_d. In figure 3.1.1 it is illustrated how for a scalar partial differential equation the term $\frac{\partial Pu}{\partial x}\Delta u_x$ contributes to row i, where i denotes the central node of the formula for Δu_x. For a system of l partial differential equations there are $l \times l$ blocks instead of scalar elements.

We now linearize also in the discretization errors which again introduces Jacobian matrices (3.1.6). These additional error terms on the "level of equation", i.e. on the consistency level where we approximate a differential equation by a difference equation, create corresponding error terms on the "level of solution". If we order the coefficients of the unknown vector Δu_d that result from the discretization of (3.1.5) in Q_d we can formally express the new error as

$$\begin{aligned} & \text{level of solution} \\ \Delta u_d &= \Delta u_{Pu} + \Delta u_{D_t} + \Delta u_{D_x} + \Delta u_{D_y} + \Delta u_{D_{xy}} = \\ &= Q_d^{-1} \cdot [\ (Pu)_d + D_t + \{ D_x + D_y + D_{xy} \}\ ] \\ & \text{level of equation} \end{aligned} \tag{3.1.9}$$

This is the error equation and the key to the solution process. Q_d^{-1} is the inverse of the matrix Q_d of figure 3.1.1 but we do not explicitly compute Q_d^{-1} which would be a full matrix. Instead, we solve iteratively the corresponding linear system. $(Pu)_d$ is the "discretized" Newton residual, i.e. it is the operator $P(\ldots)$ in (3.1.3), where discretized means that all derivatives have been replaced by difference formulas that are evaluated for $u_d = u_d^{(\nu)}$. The D_μ are discretization error terms that result from the linearization in the d_μ, e.g.

$$D_t = \left(\frac{\partial Pu}{\partial u_t}\right)_d \cdot d_t, \quad D_x = \left(\frac{\partial Pu}{\partial u_x}\right)_d \cdot d_x + \left(\frac{\partial Pu}{\partial u_{xx}}\right)_d \cdot d_{xx}, \quad \ldots \tag{3.1.10}$$

In the parentheses of the second row of (3.1.9) we have error terms that can be computed "on the level of equation" and that are transformed by Q_d^{-1} to the "level of solution". These corresponding errors are arranged above these source terms. So the overall error Δu_d has been split up into the parts that result from the corresponding terms on the level of equation. The only correction that is applied is the Newton correction Δu_{Pu} that results from the Newton residual $(Pu)_d$. It is computed from

$$Q_d \Delta u_{Pu} = (Pu)_d. \tag{3.1.11}$$

The other error terms in the first row of (3.1.9) are only used for the error control. If we applied these terms we had no error estimate any more.

If we look at the discretization error term D_x in (3.1.10) we can easily see how the x-discretization errors contribute to the solution: The discretization error estimates d_x of $u_{x,d}$ and d_{xx} of $u_{xx,d}$ are multiplied with their Jacobian matrices, afterwards added and (formally) multiplied by Q_d^{-1} to transfer the error from the level of equation to the level of solution. It is a major advantage of FDEM that we can follow explicitly the propagation of

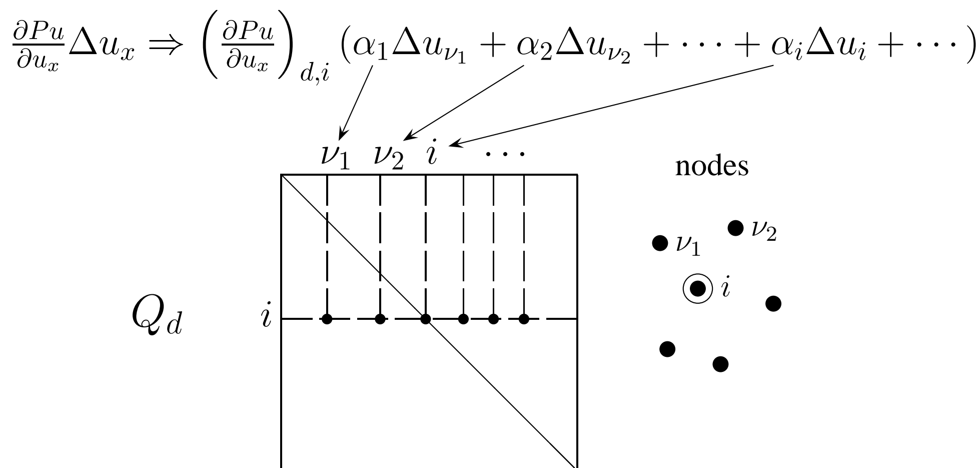


Figure 3.1.1: Illustration for the generation of row i of the matrix Q_d for a scalar PDE.

all errors.

Another question is when we stop the Newton iteration. In order to be able to give an answer we have to introduce some norms $\|\cdot\|$. By norms we reach the goal to measure the accuracy by a single scalar value. As for technical problems always the maximal values, e.g. stress or velocity, are important, we introduce maximum norms. For the solution u_d we use

$$\|u_d\| = \max_{\substack{i=1,l \\ k=1,n}} |u_{d,i,k}|, \tag{3.1.12}$$

where n is the number of nodes and l the number of partial differential equations. So $\|u_d\|$ is the maximum over all l components and all n nodes.

For the errors we want to have relative error norms. But we have to think about the possibility that a solution component may have a local zero value which makes a local relative error unfeasible. So we have to use "global relative" errors. This global relative error is computed for a component i by

$$\|\Delta u_d\|_{rel,i} \quad = \quad \frac{\underset{k=1,n}{max}|\Delta u_{d,i,k}|}{\underset{k=1,n}{max}|u_{d,i,k}|}, \tag{3.1.13}$$

i.e. we have the maximum error of the component i relative to the maximum of that component (but even this global relative error will not make sense if we have a very small component). As we can compute this global relative error for each component we are able to check the accuracy individually for each component. The overall global relative error is

$$\|\Delta u_d\|_{rel} = \max_{i=1,l} \|\Delta u_d\|_{rel,i} \tag{3.1.14}$$

i.e. the maximum of the global relative component errors.

The Newton correction Δu_{Pu} is computed from (3.1.11). The Newton residual $(Pu)_d$ gives us the stopping criterion for the Newton iteration:

$$\|(Pu)_d\| < f \cdot \max(\, 0.5 \cdot tolg, \|\{\}\| \,) \tag{3.1.15}$$

with $tolg$ from (3.4.4) (see explanation in section 3.4 on page 32) and where $\{\}$ denotes the space key error as usual, see braces in (3.1.9). f is a tuning factor which we choose $f = 10^{-3}$. The smaller we set f the smaller $\|(Pu)_d\|$ must be to stop the Newton iteration, i.e. there will be executed more Newton steps. The space key error norm $\|\{\}\|$ must be computed in each Newton step. But as it is needed for the computation of the error estimate, this may not be too much additional computation time. Especially not, if we compare

it to the time needed for the solution of the linear system of equations (3.1.11).

If the solution u_d is outside of the convergence radius the Newton-Raphson method will diverge. Therefore instead of (3.1.4) we use a damped Newton method by

$$u_d^{(\nu+1)} = u_d^{(\nu)} + \omega \cdot \Delta u_{Pu}^{(\nu)} \tag{3.1.16}$$

with an underrelaxation factor ω. Each Newton iteration step starts with $\omega = 1$. But before accepting the new iterate $u_d^{(\nu+1)}$ we check if the defect decreases, i.e. if

$$\|(Pu)_d^{(\nu)}\| < \|(Pu)_d^{(\nu-1)}\| \tag{3.1.17}$$

holds. If (3.1.17) is not true we set $\omega \Leftarrow \omega/2$, recompute $u_d^{(\nu+1)}$ and so on until either (3.1.17) holds or $\omega < 10^{-4}$. In the latter case the whole solution process of the partial differential equations is stopped and an error message is printed. This procedure converges only linearly for $\omega < 1$, but we have a very robust Newton algorithm which switches back to $\omega = 1$ as soon as possible. The price for the robustness are the additional computations of the defect if the solution u_d is outside of the convergence radius. For the first Newton step it is possible to allow the Newton residual to become greater than the Newton residual of the initial solution. This is in some cases where one starts with a zero solution guess the only possibility to find the way to the solution.

But we do not only check (3.1.15) for the stopping of the Newton iteration. We want to make sure that the relative norm of the Newton correction $\|\Delta u_{Pu}\|_{rel}$ is at least below 5%. Therefore we check together with (3.1.15) by an "and" condition that

$$\|\Delta u_{Pu}\|_{rel} < 0.05. \tag{3.1.18}$$

But if (3.1.15) holds with a factor $f_{1/10} = \frac{1}{10} \cdot f$ we stop the iteration regardless of Δu_{Pu}. If on the other hand the Newton correction is small enough, i.e. if

$$\|\Delta u_{Pu}\|_{rel} < 0.1 \cdot tol \tag{3.1.19}$$

holds, the iteration is also stopped (see page 31 for tol). This might be dangerous, because a slow convergence would look like a high accuracy, while (3.1.19) might be fulfilled although the iteration is still far from the solution. But together with the quadratic convergence of the Newton iteration and a small tolerance tol the danger is not very high.

Because of the quadratic convergence the defect decreases rapidly near the solution. Usually the solution will not change much in this situation and we have the possibility to use the "modified" Newton-Raphson method which means that a Newton step is executed with the matrix Q_d of the preceding step. This is used if the relation

$$\|(Pu)_d^{(\nu)}\| < 0.1 \cdot \|(Pu)_d^{(\nu-1)}\| \tag{3.1.20}$$

holds. The iteration converges at least linearly with a convergence factor 0.1, but the computation time for Q_d which includes the evaluation of the Jacobian matrices is saved in the modified Newton-Raphson steps.

Eventually the computation of the Newton correction Δu_{Pu} from (3.1.9) is executed by an iterative solution of the linear system. If ν denotes the index of the Newton iteration step we want to solve

$$Q_d \cdot \Delta u_{Pu}^{(\nu)} = (Pu)_d^{(\nu-1)}. \tag{3.1.21}$$

If this system for the νth Newton correction $\Delta u_{Pu}^{(\nu)}$ itself is solved by an inner iteration with iteration index μ we stop this inner iteration if

$$\|Q_d \cdot \Delta u_{Pu}^{(\nu,\mu)} - (Pu)_d^{(\nu-1)}\| \leq \|(Pu)_d^{(\nu-1)}\| \cdot epslin \tag{3.1.22}$$

holds, where

$$epslin = 0.1 \cdot \max\left[\left(\frac{\|\Delta u_{Pu}^{(\nu-1)}\|}{\|u_d^{(\nu-1)}\|}\right)^2, \frac{0.8 \cdot \|D_x + D_y + D_{xy}\|}{\|(Pu)_d^{(\nu-1)}\|}, \frac{0.4 \cdot tolg}{\|(Pu)_d^{(\nu-1)}\|}\right]. \tag{3.1.23}$$

We restrict $epslin$ by

$$10^{-4} \leq epslin \leq 10^{-1} \tag{3.1.24}$$

and because we do not have a previous Newton correction for the first Newton step we take

$$epslin = 0.1 \quad \text{for} \quad \nu = 1. \tag{3.1.25}$$

The three elements in the brackets of (3.1.23) have the following meaning: For the first term we assume that we want to solve iteratively a linear system $Ax = b$. In the kth iteration step we have x_k and $r_k = Ax_k - b$ and the error is $e_k = x - x_k$. As $Ax - b = 0$, $Ae_k = A(x - x_k) = Ax - b - Ax_k + b = -r_k$. In this notation (3.1.22) becomes

$$\begin{array}{rrcl} & \|r_k\| & \leq & epslin \cdot \|b\| \\ \Longleftrightarrow & \|-Ae_k\| & \leq & epslin \cdot \|Ax\| \\ \Longleftrightarrow & \gamma \cdot \|A\| \cdot \|e_k\| & \leq & epslin \cdot \|A\| \cdot \|x\| \\ \Longleftrightarrow & \gamma \cdot \frac{\|e_k\|}{\|x\|} & \leq & epslin. \end{array}$$

This means that it would be useless to compute by the linear solver digits which are written over by the next Newton step. In the region of quadratic convergence we expect for the error $\Delta x^{(\nu)} \approx \left(\Delta x^{(\nu-1)}\right)^2$ or a relative error norm $\left(\|\Delta x^{(\nu-1)}\|/\|x^{(\nu-1)}\|\right)^2$. In each Newton step the number of significant digits is doubled, i.e. later digits are written over by the following Newton correction.

The second and third term in the brackets mean that no digits should be computed which are in significance below the digits which are influenced by the space key error or by the prescribed tolerance tol which is transformed to $tolg$ (see on page 32).

The three coefficients 0.1, 0.8 and 0.4 on the r.h.s. of (3.1.23) are safety factors to take care of the coarse norm estimates. The restrictions (3.1.24) and (3.1.25) for $epslin$ mean that at least one and at most four significant digits of the iterated Newton correction are computed and in the first Newton step one digit is computed.

3.2 The discretization error estimate

In the error equation (3.1.9) we assume that we have estimates for the discretization errors. We want to explain the approach to these estimates for the disrcetization errors of the difference formula $u_{x,d}$. In (3.1.10) we need d_x and d_{xx}, the estimates for the discretization errors of the difference formulas $u_{x,d}$ and $u_{xx,d}$, see (3.1.8). More precisely, the difference formula for the derivative u_x had to be denoted by $u_{x,d,q}$ which indicates that this formula is of consistency order q. It holds

$$\begin{aligned} u_x &= u_{x,d,q} + \overline{d}_{x,q} = \\ &= u_{x,d,q+2} + \overline{d}_{x,q+2}, \end{aligned} \tag{3.2.1}$$

where $\overline{d}_{x,q}$ and $\overline{d}_{x,q+2}$ denote the exact discretization errors for the formulas of order q and $q+2$. If we resolve the second and the third part of (3.2.1) for the error of the actual order q and neglect the error of the higher order formula $u_{x,d,q+2}$ we get the estimate

$$d_x := u_{x,d,q+2} - u_{x,d,q} \quad \{+d_{x,q+2}\}. \tag{3.2.2}$$

In the braces we have indicated the neglected term. So the discretization error of order q is the difference of the difference formulas of order $q+2$ and actual order q. If we resolve the first and the second part of (3.2.1) for the exact error $\overline{d}_{x,q} = \overline{d}_x$, we get

$$\overline{d}_x = u_x - u_{x,d,q}. \tag{3.2.3}$$

If we compare (3.2.3) to (3.2.2) we see that we have replaced the unknown derivative u_x by a higher order formula $u_{x,d,q+2}$ for the estimate. That shows that approach (3.2.2) should only be applied if one is sure that the exact error decreases with increasing order, i.e. the neglected error $d_{x,q+2}$ must be smaller than the estimated error $d_{x,q}$. Numerical investigations with the FIDISOL program package, see [5], have shown that for the type of "central" formulas the odd orders are not better than the preceding even orders. Therefore we compute the discretization errors by the difference of the difference formulas of order $q+2$ and order q.

However, one could suppose that higher order means better solution and smaller error estimate. But if we have a coarse grid where the function values change rapidly and use a formula of high order with many nodes, there may be introduced false and irrelevant information by those nodes that are far away from the central node. So we often see that on a coarse grid higher order formulas are worse than lower order formulas. High order formulas only pay if we have high-accuracy requirement which is quite naturally coupled with a fine mesh. If the assumption of small $d_{x,q+2}$ does not hold, d_x immediately becomes very large and shows that the estimate is invalid which gives us a built-in self-control of the estimate (3.2.2). The effect of overdrawing an order lead us to the decision to use only the orders $q = 2, 4, 6$ for practical reasons. For the error estimate of the order $q = 8$ we needed the difference formula of order $q = 10$. This formula needed 286 nodes in 3-D (see (3.3.16)), so this formula is usually overdrawn on practical meshes for technical problems. The error estimate also gives us the possibility to check which order is the best one for each node and it allows us to optimize the parameters for the selection of the best m nodes for a difference formula (see next section).

In order to save time we do not explicitly compute the derivatives of order $q + 2$ as seen in (3.2.2) but we directly generate the error formulas (for the computation of the formula coefficients see section 3.3):

$$\begin{aligned} d_x \; &:= \; a_0 u_0 + a_1 u_1 + \cdots + a_{m(q+2)-1} u_{m(q+2)-1} \\ &\quad - \left(b_0 u_0 + b_1 u_1 + \cdots + b_{m(q)-1} u_{m(q)-1} \right) , \end{aligned} \tag{3.2.4}$$

where $m(q + 2)$ denotes m for 2-D from (3.3.2) and for 3-D from (3.3.16) and q has been replaced by $q + 2$, and $m(q)$ is (3.3.2) or (3.3.16). So we have

$$\begin{aligned} d_x \; &:= \; (a_0 - b_0) u_0 + (a_1 - b_1) u_1 + \cdots + (a_{m(q)-1} - b_{m(q)-1}) u_{m(q)-1} \\ &\quad + a_{m(q)} u_{m(q)} + \cdots + a_{m(q+2)-1} u_{m(q+2)-1} . \end{aligned} \tag{3.2.5}$$

Then we set

$$c_i := \begin{cases} a_i - b_i & i = 0, \ldots, m(q) - 1 \\ a_i & i = m(q), \ldots, m(q+2) - 1 \end{cases} \tag{3.2.6}$$

so that we have for the evaluation

$$d_x := c_0 u_0 + c_1 u_1 + \cdots + c_{m(q+2)-1} u_{m(q+2)-1} . \tag{3.2.7}$$

For the time derivative u_t we use backward difference formulas of order $p = 1, \ldots, 5$, see figure 3.3.9. Here we estimate the discretization error by the difference of the difference formula of order $p + 1$ and order p:

$$d_t := u_{t,d,p+1} - u_{t,d,p} . \tag{3.2.8}$$

Basically all the arguments we discussed for the spatial discretization error estimate also hold for this estimate. The coefficients of the error formulas are also stored directly like in (3.2.7).

3.3 The generation of difference and error formulas

For the generation of the difference and error formulas we make use of a finite difference method of order q which means local approach of the solution u by a polynomial of consistency order q. For reasons of simplicity we explain the approach for 2-D in detail and for 3-D we mention the differences and extensions.

The 2-D polynomial of order q is

$$P_q(x,y) = a_0 + a_1x + a_2y + a_3x^2 + a_4xy + a_5y^2 + a_6x^3 + \cdots + a_{m-1}y^q. \qquad (3.3.1)$$

This polynomial has m coefficients a_0 to a_{m-1} where

$$m = (q+1)\cdot(q+2)/2\ . \qquad (3.3.2)$$

For the determination of the m coefficients a_0 to a_{m-1} we need m nodes with coordinates (x_0, y_0) to (x_{m-1}, y_{m-1}). For example, for $q = 2$ we need $m = 6$ nodes, see figure 3.3.1.

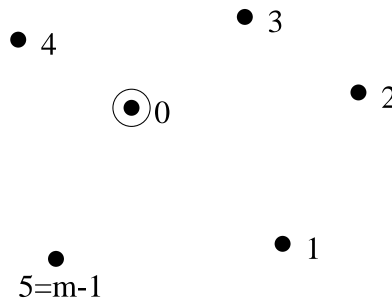


Figure 3.3.1: Example of $m = 6$ nodes for a polynomial of order $q = 2$.

In order to get difference formulas where the function values u_i appear explicitly we make use of the principle of the influence polynomials. For a point i the influence polynomial $P_{q,i}$ of order q is defined by

$$P_{q,i}(x,y) = \begin{cases} 1 & \text{for } (x_i, y_i) \\ 0 & \text{for } (x_j, y_j),\ j \neq i. \end{cases} \qquad (3.3.3)$$

This means that the influence polynomial $P_{q,i}$ has function value 1 in node i and 0 in the other $m-1$ nodes. Then the discretized solution u which we denote by u_d (the index d means “discretized”) can be represented by the composition of the function values and the influence polynomials evaluated at the formula points ($u_i = u(x_i, y_i)$):

$$u_d(x,y) := P_q(x,y) = \sum_{i=0}^{m-1} u_i \cdot P_{q,i}(x,y)\ . \qquad (3.3.4)$$

By the evaluation of $P_{q,i}$ for a grid point $x = x_j, y = y_j$ we obtain the coefficients of an interpolation polynomial at a node j, e.g. in figure 3.3.1 for node 0 where $P_{q,0} = 1$ and $P_{q,i,i\neq 0} = 0$ holds.

The difference formulas are the partial derivatives of the interpolation polynomial P_q, i.e. we have to differentiate (3.3.4). For example, for the difference formula for u_x which we denote by $u_{x,d}$ we get

$$u_{x,d} := \frac{\partial P_q(x,y)}{\partial x} = \sum_{i=0}^{m-1} u_i \cdot \frac{\partial P_{q,i}(x,y)}{\partial x}. \tag{3.3.5}$$

The other difference formulas are computed analagously, so we use $\partial^2 u_d/\partial x^2$ for $u_{xx,d}$ or $\partial^2 u_d/\partial x \cdot \partial y$ for $u_{xy,d}$.

In order to use these formulas (3.3.4) and (3.3.5) we have to evaluate the $P_{q,i}(x,y)$ resp. $\partial P_{q,i}(x,y)/\partial x$ for $x = x_j, y = y_j$. If we denote

$$\alpha_j = P_{q,i}(x_j, y_j) \tag{3.3.6}$$
$$\beta_j = \partial P_{q,i}(x_j, y_j)/\partial x \tag{3.3.7}$$

we have from (3.3.4) and (3.3.5)

$$u_d(x_j, y_j) = \sum_{i=0}^{m-1} \alpha_i \cdot u_i \tag{3.3.8}$$
$$u_{x,d}(x_j, y_j) = \sum_{i=0}^{m-1} \beta_i \cdot u_i \ . \tag{3.3.9}$$

But at first we have to determine the coefficients of the influence polynomials $P_{q,i}$. Therefore we put into (3.3.1) the coordinates (x_j, y_j) of the m surrounding nodes of the central point. So each of these m nodes creates one equation and one right hand side and we get m linear systems of equations for each node of the mesh:

$$\begin{array}{llcccccccl} & & & & & & & i & = & 0 \ \dots \ m-1 \\ \text{equ.} & 0: & 1\cdot a_{0,i} & + & x_0 a_{1,i} & + \cdots + & y_0^q a_{m-1,i} & & = & 1 \qquad 0 \\ & & 1\cdot a_{0,i} & & & & & & & 0 \qquad 0 \\ & & \vdots & & & & & & & \vdots \qquad \vdots \\ & m-1: & 1\cdot a_{0,i} & + & x_{m-1}a_{1,i} & + \cdots + & y_{m-1}^q a_{m-1,i} & & = & 0 \ \cdots \ 1\,. \end{array} \tag{3.3.10}$$

If we denote by M the coefficient matrix of the system (3.3.10) and by A the matrix where we have in the ith column the coefficients of the ith influence polynomial $P_{q,i}$ this can be written as

$$M \cdot A = I \tag{3.3.11}$$

with

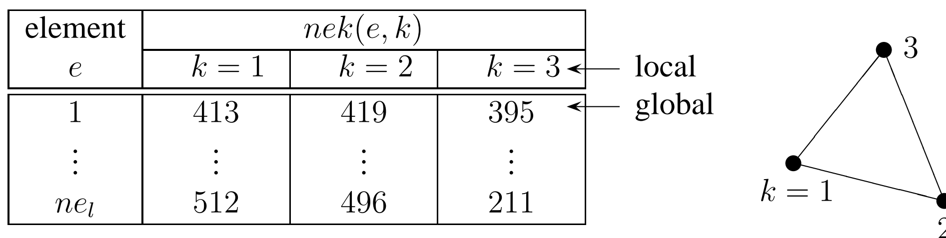
$$M = \begin{pmatrix} 1 & x_0 & y_0 & x_0^2 & x_0 y_0 & y_0^2 & \dots & y_0^q \\ 1 & x_1 & y_1 & x_1^2 & x_1 y_1 & y_1^2 & \dots & y_1^q \\ \vdots & & & & & & & \vdots \\ 1 & x_{m-1} & y_{m-1} & & & & & y_{m-1}^q \end{pmatrix}. \tag{3.3.12}$$

The solution of (3.3.11) is

$$A = M^{-1} \tag{3.3.13}$$

which means that the coefficients of the ith influence polynomial are the ith column of the inverse M^{-1} of the coefficient matrix M.

But in order to be able to form the matrix M we have to select m nodes out of all surrounding nodes of the evaluation node. As we want to have good difference and error formulas to get a good solution and error estimate, this is one of the most critical sections in the whole solution process. On the one hand we want the m nodes to be as close as possible around the evaluation node because we want to have local information in the interpolation polynomial. Wider difference stars would introduce false information if the function values change rapidly. Furthermore, they would increase the bandwidth of the resulting large sparse matrix Q_d for the solution of our partial differential equations what would lead to a larger storage requirement and higher computing time. On the other hand we also need nodes that are farther away in order to get information about the solution that closer nodes cannot give.



element	$nek(e,k)$		
e	$k=1$	$k=2$	$k=3$
1	413	419	395
$\vdots$	$\vdots$	$\vdots$	$\vdots$
ne_l	512	496	211

Figure 3.3.2: Triangle and corresponding *nek* array.

The FEM mesh, which consists of linear triangles in 2-D, is generated by a commercial mesh generator like I-DEAS or PATRAN. Each triangle consists of three nodes that are stored in an array that is denoted by *nek*, see figure 3.3.2. For the determination of the m surrounding nodes we need the inverted *nek* array that is denoted by *nekinv* where for each node the global element numbers of all those elements are stored in which the node

occurs, see figure 3.3.3.

In order to get this array we first go through all nodes in each column of the *nek* array and only count how often each element occurs. The maximum value is denoted by in_{max}. Now we can define the array *nekinv* with length equal to the number of nodes and width equal to in_{max}. Now we go again through all nodes in each column of the *nek* array but this time the element numbers are stored in the *nekinv* array. The column number in which the element number has to be stored is given by an index counter that is increased by one each time a node occurs.

Now we can determine the nearest neighbour ring for each evaluation node. The nearest neighbours of a node are all those nodes that occur in the elements the node itself belongs to, see figure 3.3.4. For each node we go through the *nekinv* array and get the elements it belongs to. Then we get all nodes that belong to these elements from the corresponding rows in the *nek* array. In order to avoid multiple storing of the same node number we use a logical array that extends over all nodes and that is initialized by *false*. In this array we enter a *true* for each neighbour node. At the end we also enter a *false* for the central node. The nodes that correspond to the columns in the logical array that contain a *true* at the end form the nearest neighbour ring. By counting the *true* entries we get the number of neighbours non_i for node i. The maximum number non_{max} of *true* entries is computed and an integer array that we denote by *fstring* (first ring) is defined with the width equal to this maximum. Now the logical array can be transformed into an integer array, see figure 3.3.5.

With the aid of this array *fstring* it is now easy to determine the next rings around the central node. For the determination of the next neighbour ring we again use a logical array for the same reason as above. This array is initialized by *false*. We have to go through all nodes of the current ring and have to enter a *true* into a logical array for all the neighbour nodes of every node in the ring (the logical array is used for the same reason as above). Then we enter a *false* for all nodes we already collected in earlier rings. Now we have stored a *true*

node nr. i	index counter in_{cnt}	k			$nekinv(i,k)$				
		1	2	3	4	5	6	7	$8=in_{max}$
1	5	2	6	3	12	9	0	0	0
2	6	6	3	1	18	8	16	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
n_l	8	61	58	64	62	60	51	69	70

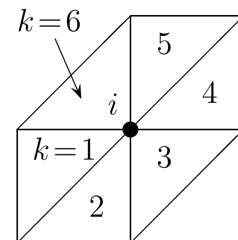


Figure 3.3.3: Array *nekinv* which gives for each node the element numbers in which it occurs.

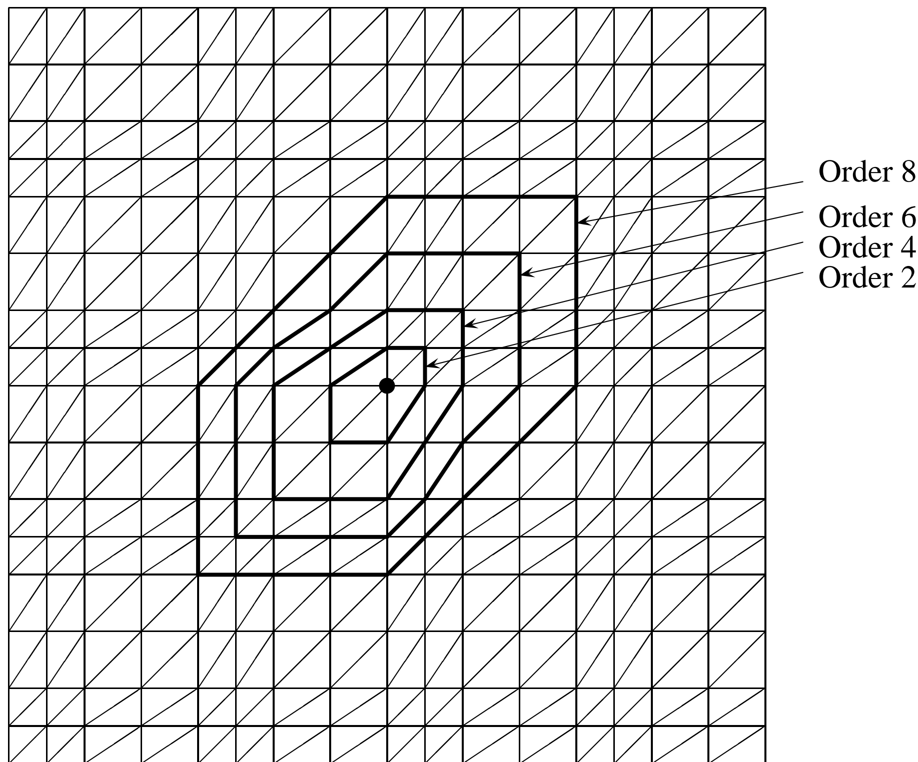
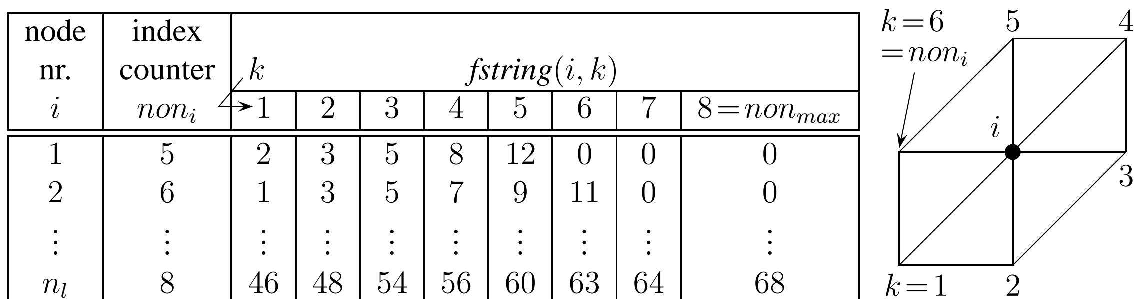


Figure 3.3.4: Nearest neighbour ring and 3 further rings for inner node.

for each node belonging to the next neighbour ring and we transform the logical array into an array that is denoted by *idstar* with length equal to n and width equal to nle_3 where n is the number of nodes and nle_3 is the maximum number of nodes to collect for a central node. As we do not know the maximum number of nodes that will be collected for a central node in advance, we set nle_3 to an order depending value at the beginning. If the width is



node nr. i	index counter non_i	k → 1	2	3	4	5	6	7	$8 = non_{max}$
		fstring(i,k)							
1	5	2	3	5	8	12	0	0	0
2	6	1	3	5	7	9	11	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
n_l	8	46	48	54	56	60	63	64	68

Figure 3.3.5: Array *fstring* with the node numbers of the nearest neighbour ring.

insufficient, we increase this value by 500 and restart the point collection.

As we saw above we only need m nodes for the determination of the matrix M (3.3.12) because we have m coefficients in each of the m influence polynomials. But then there is the risk that the matrix could become singular if the m nodes are linearly dependent. This occurs easily for a rectangular grid. So we do not search only for nodes up to order q. Instead we search for nodes up to order $q + \Delta q$ where Δq is the so-called surplus order. Δq must be at least equal to 2 because the error formulas we generate are formulas of order $q + 2$. So usually we set $\Delta q = 4$ because we also need additional nodes for the error formulas in order to avoid linear dependencies.

Another criterion that the collected nodes must fulfil is that we have to collect enough rings. In figure 3.3.6 we need for the central node at the boundary for the order $q = 2$ at least 3 nodes in the x-direction, i.e. 2 rings, and for the error order $q + 2 = 4$ we need even 4 rings. Therefore we collect $q + 2$ rings around each node. The number of surplus nodes we have collected is denoted by r.

It depends on the mesh which of the two limits is decisive and it has not to be the same for each node of the mesh. For example, on a rectangular mesh shown in figure 3.3.6 for an inner node the second limit is stronger and for a corner node it is the first one.

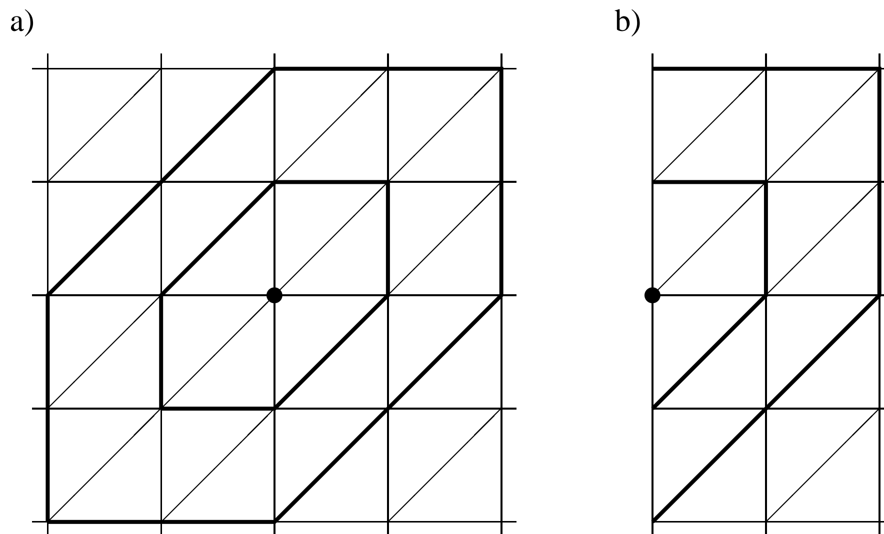


Figure 3.3.6: Illustration of node collection limits for the order $q = 2$ on a rectangular mesh a) for an inner node, b) for a boundary node.

Now the problem is to select the best m nodes out of the $m + r$ nodes we have collected.

The situation is illustrated in figure 3.3.7. We need the $m \times m$ matrix M (3.3.12) but by the surplus nodes we have the $(m+r) \times m$ matrix "M". The problem is, how to select the matrix M out of the matrix "M".

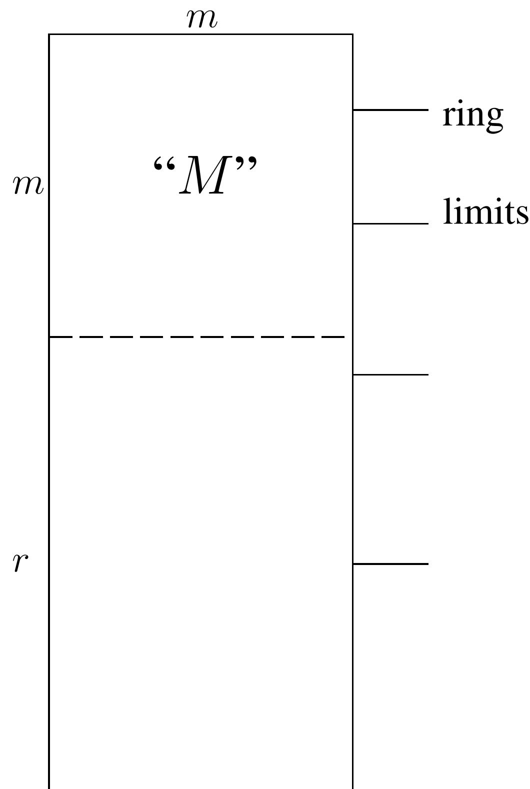


Figure 3.3.7: Illustration for the selection of m equations out of $m + r$ equations.

In order to get the matrix A we have to invert the matrix M (3.3.12). This is done by the Gauss Jordan algorithm with row pivoting to avoid the selection of two linearly dependent nodes. We search for the largest absolute value in the pivot column below the main diagonal element, shift all rows from the actual row to the row with the pivot element down by one and the pivot row replaces the actual row. To avoid difference stars with nodes that are far away from the central node we arrange the collected nodes or equations resp. in the matrix "M" in the sequence of the rings, see figure 3.3.7. The pivot search starts in the actual ring with the main diagonal element and is continued up to the last element in the actual ring. Then we compare the maximal value in this ring with a prescribed value ε_{pivot} and only accept the found pivot element if

$$|pivot| \geq \varepsilon_{pivot}. \quad (3.3.14)$$

If (3.3.14) does not hold we cross the ring limit and continue the search in the next ring.

This strategy gives narrow difference and error formulas of high quality. The two parameters Δq and ε_{pivot} are the key parameters for the generation of the formulas and therefore the key parameters for the whole solution process. The value of ε_{pivot} depends on the order q and may also depend on the type of grid. Quite robust values are

$$\varepsilon_{pivot} = \begin{cases} 10^{-2} & \text{for order 2} \\ 5 \cdot 10^{-3} & \text{for order 4} \\ 10^{-3} & \text{for order 6.} \end{cases}$$

The algorithm we just described has been developed for 2-D examples. However, it failed for certain 3-D examples. So we had to change some details which we will describe in the following.

A 3-D polynomial of order q is now

$$\begin{aligned} P_q(x,y,z) &= a_0 + a_1 x + a_2 y + a_3 z + a_4 x^2 + a_5 xy + a_6 xz + a_7 y^2 + \\ &\quad + a_8 yz + a_9 z^2 + a_{10} x^3 + \cdots + a_{m-1} z^q. \end{aligned} \tag{3.3.15}$$

This polynomial has m coefficients a_0 to a_{m-1}, where

$$m = (q+1) \cdot (q+2) \cdot (q+3)/6. \tag{3.3.16}$$

These coefficients are determined as in 2-D, see [6]. For the selection of the nodes for the matrix "M" we have now nearest neighbour balls but for reasons of simplicity we will call them "rings" as in 2-D. In the matrix "M" we first form new rings with the nodes of 2 old rings each. Then we sort the nodes by distance to the central node in each of the new rings. So there are the sorted nodes of ring 1 and 2, followed by the sorted nodes of ring 3 and 4, etc.

For the pivot search we do not only prescribe a value ε_{pivot} but also a second value α. During the elimination process we determine the mean pivot element $|pivot|_{mean}$ and the maximum pivot element $|pivot|_{max}$ in the pivot column where we take into consideration only the elements below the diagonal element. In the pivoting process we now do not pay any attention to the ring borders. We go down in the pivot column, starting with the diagonal element, and accept an element as pivot element if

$$|pivot| \geq \min\left(\varepsilon_{pivot},\ \alpha \cdot |pivot|_{mean},\ |pivot|_{max}\right). \tag{3.3.17}$$

The greater we choose ε_{pivot} and the smaller we choose α the more the second criterion is decisive. If we choose the values the other way round the first criterion is decisive. And if we choose a greater value for both ε_{pivot} and α the $|pivot|_{max}$ term is equal to the minimum

(note that here $\alpha > 1$ must hold).

If we find an element that fulfills (3.3.17) in row k this row becomes the new pivot row and all rows from the old pivot row to row $k-1$ are shifted down by one.
A typical value for α is $\alpha = 10^{-3}$ (ε_{pivot} is chosen like in 2-D) but again for some grids or problems a different value may result in a better error estimate. So the error estimate can be used to optimize α for a certain class of problems.

In the matrix "M" are the values of the coordinates x_i, y_i and higher powers of these coordinates according to (3.3.12). So in the pivoting process we would prefer nodes that have larger values of x or y. Therefore we transform the set of nodes that has been selected so that the central node becomes the origin and the largest x- or y-coordinate is at $x = \pm 1$ and $y = \pm 1$, see figure 3.3.8.

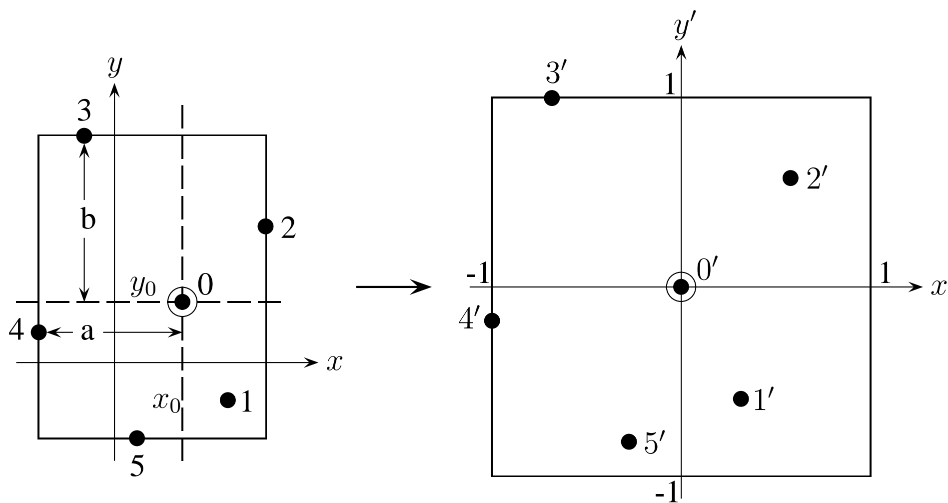


Figure 3.3.8: Illustration for the coordinate transformation $(x, y) \rightarrow (x', y')$.

If a and b are the maximal distances of a node in x- and y-direction from the central node and x_0 and y_0 are the coordinates of the central node, we have

$$x = a \cdot x' + x_0, \quad x' = \frac{1}{a} \cdot (x - x_0),$$
$$y = b \cdot y' + y_0, \quad y' = \frac{1}{b} \cdot (y - y_0). \tag{3.3.18}$$

Now in the pivoting process we would prefer nodes that are far away from the central node. To avoid this we normalize the matrix "M" to absolute row sum equal to one, i.e. we divide each element in a row by the sum of the absolute values of all elements of the row.

We determine the coefficients of the influence polynomials in the transformed and normalized system, so the resulting coefficients are a_i' and the polynomial (3.3.1) now writes

$$\begin{aligned} P_q(x,y) &= a_0' + a_1'\left(\frac{1}{a}(x-x_0)\right) + a_2'\left(\frac{1}{b}(y-y_0)\right) + \\ &\quad + a_3'\left(\frac{1}{a}(x-x_0)\right)^2 + \cdots + a_{m-1}'\left(\frac{1}{b}(y-y_0)\right)^q . \end{aligned} \tag{3.3.19}$$

Afterwards this polynomial is differentiated so that we then get the coefficients of the derivatives of the m influence polymials in the transformed and normalized system.

In order to get the coefficients a_i of the original system we have to multiply the derivatives of each influence polynomial by the absolute row sum of the corresponding row, i.e. the derivatives of influence polynomial 1 must be multiplied by the absolute row sum of row 1, the derivatives of influence polynomial 2 must be multiplied by the absolute row sum of row 2 etc. Now the x-derivatives of the influence polynomials must be multiplied by $\frac{1}{a}$, the y-derivatives must be multiplied by $\frac{1}{b}$, the xx-derivatives by $\frac{1}{a^2}$ and so on. Then we have computed the coefficients of the derivatives of the influence polynomials in the original system.

One should imagine what happens in 3-D: E.g. for a $100 \times 100 \times 100$ unstructured tetrahedral FEM mesh, generated by a mesh generator, for each of the one million nodes the nearest neighbour ring (ball) must be determined, and from that the necessary set of nodes for the $(m+r) \times m$ matrix "M". For each "M" the m appropriate nodes for the difference (and error) formulas of arbitrary order q must be selected with the criterion (3.3.17). This must be done for interior nodes and boundary nodes. As FDEM is a black box solver one never knows what mesh a user puts into the algorithm. Later we discuss the selfadaptation of the mesh and even of the order q. So the mesh and the difference formulas may be changing during the solution process. All these items necessitate an extremely robust algorithm. In such a situation it is mandatory to have an error estimate that tells us if our solution is reliable and how good it is.

Here are some remarks to "mesh-free" methods. We use in 2-D the triangular and in 3-D the tetrahedral FEM mesh only for the structure of the space, to determine the neighbourhood relations between the nodes. If we have determined for each node its nearest neighbour ring (ball), we forget the FEM mesh. From this point on we have a "mesh-free" method that operates only on the nodes. So one could use instead of the FEM mesh an arbitrary set of points in the 2-D or 3-D space, with the information which of the points are boundary points. Then one had to invent an algorithm to determine the nearest neighbour ring for each point. The simplest but most expensive algorithm is the search for the distance. But then there is the question how to distribute the nodes in the computational space. Such a

distribution will be made efficiently by a triangular or tetrahedral grid that gives automatically the structure of the space. And thus we are back at our FDEM.

Up to here we have discussed the generation of 2-D or 3-D difference formulas for spatial direction. For parabolic partial differential equations we need 1-D formulas in time for the time derivative u_t. We use backward difference formulas of consistency order p that lead in FDEM to fully implicit methods for parabolic equations. Figure 3.3.9 shows symbolically the formulas for the orders $p = 1$ to 3. For stability reasons we use the formulas only up to the order $p = 5$. The generation of such 1-D interpolation and difference formulas of type

$$P_p(t) = b_0 + b_1 t + b_2 t^2 + \cdots + b_p t^p \tag{3.3.20}$$

has been presented in detail in [7]. We use the Newton interpolation polynomial for the generation of the $p + 1$ influence polynomials that are easily determined by Newton's scheme of divided differences. Basically they also could be determined by a 1-D version of our space method.

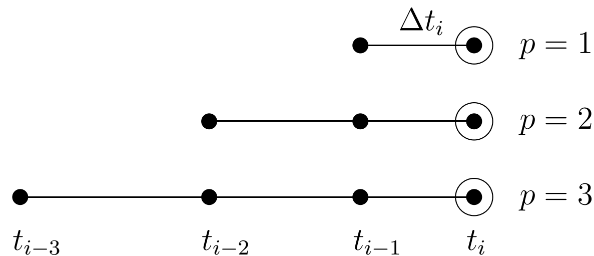


Figure 3.3.9: Illustration for the difference formulas of order p for u_t.

3.4 The selfadaptation process

Before we discuss the selfadaptation for time and space we need a scale for the accuracy on the level of equation in the sense of the error equation (3.1.9). Therefore the user prescribes a global relative tolerance tol for the solution and stops the refinement process if

$$\|\Delta u_d\|_{rel} \leq tol \tag{3.4.1}$$

holds. For the control of the solution process we need a corresponding value on the level of equation. So we use the argument that the tolerances on the level of equation and on the level of solution behave like the norms of the errors on the level of equation ($\|(Pu)_d\|$, Newton residual) and on the level of solution ($\|\Delta u_{Pu}\|_{rel}$, relative Newton correction). So

we get the following relation for $tolg$ on the level of equation:

$$\frac{tolg}{tol} = \frac{\|(Pu)_d\|}{\|\Delta u_{Pu}\|_{rel}}. \tag{3.4.2}$$

With the relative Newton correction

$$\|\Delta u_{Pu}\|_{rel} = \frac{\|\Delta u_{Pu}\|}{\|u_d\|} \tag{3.4.3}$$

it holds for $tolg$

$$\begin{aligned} tolg &= tol \cdot \|(Pu)_d\| / \|\Delta u_{Pu}\|_{rel} \\ &= tol \cdot \|u_d\| \cdot \frac{\|(Pu)_d\|}{\|\Delta u_{Pu}\|}. \end{aligned} \tag{3.4.4}$$

3.4.1 The selfadaptation in time direction

Now the control procedure in t-direction, i.e. in the initial-value direction of parabolic partial differential equations, is discussed. This contains the computation of the time step size Δt and of the consistency order p in time direction for parabolic problems. As already mentioned we use backward difference formulas of the type of figure 3.3.9 and error estimates of type (3.2.8). The user has to give the starting solution of his parabolic problem at the initial time t_0. At the time t_k we want to compute a time increment Δt_{k+1} for the next time step that makes the size of the time discretization error term D_t (3.1.10) about $1/3$ the size of the space key error term $\|D_x + D_y + D_{xy}\|$ in the error equation (3.1.9) in the sense of error balancing, or about $1/3$ the size of $tolg$. If we have equidistant time step size Δt the time discretization error is $\sim (\Delta t)^p$. Then the time step size Δt_{k+1} to calculate the next time step

$$t_{k+1} = t_k + \Delta t_{k+1} \tag{3.4.5}$$

is determined by the smallest necessary step size over all l components

$$\Delta t_{k+1} = \min_{i=1,l} \left(\left[\frac{1}{3} \max(tolg, \frac{\|D_{x,i} + D_{y,i} + D_{xy,i}\|)}{\|D_{t,i}\|} \right]_k^{1/p} \right) \cdot \Delta t_k. \tag{3.4.6}$$

The space discretization error $D_x + D_y + D_{xy}$ only depends on the space grid and the space consistency order and therefore plays the role of a key error to which the time discretization error D_t has to be adapted by a suitable choice of the time step size. Therefore we check at the time t_{k+1} whether the following condition holds

$$\|D_{t,i}\| < \max\left(tolg, \|D_{x,i} + D_{y,i} + D_{xy,i}\|\right) \tag{3.4.7}$$

for all $i = 1, \ldots, l$. If (3.4.7) does not hold, the solution is cancelled and a new value Δt_{k+1} is computed using the values of t_{k+1} instead of t_k.

The initial time step size Δt_1 is prescribed by the user. Also limits Δt_{min} and Δt_{max} for the step sizes must be given. If (3.4.6) results in $\Delta t_{k+1} < \Delta t_{min}$ the computation is continued with Δt_{min}, if $\Delta t_{k+1} > \Delta t_{max}$ the computation is continued with Δt_{max} to get a sufficiently dense information.

If the next time step t_{k+1} is determined, FDEM extrapolates an initial solution for this time step from the solution of the preceding time step t_k. If the Newton-Raphson iteration does not converge with this initial solution, the time step size Δt_{k+1} is reduced by a factor of 0.5. If Newton-Raphson diverges twice, then also the consistency order p is reduced by one. If in this way Δt_{min} is reached the computation is stopped.

Now we will explain the control of the consistency order p. For all components $i = 1, \ldots, l$ we compare the norm of the discretization error for the actual order p and the neighbouring orders $p-1$ and $p+1$. If for only one component i holds $\|D_{t,i}\|_{p-1} < \|D_{t,i}\|_p$, the order is reduced by one. The order is increased by one, if for all components i $\|D_{t,i}\|_p > \|D_{t,i}\|_{p+1}$ holds. If neither the first nor the second situation is fulfilled, the order is maintained.

The computation starts with three steps of order $p = 1$ where the time step size is adapted to (3.4.6), then there is enough information to decide if the order $p = 2$ would be better. After the next time step there is the information for the order $p = 3$ available and so on. But there is one peculiarity: We start with two time steps with the prescribed time step size Δt_1 and then we are able to estimate the error for the first time step by the difference of the first order backward difference formula and the second order central difference formula for t_1. Then Δt_1 is adapted until the condition (3.4.6) holds.

Furthermore it is possible to get together with the solution the global error (accumulated error of all time steps). At each time step t_k there is a local error in time direction that is computed from the error equation (3.1.9). But this would be the correct error only if the solution in the preceding time steps $t_{k-\nu}$ was exact. If one wants to compare the error estimate with the exact error for a known test-PDE, one has to take the global error. To calculate this global error, an additional term D_t^{fix} for the error of the preceding time steps is appended in the error equation (3.1.9):

$$\Delta u_{d,g} = Q_d^{-1} \cdot \left[(Pu)_d + D_t + \{D_x + D_y + D_{xy}\} + D_t^{fix} \right], \tag{3.4.8}$$

where

$$D_t^{fix} = \frac{\partial P}{\partial u_t} \cdot d_t^{fix}. \tag{3.4.9}$$

The value d_t^{fix} is fixed in the current time step and is computed by

$$d_t^{fix} = \sum_{k=2}^{p+1} \Delta u_k \frac{\partial P_{p,k}}{\partial t}(t) \tag{3.4.10}$$

where $\Delta u_k = \Delta u(t_k)$ ($k = 2, \ldots, p+1$) are the global errors on the level of solution of the preceding time steps and P_p is (3.3.20). So the error profiles of each time step must be stored. We store the last seven profiles because of the error estimate by the order $p = 6$ for the maximal consistency order $p = 5$ in time direction.

3.4.2 The selfadaptation of the consistency order *q*

Our goal is that each node can have its individual optimal consistency order q which is a unique feature that becomes possible only by our simple and explicit discretization error estimate. So first we compute the coefficients of the error formulas for the three consistency orders $q = 2, 4, 6$ for each node. For the optimization of the order we compute in each node i for the current solution (independent of its consistency order) the local space key error norm (see (3.1.9)) for the orders $q = 2, 4, 6$:

$$\left\| \{D_x + D_y + D_{xy}\}_i \right\|_{q=2, q=4, q=6}, \tag{3.4.11}$$

i.e. we compute this norm using difference formulas of order $q = 2, 4, 6$. What is the optimal order q for each node? One would tend to select the order with the smallest space key error norm. However, higher order is more expensive because higher order difference formulas have more nodes in the difference star, so that the corresponding rows in the large sparse matrix Q_d (see figure 3.1.1) contain more nonzero entries. Therefore we accept the higher order only if

$$\|\{\}_i\|_{\text{higher order}} \leq f \cdot \|\{\}_i\|_{\text{lower order}} \tag{3.4.12}$$

with f as a tuning parameter that is chosen to minimize the overall computation time. Presently we take

$$f_{2\leftrightarrow 4} = 0.5, \quad f_{4\leftrightarrow 6} = 0.01 \tag{3.4.13}$$

which means that the space key error norm of order $q = 6$ must be below 1% of the space key error norm of order $q = 4$ to choose the order $q = 6$.

As we have to store the coefficients of the difference and error formulas of order $q = 2, 4, 6$ for each node the possibility to set the consistency order individually for each node is very expensive. An example will illustrate this: For a $100 \times 100 \times 100$ unstructured tetrahedral FEM mesh in 3-D we have $m(2) = 10$, $m(4) = 35$, $m(6) = 84$ and $m(8) = 165$ (for the error of order $q = 6$). We have 9 derivatives (u_x, u_y, u_z, u_{xx}, u_{yy}, u_{zz}, u_{xy}, u_{xz} and u_{yz}) and

	length[1] of array for	
order	difference formulas	error formulas
2	686.65	2403.26
4	2403.26	5767.82
6	5767.82	11329.65

[1] in MByte

Table 3.4.1: Length of arrays for difference and error formulas for the orders $q = 2, 4, 6$.

thus the arrays for the difference and error formulas have the lengths shown in table 3.4.1.

These values are computed the following way:

$$mem = \frac{n \cdot n_d \cdot m(q) \cdot mem_{dp}}{f_{mb}} \tag{3.4.14}$$

where

$$\begin{aligned} n &= \quad 10^6 = \text{number of grid nodes} \\ n_d &= \quad\; 9 = \text{number of derivatives} \\ mem_{dp} &= \quad\; 8 = \text{number of bytes for a double precision variable} \\ f_{mb} &= 1024^2 = \text{factor to convert bytes into Megabytes.} \end{aligned}$$

So the total memory needed for all difference and error formulas is 27.69 GByte! In comparison, the length of the large sparse matrix Q_d for order $q = 4$ of a system of 6 partial differential equations and 10^6 nodes is 9.39 GByte. As many of the matrix entries are computed zeros the amount of nonzeros is only about one third of this value. Of course, the time consumption also increases, especially for order $q = 6$ in 3-D. However, it gives the most reliable error estimate because in each node it is checked if the order is overdrawn which is visible by a larger space key error term for the higher order.

3.4.3 Mesh refinement

The user prescribes a global relative error tol that is checked against (3.1.14) so that finally (3.4.1) holds. The computation starts with an initial mesh from a mesh generator. If the requested accuracy tol is smaller than $\|\Delta u_d\|_{rel}$ the only possibility is to refine the mesh locally. So we check at each node if

$$\|\{D_x + D_y + D_{xy}\}_i\| \leq s_{grid} \cdot tolg. \tag{3.4.15}$$

Here s_{grid} again is a tuning parameter that must be determined to minimize the solution time. The optimal value depends on the type of problem to be solved so that no special value can be recommended.

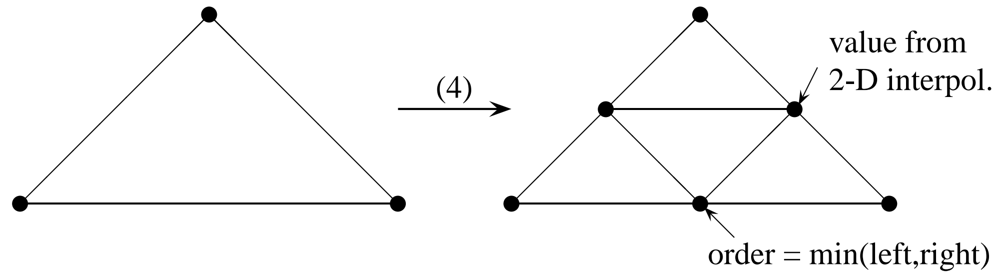


Figure 3.4.1: Refinement of a linear triangular element.

A node where (3.4.15) fails is a refinement node. If in an element at least one node is a refinement node, the whole element will be refined by halving the edges, so that four similar triangles result, see figure 3.4.1.

According to this locally refined meshes result. One of the four new elements gets the number of the old element and three new element numbers are generated. If the neighbour triangle is also refined the node on the shared edge is generated only once, of course. The function value u_d for a newly-created node is determined in the following way: We interpolate from both end points of the subdivided edge the function value for the mid-point by a 2-D interpolation formula of order q_i (3.3.8) for the end point i. Then we take the mean value of the interpolated values. The order of the new node is the minimum of the orders of its two neighbours.

In figure 3.4.2a) node 1 is a refinement node because of the error so that we have to refine element A. The result is the grid shown in figure 3.4.2b). Note that triangle B has two edges with two nodes and one edge with three nodes after the refinement process. If in a second refinement step node 2 becomes a refinement node we have to refine the triangles C, D and E. If the left neighbour triangle B is not refined, there would be more than three nodes on an edge of this triangle (node 3 would be the fourth node on this edge, see figure 3.4.2c)). For the basic organization of the element arrays we therefore limit the number of nodes on an edge to three (see Rule 2 on page 65). If a fourth node would be created on an edge, the larger triangle also must be refined (result shown in figure 3.4.2c)), but now not because of the error but for reason of data storage scheme. This induces the so-called refinement cascade.

The whole refinement process is described in detail in chapter 4.

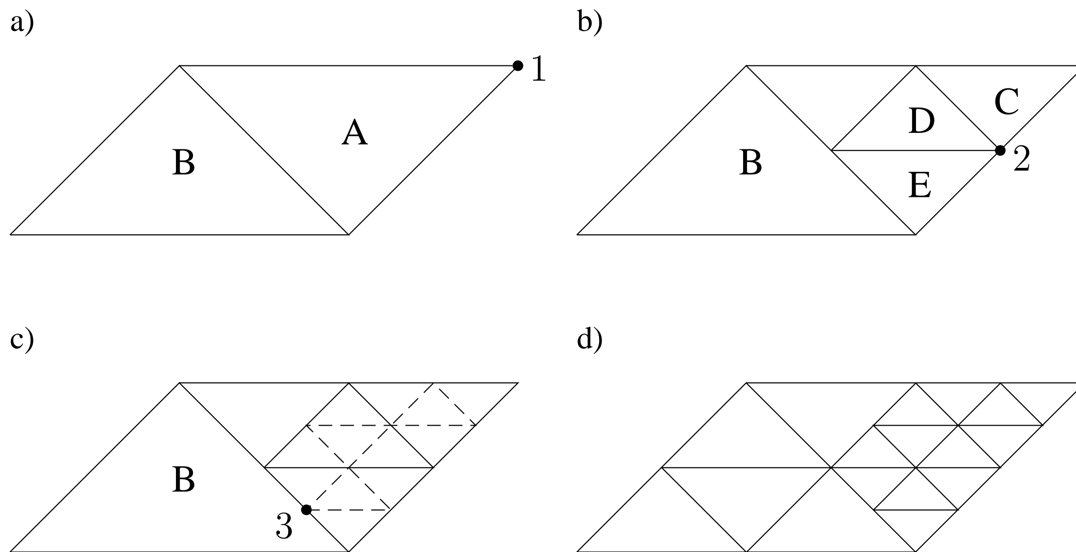


Figure 3.4.2: Illustration of the refinement cascade on a single processor: a) original grid, b) grid after one refinement, c) grid after two refinement steps without refinement cascade, d) grid after two refinement steps with refinement cascade.

3.5 Coupled domains

The finite difference element method needs uniform field equations on the whole domain. However, in technical applications we often have coupled domains with different partial differential equations. It is not possible to differentiate across these dividing lines or surfaces because there may be a jump in the derivative (figure 3.5.1a)) or even in the function itself (figure 3.5.1b)).

3.5.1 Dividing lines

In figure 3.5.2 we have a block composed of four different materials with different heat conduction coefficients. If we want to compute the heat flux in the whole block subjected to some boundary condition, we can discretize the whole solution domain, but we cannot differentiate across a material boundary. Therefore we must compute the solution separately in the subdomains and couple the different solutions.

So first each geometrical dividing line node which we get from our mesh generator (see left side of figure 3.5.2) has to be transformed to n_{dl} logical dividing line nodes where n_{dl} is the number of domains a geometrical dividing line node belongs to. One of these logical nodes keeps the old node number, the other ones get new numbers. So each of the logical nodes belongs afterwards to exactly one domain and has different difference stars and therefore a different solution and thus is unique. This is shown on the right side of figure 3.5.2.

The n_{dl} domains are coupled by n_{dl} coupling conditions. We do not have two crossing dividing lines but four dividing lines that meet in a quadruple point. The user must specify which coupling condition is valid for which part of the multiple node. The coupling conditions have the following form:

$$\begin{aligned} \underline{i}, j : (P_1 u)_{d,i} + (P_2 u)_{d,j} &= 0 \\ \underline{j}, i : (P_3 u)_{d,j} + (P_4 u)_{d,i} &= 0. \end{aligned} \quad (3.5.1)$$

The first one is for dividing line nodes on domain i that couples to domain j, and the second is for dividing line nodes on domain j. These coupling condition operators have the same form as the operator for the partial differential equations.

For a simple dividing line we have symmetric coupling conditions which means that they also can be interchanged without any difference for the result. For the heat conduction problem we have equal temperature and equal heat flux which means

$$\begin{aligned} T_1 &= T_2, \\ \lambda_1 T_{1,x} &= \lambda_2 T_{2,x} \end{aligned}$$

with heat conduction coefficients λ_1 and λ_2 (see figure 3.5.2). For nodes of the first domain we have equal temperature as coupling condition, and for nodes on the second domain we need equal heat flow. This could also be the other way round.

As we saw before a geometrical node on n_{dl} domains needs n_{dl} coupling conditions but here we have to pay attention because it is for example not allowed to give the following

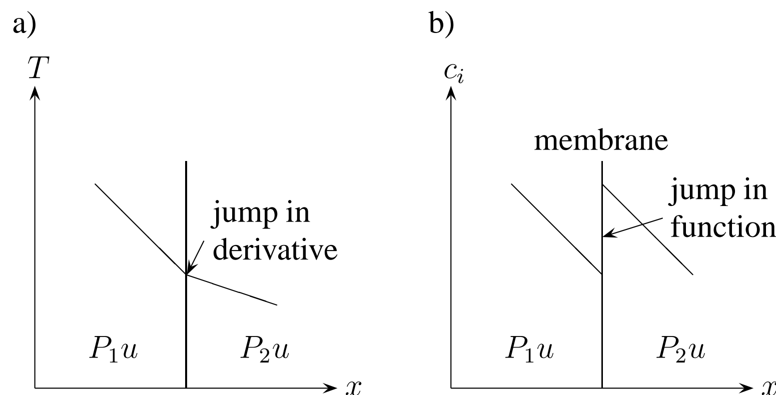


Figure 3.5.1: Illustration for dividing lines (DLs) and coupling conditions (CCs) with a) jump in derivative, b) jump in function.

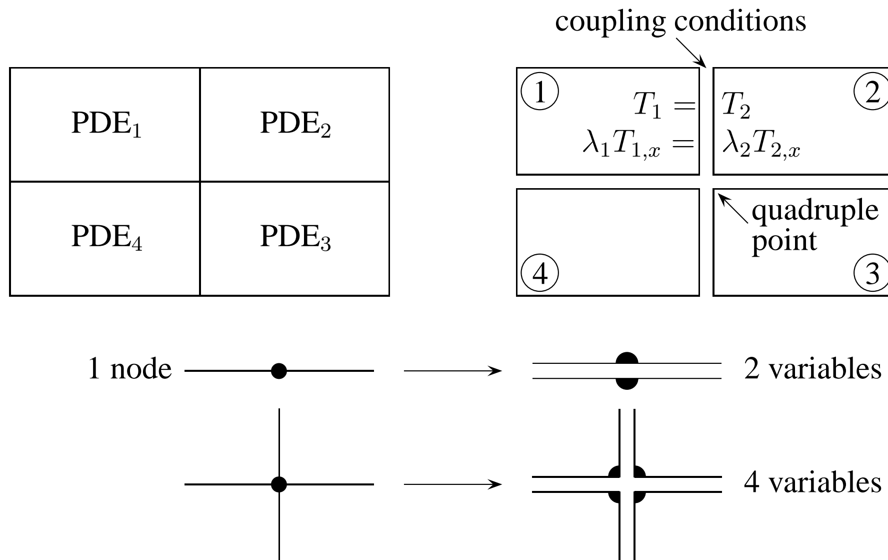


Figure 3.5.2: Illustration of heat flux problem.

coupling conditions for a quadruple point:

$$\begin{aligned}
\underline{1}, 2: &\quad T_1 - T_2 = 0 \\
\underline{2}, 3: &\quad T_2 - T_3 = 0 \\
\underline{3}, 4: &\quad T_3 - T_4 = 0 \\
\underline{4}, 1: &\quad T_4 - T_1 = 0.
\end{aligned}$$

The reason is that there is no unique solution because the coupling conditions are linear dependent. So we have to change at least one of the coupling conditions, for example we change the last one into

$$\underline{4}, 1: \quad \lambda_4 T_{4,y} - \lambda_1 T_{1,y} = 0.$$

The coupling conditions need values of both sides of the dividing line as we can see in figure 3.5.2. Here we need the temperature of the first and the second domain for the first coupling condition and the same goes for the derivative $T_{k,x}$ that we need for the second coupling condition.

As we cannot differentiate across a dividing line we use one-sided difference stars at the dividing lines that use function values of the corresponding subdomain. Thus the dividing lines are treated as “interior” boundaries. As a mesh generator delivers a configuration as shown on the left side of figure 3.5.2, at the beginning of the solution process the new variables must be generated so that the logical configuration at the right of figure 3.5.2 results. Here we have a grid that goes straight through the whole domain, i.e. we have matching

grid on both sides of the dividing lines.

One geometrical dividing line node produces n_{dl} logical dividing line nodes and each of these n_{dl} logical dividing line nodes produces l rows in the large sparse matrix Q_d. Each of these rows consists of two parts. One that is computed from the difference formulas and the Jacobian matrices from the own node and the other part is computed from the difference formulas and the Jacobian matrices of the opposite twin node.

3.5.2 Sliding dividing lines

In practical applications the subdomains may slide relatively to each other and they may need quite different grids, see figure 3.5.3.

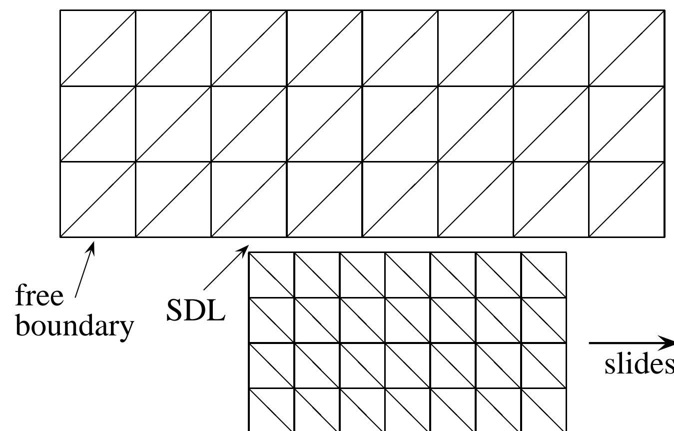


Figure 3.5.3: Illustration for sliding dividing line (SDL).

Here the nodes of the lower boundary of the upper domain and the nodes of the upper boundary of the lower domain may be free or coupling boundary nodes. Such an interface is called sliding dividing line (SDL). The problem is that we do not have two or more coupled nodes that result from one geometrical node like for the dividing lines with the matching grid. So how can we couple the solutions of the subdomains across a sliding dividing line? The solution is illustrated in figure 3.5.4. A geometrical node on a sliding dividing line generates a fictitious opposite node on the other subdomain. However, for a finite difference method function values and derivatives are only known at the geometrical nodes. In order to get these values for the fictitious opposite node (e.g. node B in figure 3.5.4) we first search for the nearest geometrical node of B on the sliding dividing line which is node A. For this node A we know the coefficients of the influence polynomials. Now these polynomials are not only evaluated for x_A, y_A to get interpolation, difference and error formulas at A but also for x_B, y_B so that we get the coefficients of the difference

and error formulas for node B. These formulas for the fictitious opposite node B are used in the coupling conditions between the geometrical node and its fictitious twin node like for the matching dividing line nodes. Node B is an "artificial" matching node.

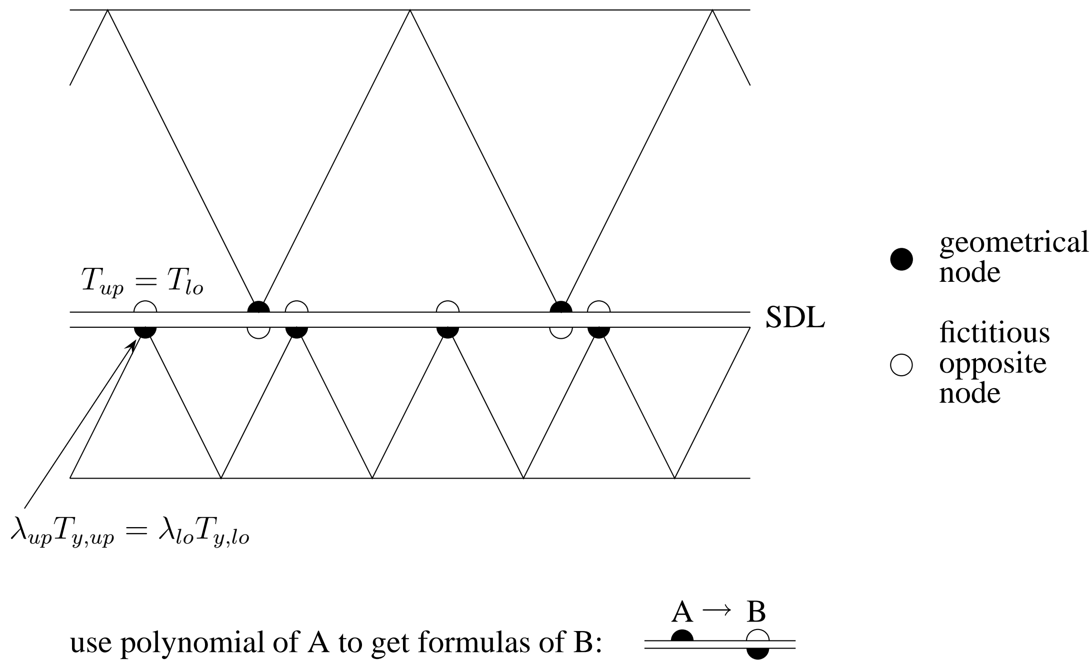


Figure 3.5.4: Illustration of the coupling across a sliding dividing line.

In contrast to the dividing lines where we have two equations for the two logical variables (see (3.5.1)) we have at a coupling node of a sliding dividing line only one variable for the geometrical node. The formulas of the fictitious opposite node contain the variables of the formulas of the nearest opposite geometrical node, but no new variable. Therefore we prescribe in the example of the heat conduction problem in figure 3.5.4 for the geometrical nodes of the upper domain equal temperature $T_{up} = T_{lo}$ and for the geometrical nodes of the lower domain equal heat flux $\lambda_{up}T_{y,up} = \lambda_{lo}T_{y,lo}$.

As already mentioned the nodes on the sliding dividing lines may be free boundary nodes or coupling boundary nodes. We will present now the 2-D algorithm to determine which node has which property. The exemplary problem is illustrated in figure 3.5.5.

We have one domain that slides from the left to the right and back below two static domains that are coupled by a dividing line. So there are sliding dividing lines on the lower

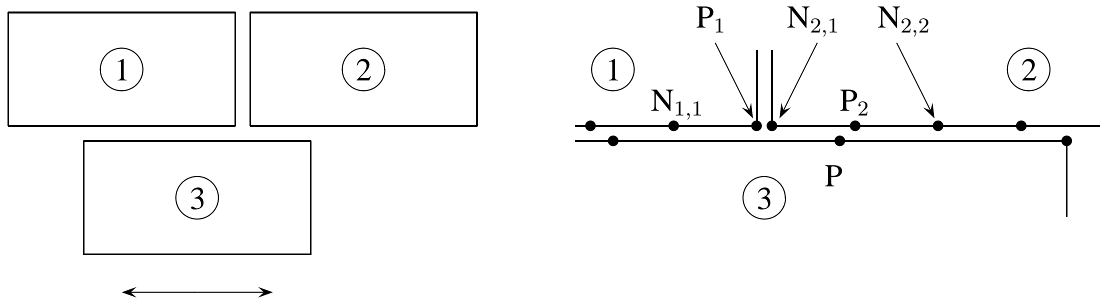


Figure 3.5.5: Illustration of algorithm for property free/coupling node.

boundaries of the two upper domains and on the upper boundary of the lower domain. We want to show how a node on the sliding dividing line on the lower domain gets the property free or coupling node. If it is a coupling node we have to know to which of the two upper domains the node couples.

So first we search for each sliding dividing line node of the lower domain for its next node on each sliding dividing line of each upper coupling domain. When we do this for the first time, we compute the distance to each node on the coupling sliding dividing line for the first node. Then we continue with the neighbours of the first node and begin the search of their next upper sliding dividing line nodes with the already found next nodes of the first node. We repeat this step until all sliding dividing line nodes have found their next nodes on all upper coupling sliding dividing lines. Now we must check which of the next nodes on the other domains is the next opposite node. In figure 3.5.5 the situation is illustrated: We search the next opposite node for node P on domain 3. We find a next node P_1 on domain 1 and a next node P_2 on domain 2. The neighbour nodes of the nodes P_i are denoted by $N_{i,j}$ where $N_{i,j}$ is the jth neighbour node on the sliding dividing line of node P_i on domain i. Now we check for each coupling domain i and each neighbour node $N_{i,j}$ if

$$\left|\overline{PN_{i,j}}\right| \leq \left|\overline{P_iN_{i,j}}\right| \tag{3.5.2}$$

holds. This means that the distance between node P and the neighbour node $N_{i,j}$ of the investigated node P_i is smaller than the distance between the investigated node P_i and its neighbour $N_{i,j}$, i.e. the position of node P is between these two nodes. This criterion can be fulfilled once at most. If it is fulfilled for a node P_i and one of its neighbours this node P_i is the fictitious opposite node and we can continue with the next sliding dividing line node. If it is not fulfilled for any of the nodes P_i the node P gets the property “free boundary node” because in this case is not between the investigated next nodes and their neighbour nodes on any of the coupling domains. In the next time step we start the search for the next nodes on the other sliding dividing lines at the next node of the current time step.

In figure 3.5.6 you can see an exemplary situation for three domains where we have sliding dividing lines between the two upper domains and the lower domain. As the mesh on the lower domain is coarser than those of the upper domains, there are some sliding dividing line nodes that have the same fictitious opposite node, especially on domain 1. The two sliding dividing line nodes at the left end of the sliding dividing line do not find a fictitious opposite node on one of the coupling sliding dividing lines as described above and therefore are free boundary nodes.

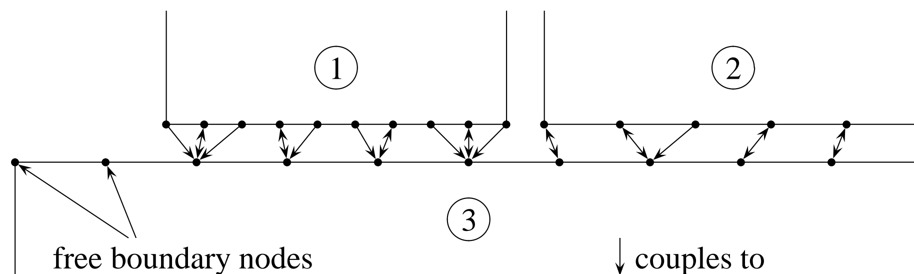


Figure 3.5.6: Illustration of property free/coupling node.

At the end of this subsection we repeat the properties of the mesh refinement at dividing lines and sliding dividing lines. In the case of dividing lines we always must have matching grids. So if we half an edge on a dividing line the corresponding edge on the other grid must also be halved. This means that the refinement cascade is continued across dividing lines and the refinement in a domain is not independent of the other domains. As we have non-matching grids for sliding dividing lines we can refine the domains separately from each other. The refinement on an edge of a sliding dividing line has no consequences for the grid on the other side and the refinement cascade stops at the domain border.

3.6 Parallelization

The numerical solution of partial differential equations needs much computation time and memory. These are the two main reasons for parallel computation. Especially in 3-D we have very large problems with many unknowns where the matrix Q_d needs much space and where the arrays, especially for computations with order control, for the coefficients of the difference and error formulas are even larger as shown in section 3.4 on page 35. Furthermore, the solution time increases. Therefore we need an efficiently parallelized program that is executed on a distributed memory parallel computer with message passing (MPI). Here the essential drawback is the communication. If we only use local data the execution time depends on the bandwidth and the latency of the cache hierarchy or of the much

slower memory. But if we need to access data from other processors, the bandwidth and the latency of the communication network are essential parameters.

Of course, it makes no sense to store all data locally on a distributed memory parallel computer, so the essential goal of the parallelization is the minimization of communication. Principally, you have two possibilities: the first is to exchange data exactly at the moment one processor needs data from another. Here the requesting processor has to wait until the data have arrived without doing anything but waiting. The second possibility is to exchange the data that is needed on each processor in a preparatory step, following the principle of the separation of selection and processing of the data. This means that the same data have to be stored on several processors, which means storage overhead, but on the other hand it avoids communication and therefore saves computation time, see [8]. The FDEM program package has been designed with this principle in mind.

As one of our goals was that FDEM should run on all types of parallel computers with shared and distributed memory, the only possible choice is message passing. Therefore we use the quasi-standard MPI, because it has been demonstrated in many examples that MPI is more efficient than the shared memory quasi-standard OpenMP even if there is a global address space over the distributed memory.

on processor

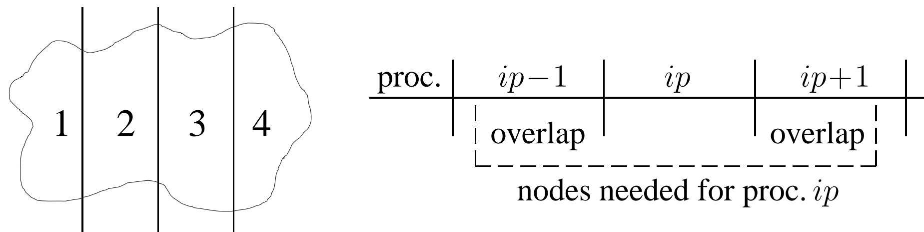


Figure 3.6.1: Illustration of the distribution of the data to the processors.

At the start of the program the following information is given by the (commercial) mesh generator: We have an element array where for each global element are given the global node numbers that are contained in this element. In the coordinate array there are the values of the coordinates for each global node and in the boundary node array(s) there is stored which nodes are external (and internal) boundary nodes and on which boundary they are. For the generation of the difference and error formulas we need rings of neighboured nodes. Therefore we sort the nodes by their x-coordinate, at first locally on each of the np processors, then globally by a special algorithm by which up to $np/2$ processors are

active in parallel. By the sorting the nodes are distributed with increasing x-coordinate in np (nearly) equal parts on the np processors. This results in an automatic one-dimensional domain decomposition, see figure 3.6.1. Note that FDEM is also a black-box solver with respect to the domain: We do not know which 2-D or 3-D domain the user will deliver. The same sorting takes place after each mesh refinement to guarantee an efficient load balancing.

Now we sort the elements on the processors. An element belongs to a processor if it has a node on this processor. If an element has nodes on two processors it always belongs to the left one which is shown in figure 3.6.2. Here we make use of the "basket principle": Each processor puts its old, static information in a basket that is sent around in a nearest neighbour ring in np tacts. Here the np processors are conceptually treated as a linear arrangement with wrap-around, or more precisely as a ring. In each tact the processors take out of the current basket the information they need, see figure 3.6.3 for $np = 4$ processors.

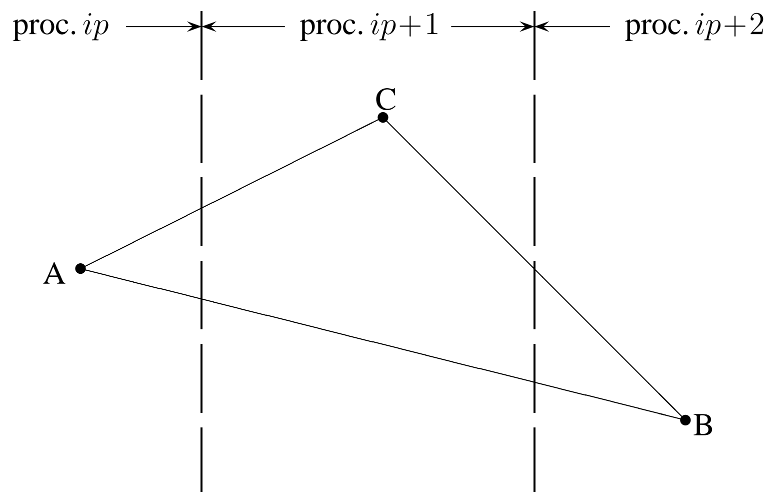


Figure 3.6.2: Illustration for the owning of triangles.

For the exchange of data we have in many cases the problem that the receiving processor of the MPI messages must have the information about the length of the data they receive in advance. As this information often is unknown the processors must exchange it in a preparing step and afterwards exchange the real data.

As we mentioned before, we want to execute the generation and evaluation of the difference and error formulas, the generation of the right hand side $(Pu)_d$ and of the large sparse matrix Q_d purely local on the processors, without communication. So we must store on

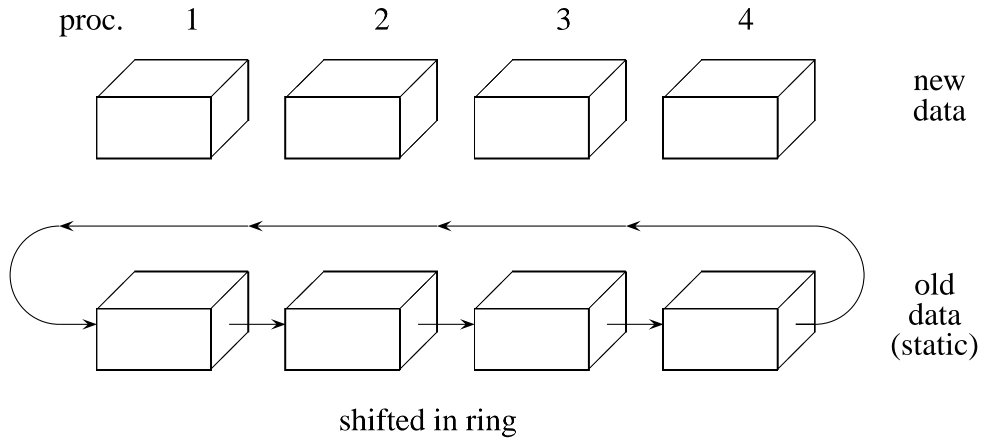


Figure 3.6.3: Illustration of the basket principle for $np = 4$ processors.

each processor also the data of the adjacent processor(s) that are needed for the rings of neighboured nodes. This is indicated as overlap data (see figure 3.6.1). Note that an overlap may extend over several processors. In order to determine the width of the overlap we first compute the average edge length h_{mean} of the elements. On page 26 in section 3.3 we explained the two possible ring limits. We want to form the formula for the width of the overlap if the first ring limit is decisive. If we choose the consistency order q for the computation of the solution, we must collect nodes up to order $\bar{q} = q + \Delta q$ for the error formulas. Furthermore, we assume that we have a mesh consisting of squares, see figue 3.6.4. With the help of this mesh we can estimate the necessary overlap for a triangular mesh as the overlap for this mesh is wider than for a triangular mesh.

We must now compute how many rings of squares we need to collect $m(\bar{q})$ nodes. For one ring we have nine nodes, for two rings we have 25 nodes and for $\tilde{x}$ rings we have $(2\tilde{x}+1)^2$ nodes. So we set

$$m(\bar{q}) = (2\tilde{x}+1)^2 \tag{3.6.1}$$

and therefore yield

$$\tilde{x} = \frac{1}{2}\left(\sqrt{m(\bar{q})} - 1\right). \tag{3.6.2}$$

To be on the safe side we use a safety factor $a_{overlap} \geq 1$ and with the average edge length h_{mean} we compute the width of the overlap by

$$x_{overlap,1} = \frac{1}{2} \cdot a_{overlap} \cdot h_{mean} \cdot \left(\sqrt[dim]{m(q+\Delta q)} - 1 \right) \tag{3.6.3}$$

where dim denotes the dimension of the computational domain, i.e. the formula is also valid for 3-D. This is because in 3-D it holds

$$m(\bar{q}) = (2\tilde{x}+1)^3. \tag{3.6.4}$$

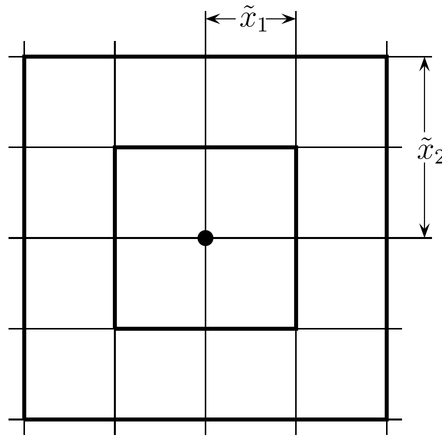


Figure 3.6.4: Illustration of rings for a rectangular mesh.

If the second ring limit is decisive, the overlap width is

$$x_{overlap,2} = a_{overlap} \cdot h_{mean} \cdot (q + \Delta q) \,. \tag{3.6.5}$$

Here the formula is easier to explain. We need $q + \Delta q$ rings and each ring has an average width of h_{mean}, so that we get formula (3.6.5) when we use the safety factor $a_{overlap}$.

Now we compute $x_{overlap}$ which is the maximum of $x_{overlap,1}$ and $x_{overlap,2}$ and then store the nodes of the left and right processors

$$\text{from} \quad x_{left} = x_{min} - x_{overlap} \tag{3.6.6}$$
$$\text{to} \quad x_{right} = x_{max} + x_{overlap}. \tag{3.6.7}$$

There is an array where for each processor is stored the information about the x-coordinates of its own leftmost and rightmost node x_{min} and x_{max}. This array is available on each processor so that it knows which data are stored on which processor. By this knowledge each processor knows with how many processors it has to exchange data on the left and on the right side. The transfer of the overlap data is made in such a way that the whole data of these concerned processors is stored on the own processor and then superfluous data (of the last overlap processors on the left and right side) is deleted.

After the overlap of the node data is created, the element overlap is generated. This is done in the same way we distributed the elements to their own processors.

The solution process of the partial differential equations starts with the data distributed onto the processors where on each processor a local numbering over all own and overlap data

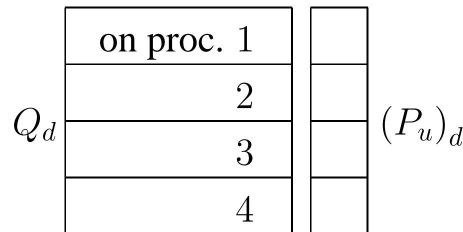


Figure 3.6.5: Distribution of the matrix Q_d and the r.h.s. $(Pu)_d$ in row blocks to $np = 4$ processors.

is used. So each processor can compute its part of the matrix Q_d (see Fingure 3.1.1) and the right hand side $(Pu)_d$ (see (3.1.11)) completely independent of the other processors without communication as if it would be a single processor and not only one processor in a parallel computer, see figure 3.6.5 for $np = 4$ processors.

The key for this seemingly quite simple procedure is the overlap and the local numbering. It is clear that after each Newton step the values for u_d for the overlap nodes must be exchanged between the processors.

3.7 Remarks to LINSOL

LINSOL (LINear equation SOLver) is responsible for the iterative solution of the large sparse linear systems

$$Ax = b,\ A \in \mathbb{R}^{n\times n},\ x, b \in \mathbb{R}^n, n \in \mathbb{N} \tag{3.7.1}$$

which arises in the FDEM computations (see (3.1.11)). LINSOL has been developed together with the previous PDE solvers FIDISOL [5], Chapter 17 and CADSOL [9] and has been enhanced continuously since that time. It is now publicly available, see [10] and the references given there.

After a processor has computed its part of the linear system of equations, see figure 3.6.5, LINSOL is called independently so that the processors are automatically synchronized by the data.

To solve such large systems of linear equations we require two properties of the linear solver: robustness and efficiency. Because of reasons of storage limitations LINSOL originally was a purely iterative solver with different types of CG (conjugate gradient) methods (presently implemented: 14 CG methods). With these methods we generate several polyalgorithms with automatic method switching from very efficient but less robust to less efficient but highly robust methods. We switch to the next method if the convergence of the

currently used method is not satisfying any more.

As many of our technical problems could not be solved efficiently by pure CG methods we developed a very sophisticated parallelized (I)LU preconditioning [11] (this reference can be accessed via [10], documentation). For full LU preconditioning one has a direct solver with automatic post-correction. LINSOL also has three efficient bandwidth optimizers [12] which are indispensable especially for 3-D problems.

The matrix Q_d may be stored in eight different data structures: full and packed diagonals, rows and columns, main diagonal and starry sky. It is possible to use several data structures in the same matrix; the matrix is composed by these basic elements and an information array gives the necessary information. For FDEM we only use packed rows. The data structures are split up into row and column blocks to support an efficient parallelization of the matrix-vector multiplication [8] which is the kernel operation of all iterative solvers. For the (I)LU factorization we use a single wrap-around over the processors with an active buffer window for efficient load balancing. The factorized parts L and U then are reorganized in packed rows and columns for an efficient forward elimination and backward substitution, again following the principle of the separation of selection and processing of the data.

4 The algorithm of the mesh refinement

The mesh refinement consists of two main parts. The first part is this: We determine all elements that have to be refined, either because of the error or because of the refinement cascade. Therefore we check for each node if (3.4.15) holds and if not, all elements that contain this node become refinement elements (see section 4.2). Then we start the refinement cascade (see section 4.3) whereby also elements become refinement elements that would get a fourth node on one of their edges by the refinement of a neighbour element.

The second part is this: We begin with the largest refinement elements (refinement stage 0) and generate new nodes on the mid-points of the edges. Here we must pay attention to avoid multiple generation of the same physical node. The new nodes must also be stored in the neighbour elements. The function values of the new nodes are interpolated by the interpolation formulas of the two end points of the edge, the order q is the minimum of the orders of the two end points. Then we check which of the new nodes are external boundary nodes, dividing line or sliding dividing line nodes. Now we generate the new elements, one element gets the old element number, the other ones get new global element numbers. This is explained for a single processor in section 4.4 and for distributed memory parallel computers in section 4.5. Then we do the same for the second largest elements (refinement stage 1), then for the third largest elements etc. We call the refinement of the elements of a certain refinement stage a refinement step.

On a distributed memory parallel computer we have to execute after each refinement step an intermediate step to update the element information on the processors before we can continue with the elements of the next refinement stage.

4.1 Context

Here we consider the basics that are used throughout the mesh refinement algorithm. The realization of the communication is discussed as well as frequently used communication patterns and procedures that we adopt from the LINSOL program package.

4.1.1 Implementation language

The FDEM program package is implemented in Fortran 90. Fortran has been designed for efficient scientific computing. Although there are many people saying Fortran was a dead language and ennoble functional languages, there is not any alternative concerning efficiency and performance on a distributed memory parallel computer at the moment. Therefore the LINSOL package is also implemented in Fortran 90.

4.1.2 The Portable Message Passing Interface

Generally, we can categorize programming models by how memory is used. In the distributed global shared memory model each process accesses a shared address space, while in the message passing model an application runs as a collection of autonomous processes, each with its own local memory. In the message passing model processes communicate with other processes by sending and receiving messages. When data is passed in a message, the sending and receiving processes must operate to transfer the data from the local memory of one processor to the local memory of the other.

Message passing is used widely on parallel computers with distributed memory and on clusters of servers. Besides the high efficiency the advantages of using message passing include portability and universality as message passing is implemented on most parallel platforms and the model makes minimal assumptions about underlying parallel hardware. However, creating message-passing applications may require more effort than letting a parallelizing compiler produce parallel applications.

In section 3.6 we already mentioned that the concept for the parallelization we have chosen for FDEM is the message passing paradigm. This has been done for reasons of efficiency—the virtual shared memory programming model is easier to program but it is never as efficient as using explicit message passing—and because FDEM should be portable to a high degree. Message passing libraries exist on shared and distributed memory multiprocessors but these libraries did not have a common subset of basic message passing routines in the early days of MPI. So you had to adapt your routines to the message passing library for each desired target system. As this made the code hardly portable we have developed the portable message passing interface (P_MPI) in our research group, see [10], that we still use although it was not necessary nowadays and that we want to explain shortly in the following. Additionally, P_MPI is an essential simplification compared to MPI as it uses only the small subset of operations that we need for FDEM.

P_MPI, in contrast to MPI or PVM, does not support different types of send and receive, no group functions and only one kind of collective communication which is *gather* and *scatter*, but as this is not used in the refinement process we do not deal with this any further. Apart from that we only provide interfaces for six important basic message passing routines—four point-to-point communication routines plus a start-of-communciation and an end-of-communication routine—with parameters that are common to all usual message passing interfaces. The interface is written entirely in Fortran 77.

So P_MPI contains the subroutines `COMBGN` for the initialization and `COMEND` for the closure of the communication. `COMBGN` returns some important communication parameters to the user such as the number of processors np and a one-dimensional integer array *tids* that

i	1	2	3	$\cdots$	i	$\cdots$	np	logical number
$tids(i)$	0	1	2	$\cdots$	$i-1$	$\cdots$	$np-1$	physical number

Table 4.1.1: Array *tids* with physical processor numbers.

contains the physical processor numbers for each logical processor, see table 4.1.1. This array *tids* is an input parameter for the send and receive routines.

MPSNDA(tid_{rcv}, $type_{msg}$, l_{msg}, msg, id_{msg}, err)
MPSNDW(tid_{rcv}, $type_{msg}$, l_{msg}, msg, id_{msg}, err)
MPRCVA(tid_{snd}, $type_{msg}$, l_{msg}, msg, id_{msg}, err)
MPRCVW(tid_{snd}, $type_{msg}$, l_{msg}, msg, id_{msg}, err)

parameter	type	property	meaning
tid_{rcv}	I	in	physical processor number of the receiving processor
tid_{snd}	I	in	physical processor number of the sending processor
$type_{msg}$	I	in	message type
l_{msg}	I	in	length of the message (in bytes)
msg	I		starting address of the first element of the message of
	R		length l_{msg} that may be of type I, R, *DP* or $C{*}1$
	DP	in	input parameter for MPSNDA and MPSNDW
	$C{*}1$	out	output parameter for MPRCVA and MPRCVW
			array: $msg(l_{msg})$
id_{msg}	I		message id (identifies matching start and wait routines)
		in	input parameter for MPSNDA and MPSNDW
		out	output parameter for MPRCVA and MPRCVW
err	I	out	error number

I: integer, R: real, *DP*: double precision, $C*1$: character

Table 4.1.2: Parameter list of the communication routines.

These communication routines are MPSNDA, MPSNDW, MPRCVA and MPRCVW. The nonblocking send call MPSNDA indicates that the system may start to copy data out of the send buffer and the nonblocking MPRCVA indicates that the system may start to write data into the receive buffer. To complete a nonblocking communication we use the routines MPSNDW and MPRCVW that have the functionality of waits. These four communication routines all have the same parameter list, see table 4.1.2.

As the send and receive routines are asynchronous, we are able to proceed with the pro-

gram execution while the messages are exchanged. Theoretically, we can hide the communication overhead completely behind the computational effort if the computer provides autonomous communication units—process communication can overlap process computation. An example of such a computer was the Fujitsu VPP with its autonomous DTU (Data Transfer Unit) and crossbar switch.

4.1.3 External procedures

We use five external procedures from the LINSOL core package in the mesh refinement algorithm. The first two routines `LL4INM` and `LL4RNM` compute a global maximum of integer vectors or double precision vectors, respectively. The third one (`LL4ISM`) computes a global sum of integer vectors.

`LL4INM`	global integer maximum
`LL4RNM`	global double precision maximum
`LL4ISM`	global integer sum
`HEAPSORTIX`	heap sort
`SECONDS`	get CPU time and wall clock time

Table 4.1.3: List of external procedures.

These routines first compute the local result of the operation, e.g. the maximum of their input vector, and afterwards the global result is computed by the means of the P_MPI routines of subsection 4.1.2. The routines are symmetric, i.e. if the completion is successful each processor has got the result. We often use these routines to compute intermediate results which must be available on each processor, e.g. a maximum number in order to be able to allocate a send or receive buffer with the necessary length.

If we exchange node or element numbers between processors, this can only be done by global numbers. So the receiving processors must transform these global numbers to their own local numbers to process the received data. This transformation is done locally by a binary search in the array of global node or element numbers. But a binary search is only possible if the nodes and elements are sorted by ascending numbers. So we first have to sort the nodes and elements for what we use the routine `HEAPSORTIX`.

Finally, routine `SECONDS` is used for measuring the consumed CPU time and wall clock time.

4.1.4 Communication patterns

Here we present two communication patterns that are frequently occurring in the implementations. As each processor is processing the refinement of its elements independently,

the processors have no information about the activities of the neighbour processors. So when we come to a point where the processors have to exchange data with some of their neighbour processors, the target processor has no information about the length of the message it will receive from which processor and, even worse, the processors do not have the information if they will receive a message at all.

So one possibility was to send a message with a global length that is computed by the maximum length of the data that any processor has to send. Then we would send messages with this length to all overlap processors. But usually only data at the processor borders, i.e. only a small part of the data a processor owns, is concerned. Thus, firstly we would waste time by sending superfluous data to neighbour processors. There we even waste time twice because the time for sending the message increases and then the target processor must extract the data from the received message it is really in need of. And secondly we would waste even more time by sending superfluous messages to overlap processors that will fail to extract useful data from the message afterwards.

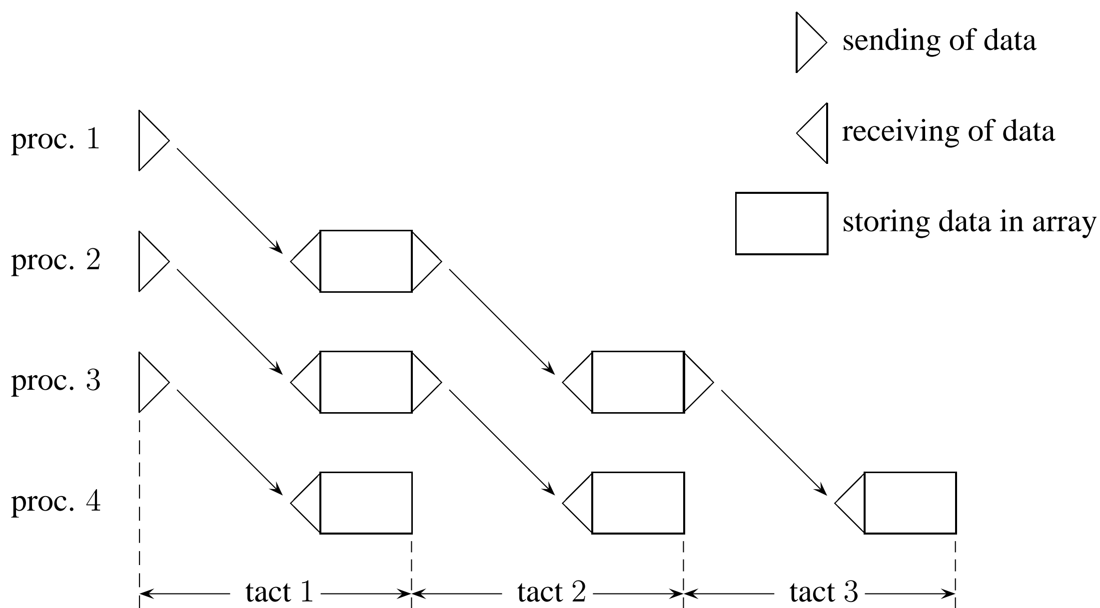


Figure 4.1.1: Illustration of the passing of the message lengths to the right for $np = 4$ processors and $np_{max,r} = 3$.

So we split the communication into two parts. In the first part we only send the lengths of the messages, the real data follow in the second part. This first part consists of $np_{max,r}$ cycles for the passing of the messages to the right and $np_{max,l}$ for the passing of the messages to the left. Here, $np_{max,r}$ and $np_{max,l}$ are the maximum number of overlap processors on the right side and on the left side, respectively. The passing of the message lengths

in $np_{max,r} = 3$ tacts to the right is illustrated in figure 4.1.1 for $np = 4$ processors. At the beginning each processor stores the message lengths (number of elements to be sent) of the messages it must pass to the neighbour processors in a send buffer *sndbuf* with length (number of elements) equal to l_{sndbuf}. In $sndbuf(1)$ we store the message length for the direct neighbour processor, in $sndbuf(2)$ the message length for the second neighbour processor and so on, see table 4.1.4.

	send to right		send to left	
i	$sndbuf(i)$	target proc.	$sndbuf(i)$	target proc.
1	17	$ip+1$	9	$ip-1$
2	13	$ip+2$	0	$ip-2$
3	1	$ip+3$	2	$ip-3$

Table 4.1.4: Example of storage scheme of send buffer *sndbuf*, i is the cycle number.

If the messages must be sent to the right all processors

$$\text{from} \quad ip = i$$
$$\text{to} \quad ip = np - 1$$

send their current buffer *sndbuf* to their direct right neighbour processor in each cycle i. These messages have the length (in bytes)

$$\ell = l_{sndbuf} \cdot mem_{int} \tag{4.1.1}$$

where mem_{int} is the memory requirement for an integer variable. The processors

$$\text{from} \quad ip = i + 1$$
$$\text{to} \quad ip = np$$

are waiting for a message from their direct left neighbour processor that is written into the buffer *rcvbuf*. The length of *rcvbuf* must be greater or equal to l_{sndbuf}. Usually, l_{sndbuf} is computed by

$$l_{sndbuf} = np_{max,r} \tag{4.1.2}$$

where

$$np_{max,r} = \max_{ip=1,\ldots,np} np_{sr,ip} \tag{4.1.3}$$

and $np_{sr,ip}$ is the number of overlap processors on the right side (index r) to which processor ip may have to send information, i.e. $np_{max,r}$ is a global maximum. After having received the message the processors take out the relevant data and in the next cycle the just received message is sent to the right neighbour processor. The relevant information in

the received message of cycle i is $sndbuf(i)$ because in the first cycle the first right neighbour processor receives the data, in the second cycle the second right neighbour processor receives the data etc. In the later communication step where the data is sent, we save communication by omitting the exchange of information if the length of a message is zero, e.g. in 3-D it may be the situation that the refinement elements in the overlap are only on the direct neighbour processors. Then we only send messages to these processors and skip sending messages to the other overlap processors. The code for this communication pattern is shown in listing 1 for the passing of a single integer value to the right.

```
! tp: physical number of target processor
tp = tids(myproc+1)
! sp: physical number of source processor
sp = tids(myproc-1)

! for each communication cycle
do i = 1,np_maxr-1
  ! processors from ip=i to ip=np-1 send
  ! integer variable ival_snd to tp
  if ((myproc >= i).and.(myproc < np)) then
    call MPSNDA(tp,nmsg+myproc,iint,ival_snd,mids,ierr)
    call MPSNDW(tp,nmsg+myproc,iint,ival_snd,mids,ierr)
  end if
  ! processors from ip=i+1 to ip=np receive
  ! integer variable from sp in ival_rcv
  if (myproc > i) then
    call MPRCVA(sp,nmsg+sp+1,iint,ival_rcv,midr,ierr)
    call MPRCVW(sp,nmsg+sp+1,iint,ival_rcv,midr,ierr)
    { processing of ival_rcv }
    ! set ival_snd for next communication cycle
    ival_snd = ival_rcv
  end if
end do
! increase message counter nmsg
nmsg = nmsg+np
```

Listing 1: Code for communication pattern used to send message lengths to the right.

If we have to send data to the left overlap processors all processors

$$\begin{aligned} \text{from} \quad & ip = 2 \\ \text{to} \quad & ip = np + 1 - i \end{aligned}$$

send their current buffer *sndbuf* to their left neighbour processor in each cycle i. The maximum number of overlap processors $np_{max,l}$ on the left side (index l) by which we compute the buffer length l_{sndbuf} is computed by

$$np_{max,l} = \max_{ip=1,\ldots,np} np_{sl,ip} \tag{4.1.4}$$

where $np_{sl,ip}$ is the number of overlap processors on the left side to which processor ip may have to send information, i.e. $np_{max,l}$ is a global maximum like $np_{max,r}$. The passing of the message lengths to the left is illustrated in figure 4.1.2 again for $np = 4$ processors but this time the maximum number of overlap processor is 2, i.e. it holds $np_{max,l} = 2$. The similarity of figure 4.1.1 and 4.1.2 is palpable, only the processor numbers on the left side are changed. So this part of the communication can be done by the same procedure regardless of the direction of the communication.

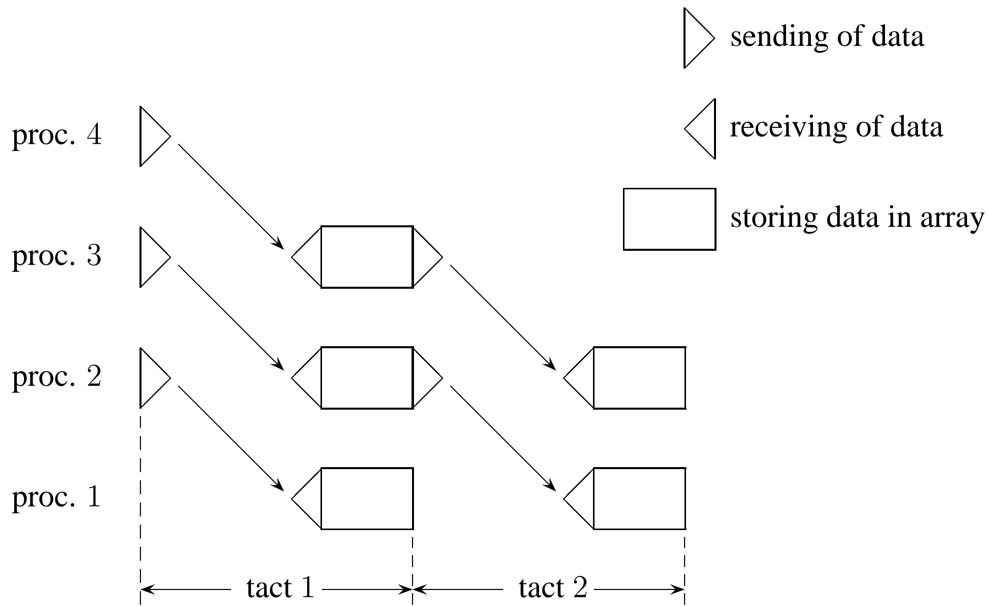


Figure 4.1.2: Illustration of the passing of the message lengths to the left for $np = 4$ processors and $np_{max,l} = 2$.

After this preparatory step the information is available on each processor which of its neighbour processors will send messages and, equally important, the length of each message it will receive is also known. So the second part of the communication can start where the real data is passed to the neighbour processors. This part consists of

$$\left.\begin{array}{c} np_{max,r} \\ np_{max,l} \end{array}\right\} \text{cycles for sending to the} \left\{\begin{array}{c} \text{right} \\ \text{left.} \end{array}\right.$$

We begin with the communication to the direct neighbour processor and continue with the next neighbour processor until the communication with the last overlap processor is finished. For the communication to the right, processor ip sends the message for processor $ip + i$ to this processor in communication cycle i if

$$ip \leq np - i \quad \wedge \quad \ell_{msg} > 0 \tag{4.1.5}$$

receiving processor

	1	2	3	4	5	6	7	8
1		R1	R2	R3				
2	L1		R1	R2	R3			
3	L2	L1		R1	R2	R3		
4	L3	L2	L1		R1	R2	R3	
5		L3	L2	L1		R1	R2	R3
6			L3	L2	L1		R1	R2
7				L3	L2	L1		R1
8					L3	L2	L1	

sending processor

Figure 4.1.3: Illustration of the communication pattern for $np = 8$ processors and $np_{max,r} = np_{max,l} = 3$.

holds. All processors with

$$ip > i \quad \wedge \quad \ell_{msg} > 0 \tag{4.1.6}$$

become the target processors for this cycle. The message length ℓ_{msg} is known on the target processor by the formerly introduced communication pattern. The code for the communication pattern is shown in listing 2 for the passing of messages to the right.

For the communication to the left, processor ip sends the message for processor $ip - i$ to this processor in communication cycle i if

$$ip > i \quad \wedge \quad \ell_{msg} > 0 \tag{4.1.7}$$

holds. The processors with

$$ip \leq np - i \quad \wedge \quad \ell_{msg} > 0 \tag{4.1.8}$$

are ready to receive a message from processor $ip - i$. The communication pattern for this second step of the communication is illustrated in figure 4.1.3 for $np = 8$ processors.

```
! set message counter for comm. to global message counter
msgcnt = nmsg
! for each right overlap processor
do i = 1,np_maxr
  ! increase message counter
  msgcnt = msgcnt+1

  ! tp/sp: physical number of target/source processor
  tp = tids(myproc+i)
  sp = tids(myproc-i)

  ! processors with target proc.<=np and message length>0
  ! send data in sndbuf to the target processor
  if ((myproc <= np-i).and.(sndcnt(i) > 0)) then
    ! l_msg: message length in bytes
    l_msg = sndcnt(i)*mem_int
    call MPSNDA(tp,msgcnt,l_msg,sndbuf,mids,ierr)
    call MPSNDW(tp,msgcnt,l_msg,sndbuf,mids,ierr)
  end if

  ! processors with source proc.>i and message length>0
  ! receive data in rcvbuf from the source processor
  if ((myproc > i).and.(rcvcnt(i) > 0)) then
    ! l_msg: message length in bytes
    l_msg = rcvcnt(i)*mem_int
    call MPRCVA(sp,msgcnt,l_msg,rcvbuf,midr,ierr)
    call MPRCVW(sp,msgcnt,l_msg,rcvbuf,midr,ierr)
  end if
end do
! set global message counter to message counter for comm.
nmsg = msgcnt
```

Listing 2: Code for communication pattern used to send messages to the right.

The letter "L" means sending to the left, "R" stands for sending to the right and the numbers 1 to 3 are the cycle numbers. So if we send to the right, processor 3 passes the data for processor 5 in the second cycle ("R2"), for example.

This two-stage execution of the communication is the key for an optimal data exchange in this environment: After getting knowledge of the number of communication tacts by computing the global maximum of the number of overlap processors $np_{max,l}$ and $np_{max,r}$, the processors first get the information which data they will receive from which processor. Afterwards the exchange of the real data will take place with minimal communication overhead.

In this kind of communication is implied another important characteristic concerning performance on distributed memory parallel computers. Before the data exchange we collect all data that must be passed to an overlap processor and try to send as few messages as possible because the performance would suffer badly if we exchanged the data for each node or element individually by a separate message because of the startup time, the time until the communication network is ready to send the first byte to the target processor.

Remark: Strictly speaking, the term processor is different from the term process. A processor is a hardware unit of a computer on which one or several processes can be executed, and a process is an instance of a program executing autonomously. However, assuming the most common situation where only one process within a parallel computation is executed on one processor, we loosely use these two terms interchangeably in the rest of this chapter.

4.2 Refinement nodes and elements

At the beginning of each section or, if necessary, each subsection we introduce the most important notations, that you must remember to understand the following text, in a box.

Important notations:

nekinv: integer array for elements a node belongs to
refel: logical array for refinement elements
refpt: integer array for refinement node numbers

SUMMARY: We refine the mesh by dividing the elements into four elements in 2-D or eight elements in 3-D, respectively, by bisection of the edges. As we compute our solution and the error estimate for the nodes of the mesh, we must determine those nodes for which the error exceeds a given tolerance first, and from these refinement nodes we get the elements that have to be refined in the current cycle. Therefore we have to put up our first rule for the mesh refinement.

For the mesh refinement we first have to compute $tolg$ (3.4.4)—which is the tolerance on the level of equation, see on page 32—after the end of the Newton iteration as we saw in subsection 3.4.3. For (3.4.4) we need the relative norm of the corrections and the last defect of the Newton iteration. The value $tolg$ is compared to the space key error term $\|D_x + D_y + D_{xy}\|_i$ of each component i of a node. If for one of the components the accurcy check (3.4.15) does not hold, the node becomes a refinement node (because of the error). All refinement node numbers are stored in a one-dimensional **integer array *refpt*** of length n_l, see listing 3. n_l is the number of nodes owned by each processor, on a single processor it holds $n_l = n$ (total number of nodes). The "exit" in listing 3 means that we do not regard the remaining components if we already found one for which (3.4.15) does not hold, but we continue with the next node. This accuracy request is done by each processor

independently. If we want to refine the mesh locally but there are not any refinement nodes in a cycle, we reduce the safety factor s_{grid} by

$$s_{grid,new} = 0.1 \cdot s_{grid,old} \tag{4.2.1}$$

and again check (3.4.15) for each component of each node. This is repeated until there is at least one refinement node. Here you must remember that we go into the refinement process if the global relative error $\|\Delta u_d\|_{rel}$ (3.1.14) does not meet the given relative tolerance tol. If we do not find refinement nodes, we must reduce the corresponding tolerance $tolg$ by reducing the (empirical) safety factor s_{grid}, i.e. we must "sharpen" the accuracy request on the level of equation by (4.2.1).

```
! compute tolg on level of equation
tolg = tol*def_max/du_max

! initialize number of ref. nodes
nrofrp = 0
! for each local node
do i = 1,n_l
  ! for each component
  do j = 1,l
    ! compute absolute value of disc.err.term
    dis_l = dabs(dis((i-1)*l+j))

    ! check if node is ref. node
    if (dis_l > s_grid*tolg) then
      ! increase number of ref. nodes
      nrofrp = nrofrp+1
      ! insert node into refpt
      refpt(nrofrp) = i
      ! continue with next node
      exit
    end if
  end do
end do
! insert number of ref.nnodes into refpt
refpt(n_l+1) = nrofrp
```

Listing 3: Code for determination of refinement nodes.

Now we go through the array of refinement nodes, and for each refinement node we get the element numbers it belongs to from the **integer array *nekinv***, i.e. the inverted **integer array *nek*** where we store the structure of the space, see section 3.3 on page 23. These elements are refinement elements, see rule 1.
For the determination of the refinement elements we use a **logical array *refel*** with length equal to the maximum number of elements $ne_{n,max}$ on a processor (including the overlap)

Rule 1
All elements that contain a refinement node have to be refined and therefore become refinement elements.

and width equal to the number of refinement steps that already have taken place, starting with 0. For a single processor it holds $ne_{n,max} = ne$. The array *refel* has the shape shown in table 4.2.1 and is initialized with *false*. We enter the value *true* into *refel* in the row of the local number of the refinement element and the column of its refinement stage. The code for this part is shown in listing 4. The refinement stage is 0 for the original elements from the mesh generator and is increased by one for each refinement of these elements. The array *refel* has the following purpose: A refinement node usually belongs to several elements. These elements may have still further refinement nodes. We have to care that these elements occur only once in the list of refinement elements. The refinement takes place in "refinement steps". A refinement step is the refinement of all refinement elements of a certain refinement stage, starting with the basic mesh of refinement stage 0. Therefore we must know separately the refinement elements for each refinement stage. This can be seen in *refel*.

element number	refinement stage	*refel*			
		stage 0	stage 1	stage 2	stage 3
1	1	false	true	false	false
2	0	true	false	false	false
3	3	false	false	false	true
4	2	false	false	false	false
⋮	⋮	⋮	⋮	⋮	⋮
ne_l	0	true	false	false	false

Table 4.2.1: Shape of the logical array *refel*. *true* means that the corresponding element in the corresponding refinement stage is a refinement element. A *true* can occur only in the refinement stage of the corresponding element. The element 4 is not refined.

Optionally, it is possible to refine the whole subdomain if at least one node of this subdomain is a refinement node. Then for each element in this subdomain a *true* is entered in the column that corresponds to its refinement stage.

At the end of this first step of the mesh refinement we have an array *refel* where all the refinement elements are marked by *true* so that all elements that have to be refined because of the error are known now. The refinement nodes that we had to determine first are of no

```
! initialize refel
refel = .false.

! for each ref. node
do j = 1,nrofrp
  ! ptnr: number of current ref. node
  ptnr = refpt(j)
  ! for each element ptnr belongs to
  do i = 1,inne(ptnr)
    ! elnr: number of current element
    elnr = nekinv(ptnr,i)
    ! rst: refinement stage of element
    rst  = refst(elnr)
    ! insert true in refel for element elnr
    refel(elnr,rst) = .true.
  end do
end do
```

Listing 4: Code for entering refinement elements because of the error into *refel*.

account for the further mesh refinement and therefore the array *refpt* is deallocated.

4.3 Refinement cascade

Now we want to explain the refinement cascade that has been already mentioned in subsection 3.4.3, in detail. We look for elements that have to be refined not because of the error but for reason of data storage scheme. For these elements we also enter a *true* in the logical array *refel* for the refinement elements. Here we want to recall: In order to have a clear and fixed storage scheme, we have for a triangle six nodes: the first three are the corner nodes (always present), the next (optional) three nodes are nodes in the mid of the edges that result from a refinement of neighbour elements. If there are no nodes on the edges, these storage locations remain empty. For a tetrahedron in 3-D we have four nodes for the corners and six possible nodes on the edges. The triangles in 2-D and tetrahedrons in 3-D of the initial grid always have 3 and 4 nodes, respectively. It should also be recalled that the triangles and tetrahedrons only serve for the structure of the space, i.e. to search for neighbour nodes.

4.3.1 Refinement cascade on a single processor

SUMMARY: Here we first introduce the refinement cascade on a single processor and put up our second rule that gives the reason for the refinement cascade. Afterwards we mention the difficulties that come from the dividing lines and explain the method to overcome them. At the end of this subsection we will see that the logical array *refel* for

the refinement elements is unsuitable for the further mesh refinement process so that we have to transform it.

Important notations:

dlote:	integer array for twin node information of dividing line edges
dloteadr:	integer array for starting addresses of dividing line elements in *dlote*
indrel:	integer array for refinement element numbers
narpl:	integer array for information about storage of data in *indrel*
nek:	integer array for node numbers of the elements
nekinv:	integer array for elements a node belongs to
refel:	logical array for refinement elements

On a single processor we only have to go once through the logical refinement element array *refel* for each refinement stage (beginning with the highest refinement stage). For each refinement element we examine the three edges in 2-D. Let the refinement stage of the actual element be s. If an edge already is subdivided by a third node we do not need to make any further investigations because then the neighbour elements of this edge must be of a higher refinement stage $s + 1$ (see figure 4.3.1a)). If there is no third node on the edge, the neighbour elements may be of the same refinement stage (see figure 4.3.1b)) or of refinement stage $s - 1$ (figure 4.3.1c)). In the latter case neighbour elements of this type must also be refined because else four nodes would be on an edge. The difference of the refinement stages of neighboured elements after refinement must be at most 1, see rule 2. The code for the execution of the refinement cascade is shown in listing 5.

Rule 2

The difference of the refinement stages of two neighboured elements must be at most 1*, i.e. the number of nodes on an edge of an element is limited to three.*

But how do we find the neighbour elements of a refinement element that are of refinement stage $s-1$, see figure 4.3.1c)? For each refinement element that must be refined because of the error we look for each edge if the neighbour element is of a lower refinement stage. For the current edge we look for the elements the first end point of this edge belongs to in the *nekinv* array where we store the elements a node belongs to. Then we look in the *nek* array, i.e. the array with the node numbers of each element, if these elements also contain the second end point of the edge. Here we must only look in the entries $elpt + 1, \ldots, elpt + n_{edge}$ for the nodes where $elpt$ is the number of corner nodes per element and n_{edge} is the number of edges per element. This is because the first $elpt$ nodes are the corner nodes and the second node must be the mid-point of an edge of the searched neighbour element, see fig-

ure 4.3.1c). Afterwards we change the roles the two nodes play and look in each element that contains the second end point if the first one is the mid-point of an edge in this element. All elements where both nodes are contained—one as end point, the other as mid-point of an edge—are neighbour elements of a lower refinement stage (of course, the own element is excluded), see figure 4.3.2. We search for the neighbour element of element 202 for the edge with the end points 11 and 102. We get the elements node 11 belongs to from the *nekinv* array. Then we look for these elements in the *nek* array if node 102 occurs in one of the columns 4 to 6 because there the numbers of the mid-points are stored. We can affirm this for element 21 and thus element 21 gets a *true* in the array *refel* and must be refined because of the cascade. In the elements node 102 belongs to we cannot find node 11 in the last 3 columns. So element 21 is the desired neighbour element. The code for the neighbour search is shown in listing 6. In 2-D there is only one neighbour element, in 3-D there may be many elements that contain the same edge.

For a problem with domains coupled by dividing lines the search for the neighbour elements is not that easy. For each dividing line edge we also have to search for the corresponding dividing line edge on the other side of the dividing line and have to store these edges because we need the information during the refinement process later.

If a refinement element of stage s is a dividing line element, i.e. at least one of its edges is a dividing line edge, we do not only examine if there are neighbour elements on this side of the dividing line, but we also look for neighbours on the other side because we always must have matching grids on both sides of the dividing line and therefore we have to insert the new nodes into the dividing line elements on both sides of the dividing line. By "this side" we denote the subdomain the refinement element belongs to, the "other side" is the coupling subdomain for the edge curently under consideration. Therefore we look if both end nodes of the examined edge are dividing line nodes on the same dividing line. If yes,

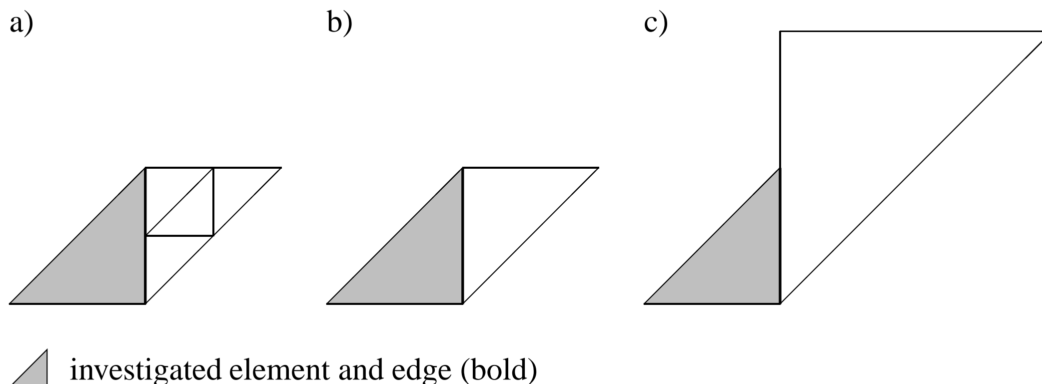


Figure 4.3.1: Illustration of neighboured elements: neighbour element of left element is of a) higher, b) same, c) lower refinement stage.

```
!**   elements to be refined because the difference of their
!**   refinement stage and that of one of their neighbour
!**   elements that has to be refined would be greater than 1
      ! initialize counter cnt
      cnt = 1

      ! for each refinement stage
      do s = rs_max,0,-1
        ! set starting address of current ref. stage
        narpl(s+1,1) = cnt

!**     for each refinement element look for the neighbour
!**     elements on those edges that have no middle point
        ! for each local element
        do i = 1,ne_l
          ! check if element is ref. element
          if (refel(i,1)) then
            ! insert element into indrel and increase cnt
            indrel(cnt) = i
            cnt = cnt+1

            ! for each mid-point
            do j = elpt+1,nolnod
              ! check if mid-point is nonexistent
              if (nek(i,j) == 0) then
                ! set end points of the current edge
                nod_1 = nek(i,hlpnek(j-elpt,1))
                nod_2 = nek(i,hlpnek(j-elpt,2))

                ! if edge in overlap increase counter notp
                if ((nod_1 > n_l).and.(nod_2 > n_l))
     &            notp = notp+1

                ! search for neighbour elem. of lower ref. stage
                call NEIGHB22(iinfo,nek,nenr,nekinv,inne,refel,
     &                        sent,sndrct,sndlct,sndrto,sndlto,
     &                        rcvr,nod_1,nod_2,s,nnr)
              end if
            end do
          end if
        end do

        ! set number of ref. elem. for current stage
        narpl(s+1,2) = cnt-narpl(s+1,1)
      end do
```

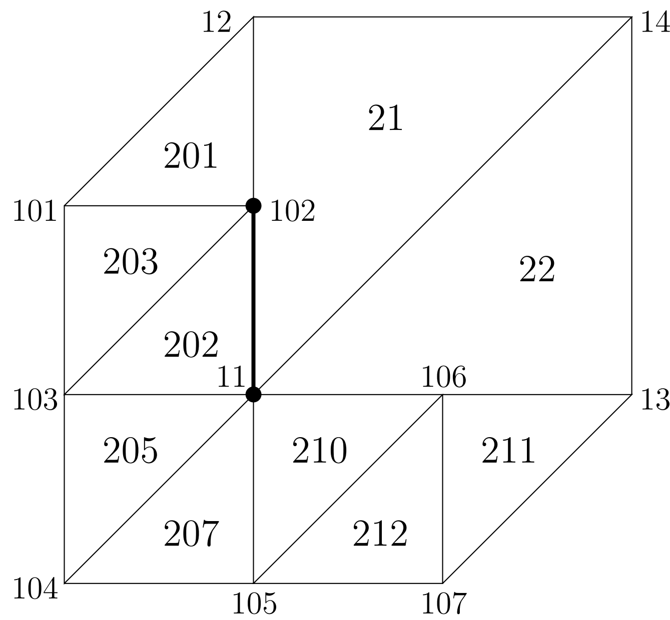
Listing 5: Code for determination of local refinement elements due to the cascade rules.

```
! initialize counter ctr
ctr = 0

! for each mid-point
do i2 = elpt+1,nolnod

  ! for each element node nod_1 belongs to
  do i1 = 1,inne(nod_1)
    ! elnr: number of current element
    elnr = nekinv(nod_1,i1)
    ! check if examined node of elem. elnr is nod_2
    if (nek(elnr,i2) == nod_2) then
      ! increase ctr and insert element into nbe
      ctr = ctr+1
      nbe(ctr) = elnr
    end if
  end do

  ! for each element node nod_2 belongs to
  do i1 = 1,inne(nod_2)
    ! elnr: number of current element
    elnr = nekinv(nod_2,i1)
    ! check if examined node of elem. elnr is nod_1
    if (nek(elnr,i2) == nod_1) then
      ! increase ctr and insert element into nbe
      ctr = ctr+1
      nbe(ctr) = elnr
    end if
  end do

end do
```

Listing 6: Code for neighbour element search.

we take their twin nodes on the other side of the dividing line and look for all neighbour elements that contain both of them. If there are elements of refinement stage $s - 1$ these elements also become refinement elements because of the refinement cascade and we enter a *true* into the corresponding row of the *refel* array in column $s - 1$. This is illustrated in figure 4.3.3.

As we need the neighbour elements on the other side of a dividing line later again, we also store the neighbour relations on a dividing line in an appropriate form. So we have to know for each element if it is a dividing line element at all, and if yes, which of its edges are dividing line edges, how many coupling subdomains there are and we need the numbers of the respective twin nodes. For the storage of this information we allocate two



$node$	$nekinv(node,k)$					
	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$
11	21	22	202	205	207	210
102	21	201	202	203	0	0

gives element numbers for node number

el	$nek(el,k)$					
	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$
21	11	12	14	102	0	0
22	11	14	13	0	0	106
201	12	102	101	0	0	0
202	11	103	102	0	0	0
203	101	102	103	0	0	0
205	11	104	103	0	0	0
207	11	105	104	0	0	0
210	11	106	105	0	0	0

gives node numbers for element number

Figure 4.3.2: Illustration of the search for neighbour elements of lower refinement stage.

one-dimensional **integer arrays** that are denoted by ***dloteadr*** and ***dlote***. The array *dloteadr* must have as much entries as we have elements, so its length is equal to ne where ne is the number of mesh triangles in 2-D. The computation of the length of array *dlote* is not that

easy as we do not want to waste storage:

$$l_{dlote} = n_{DL\,elem.} \cdot n_{edge} + n_{DL\,edges} \cdot (3 \cdot n_{CSD,max} + 1) \tag{4.3.1}$$

where $n_{CSD,max}$ is the maximum number of coupling subdomains a dividing line edge is coupling to and $n_{DL\,elem.}$ and $n_{DL\,edges}$ is the number of dividing line elements and dividing line edges, respectively. These values are known from the generation of the new logical dividing line nodes at the beginning of the program execution, see section 3.5. n_{edge} has already been introduced on page 65, it is the number of edges in an element. The formula for l_{dlote} is explained in the following when we describe the contents of the array *dlote*.

In *dloteadr* we store in row el the address $addr_{el}$ of the beginning of the data in *dlote* for element el:

$$dloteadr(el) = addr_{el}, \tag{4.3.2}$$

i.e. if an element el is not a dividing line element, it holds $dloteadr(el) = 0$.

For the understanding of the following, you should always have a look at figure 4.3.4. In the entries $addr_{el}, \ldots, addr_{el}+n_{edge}-1$ of *dlote* we again store addresses, this time the starting addresses for the data of edge $1, \ldots, n_{edge}$—that we denote by $addr_{edge\,1}$ to $addr_{edge\,n_{edge}}$—in *dlote* itself:

$$\begin{array}{lcl} dlote(addr_{el}) & = & addr_{edge\,1} \\ \vdots & & \vdots \\ dlote(addr_{el} + n_{edge} - 1) & = & addr_{edge\,n_{edge}}. \end{array} \tag{4.3.3}$$

This explains the first part of formula (4.3.1) as we need n_{edge} entries for each dividing line element in *dlote*. The real data for edge j of element el starts at $addr_{edge\,j}$ in *dlote*. Here

Figure 4.3.3: Illustration of the neighbour element search with dividing lines.

we enter the number of subdomains $n_{sect,j}$ edge j is coupling to:

$$dlote(addr_{edge\,j}) = n_{sect,j}. \tag{4.3.4}$$

In 2-D this number is always one, but in 3-D it may be any number, e.g. it is three for a cross of dividing surfaces. We need one entry for each dividing line edge to store this number in *dlote*, see (4.3.1). In $addr_{edge\,j}+3\cdot k-1$ and $addr_{edge\,j}+3\cdot k$ we store the numbers $nod_{1,j,k}$ and $nod_{2,j,k}$ of the twin nodes on the other side of the dividing line for the kth coupling subdomain:

$$dlote(addr_{edge\,j} + 3 \cdot k - 1) = nod_{1,j,k} \tag{4.3.5}$$
$$dlote(addr_{edge\,j} + 3 \cdot k) \qquad = nod_{2,j,k}. \tag{4.3.6}$$

At the moment, we leave free entry $addr_{edge\,j}+3\cdot k-2$, there we will store the number of elements that share edge j on the other side of the dividing line later (see on page 112). So we need another $3 \cdot n_{CSD}$ entries in *dlote* per dividing line edge with n_{CSD} being the number of coupling subdomains for the respective edge. For the allocation of *dlote* we need the length l_{dlote} in advance, so we simply compute the maximum of coupling subdomains $n_{CSD,max}$ by

$$n_{CSD,max} = \max_{DL\,edges} n_{CSD}. \tag{4.3.7}$$

The storage scheme of the arrays *dlote* and *dloteadr* is illustrated in figure 4.3.4 for the case of a cross of four dividing lines. Let element 100 be a refinement element. So in $dloteadr(100)$ we store the address of the beginning of the data for element 100 in *dlote* which is denoted by $addr_{100}$ here. In $dlote(addr_{100})$ to $dlote(addr_{100} + 2)$ we store the entry in *dlote* itself where the data for edge 1 to 3 begins. As edge 3 is not a part of any dividing line the value is not changed and remains zero. The edge data consists of the number of subdomains the edge is coupling to (in this case 1), the zero that reserves a location for the number of neighbour elements on the other side of the dividing line that is inserted later, and the two node numbers of the twin nodes, 7499 and 7500 for edge 2 and 2501 and 2551 for edge 3. As there is no data for edge 3 the data for the edges of the next dividing line element follows directly behind the data for edge 2 of element 100. The code for this is shown in listing 7.

For sliding dividing lines there is nothing to do, because the refinement stops at the boundary of a subdomain.

For the real refinement of the elements the logical array *refel*, where we inserted a *true* into the rows of the refinement elements in the column that corresponds to the refinement stage of the element, is rather impractical because we had to go through the columns and had to check for each element if the entry in the array *refel* is *true*. As it affects the performance we want to avoid this kind of if-construct and want to introduce an integer array where

```
    ! insert address of edge j into dlote
    dlote(addrold+j-1) = addr
    addr = addr+1

    ! for each dividing line
    do i = 1,2*n_inb
      ! check if both end points belong to same DL
      if (dlsect(new_1,i).and.dlsect(new_2,i)) then
        ! sect: number of subdomain the end points belong to
        sect = tnodst(new_1,3)

        ! initialize potsect
        potsect = .false.
        ! for each subdomain
        do k = 1,n_sect
          ! check for end points if they couple to sub-
          ! domain k and if this is true for both nodes
          ! we set potsect(k,1) = true
          if (tnodst(new_1,3+k) > 0) potsect(k,1) = .true.
          if (tnodst(new_2,3+k) > 0) potsect(k,2) = .true.
          potsect(k,1) = potsect(k,1).and.potsect(k,2)
        end do
        ! set potsect = false for own subdomain
        potsect(sect,1) = .false.

        ! for each subdomain
        do k = 1,n_sect
          ! if subdomain k is coupling subdomain
          if (potsect(k,1)) then
            ! insert twin node numbers into newpt and dlote
            newpt(k,1) = tnodst(new_1,3+k)
            newpt(k,2) = tnodst(new_2,3+k)
            cnt = cnt+1
            ! insert twin nodes into dlote
            dlote(addr+1) = tnodst(new_1,3+k)
            dlote(addr+2) = tnodst(new_2,3+k)
            addr = addr+3
            ! insert number of subdomain into indbd
            intbd(cnt) = k
          end if
        end do
        exit

      end if
    end do
```

Listing 7: Code for twin node search for dividing lines.

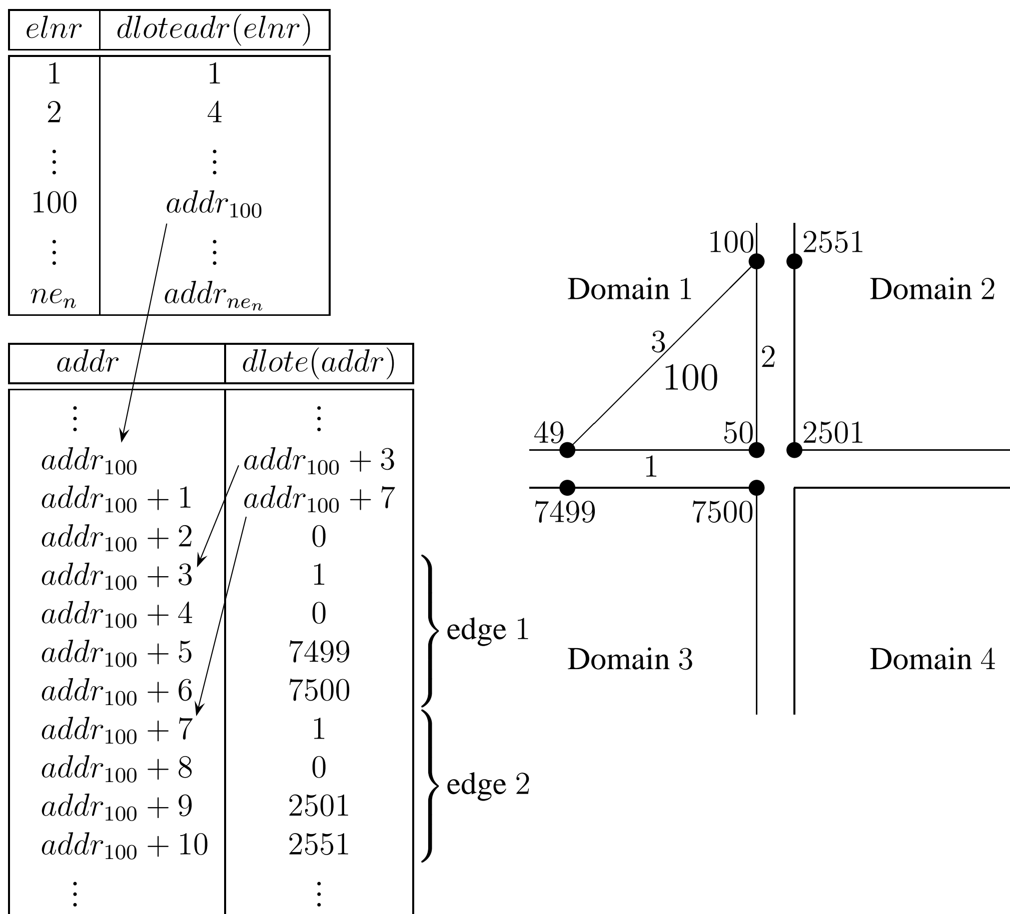


Figure 4.3.4: Illustration of the arrays *dlote* and *dloteadr*.

the numbers of the refinement elements are stored explicitly. We start in *refel* with the rightmost column and go through one column after the other until we completed column 1. The elements of refinement stage s are stored in column $s+1$ of *refel* (remember that the largest elements have refinement stage 0). For the elements that have a *true*-entry in the current column s (elements of refinement stage $s-1$!) we search for the neighbour elements of refinement stage s that must also be refined because of the refinement cascade. As we store the information for these elements in column $s-1$ we have to insert a *true* for these elements into *refel* in column $s-1$. This means that we do not add any elements in the current column s but only in the column $s-1$ that will be treated next, i.e. all elements of the current refinement stage are known when we go through this column. So we can transform the logical array *refel* to a one-dimensional **integer array *indrel*** with length ne_l after having determined the refinement elements of a refinement stage (see listing 5). In *indrel* we store the numbers of the refinement elements sorted by refinement stage, starting with the finest elements. The starting address of each refinement stage is stored in an

integer array *narpl* with length equal to $rs_{max}+1$ and width equal to 2, see figure 4.3.5, where rs_{max} is the highest refinement stage and it holds

$$rs_{max} = n_{cyc} - 1 \tag{4.3.8}$$

where n_{cyc} is the number of the current computation cycle. This is because the elements of the initial grid ($n_{cyc} = 1$) have refinement stage 0. So in $narpl(rs+1,1)$ is stored the starting address of the refinement elements of refinement stage rs in *indrel*. In $narpl(rs+1,2)$ we store the number of refinement elements of refinement stage rs. The refinement cascade on a single processor is illustrated in figure 3.4.2, the algorithm for this preparation step of the mesh refinement is shown in algorithm A.

element number	*refel*			
	stage 0	stage 1	stage 2	stage 3
1	false	true	false	false
2	true	false	false	false
3	false	false	false	true
4	false	false	false	false
⋮	⋮	⋮	⋮	⋮
ne_l-3	false	true	false	false
ne_l-2	false	false	false	true
ne_l-1	false	false	false	false
ne_l	false	false	true	false

narpl

ref. stage	addr of 1st el.	number of el.
0	85	35
1	42	43
2	17	25
3	1	16

i	$indrel(i)$	
1	3	stage 3
⋮	⋮	
16	ne_l-2	
17	7	stage 2
⋮	⋮	
41	ne_l	
42	1	stage 1
⋮	⋮	
84	ne_l-3	
85	2	stage 0
⋮	⋮	
119	471	

Figure 4.3.5: Illustration for the transformation of the array *refel* to the arrays *indrel* and *narpl*.

RECAPITULATION: By the means of the refinement cascade we ensure that the mesh does not become degenerated in the next cycle and that the storage scheme is kept. The scheme of information of the array *refel*, see table 4.3.1, is the basic key for the refinement cascade. We only have to care about neighbour elements of refinement elements that have a lower refinement stage so that the difference of the refinement stages does not become greater than one after the refinement and we have not more

than three nodes on an edge. After the refinement cascade we transform the logical array *refel* for the refinement elements into an integer array *indrel* that is suitable for the fast processing of the refinement of the elements. The refinement cascade continues on the other side of dividing lines as we have matching grids. In the array *dlote* we have to store the twin nodes on the other side of the dividing line for each dividing line edge of each dividing line element, and in 3-D also the number of dividing lines an edge couples to. For sliding dividing lines the refinement does not continue on the other side of the sliding dividing line so that the refinement cascade is limited to the subdomain where it has been started.

indrel: integer array for refinement element numbers
refel: logical array for refinement elements
refpt: integer array for refinement node numbers

A1 Determine refinement nodes because of the error (store in *refpt*).

A2 Determine refinement elements because of the error (store in *refel*).

A3 Determine refinement elements because of the refinement cascade (store in *refel*).

A4 Transform logical array *refel* to integer array *indrel*.

Algorithm A: Algorithm for the preparation step of the mesh refinement on a single processor.

After the execution of the refinement cascade we check if we will exceed the limit for the number of elements if we refine the mesh as desired. The user prescribes the maximum number of elements ne_{max} that determines the length of the arrays for the element information. If this number is exceeded we must stop the computation and the user must increase this value. So we must

$$\text{check if } \; ne + el_{new} \cdot \sum_{rs=0}^{rs_{max}} narpl(rs+1,2) > ne_{max}$$

and if yes, we print out an error message and stop the computation.

4.3.2 Refinement cascade on a distributed memory parallel computer

SUMMARY: On a distributed memory parallel computer it must be clear which processor is authorized to refine a refinement element. It is not of necessity that this is the processor the refinement of the element originates from. We put up two more rules that ensure the assignment of a refinement element to a single processor. So we determine the refinement processor for each refinement element that is in the overlap, collect all

these elements for each processor in special arrays and finally send them to the processors that must refine them. We have to avoid multiple sending of the same element numbers. The receiving processors must insert the new elements into their refinement element arrays.

Important notations:

indrel: integer array for refinement element numbers
lsent: logical array for elements that have already been sent to an overlap processor
narpl: integer array for information about storage of data in *indrel*
nenr: integer array for element information
nenrs: integer array for corresponding local number for a global element number
rcvl: integer array for updating element information in left overlap
rcvr: integer array for updating element information in right overlap
refel: logical array for refinement elements
sndlct: integer array, counter for *sndlto*
sndlto: integer array for elements to be sent to the left during ref. cascade
sndrct: integer array, counter for *sndlto*
sndrto: integer array for elements to be sent to the right during ref. cascade

Now we want to discuss the refinement cascade on a distributed memory parallel computer. Here we determine all refinement elements that must be refined because of the error exactly like on a single processor. As it must be clear by which processor an element is refined, we put up another rule:

Rule 3
The refinement of an element is always done by the processor that owns the refinement element.

But in order to know what to do with this rule, we need yet our fourth rule that we already mentioned in section 3.6:

Rule 4
An element is always owned by that processor its leftmost node is owned by.

Because of rule 3 we have to take care of the refinement elements in the left overlap. These elements are not refined by the processor that owns the refinement node (of course, there may be another refinement node of the same element on the own processor) and therefore the own processor of the refinement node must send this information to the own processor

of the refinement element. For the right overlap there is nothing to do for the time being, because the elements always belong to the processor where the leftmost node belongs to and therefore the own processor of all elements that have to be refined because of the error is the own processor of the refinement node at the same time.

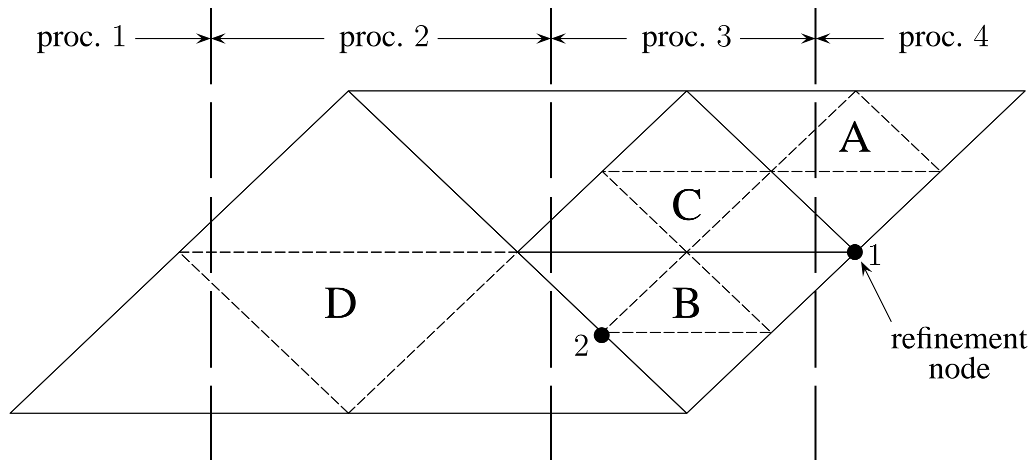


Figure 4.3.6: Illustration of the refinement cascade on $np = 4$ processors.

An exemplary situation is illustrated in figure 4.3.6 for four processors. Note that the same situation has been illustrated in figure 3.4.2 for a single processor. Let node 1 be a refinement node because of the error. The triangles A, B and C (solid lines) are refinement elements of refinement stage s. By the refinement of element B there would be created a fourth node (node 2) on the right edge of element D. As the refinement node 1 is on processor 4, but neither element A nor B nor C are elements owned by processor 4, this processor has to send the information about the necessary refinement of the three elements to processor 3 (for element A) and processor 2 (for elements B and C). On processor 2 we look for the neighbours of element B and see that element D is of refinement stage $s - 1$ and therefore has to be refined. But element D is on processor 1, so that processor 2 has to send the information about the refinement of element D to processor 1. There the element must be marked as a refinement element, so we must enter a *true* into the logical array *refel* of processor 1, in the row that corresponds to the local element number in column $s - 1$ to be specific.

But how is this complex algorithm implemented? We have to allocate five additional arrays with the names, lengths and types shown in table 4.3.1. The efficiency of the mesh refinement goes down drastically, if all the element numbers are sent individually. Therefore we compile all the element numbers to be sent to the left and right in special arrays.

These **arrays** are ***sndlto*** for the element numbers to be sent to the left where ***sndlct*** is the corresponding counter array for the overlap processors that counts the number of elements in the columns of *sndlto*. For the element numbers to be sent to the right we have the **arrays *sndrto*** and ***sndrct***. In the one-dimensional logical **array *lsent*** we store a *true* for all element numbers that we already sent to the left or right in order to avoid multiple sending of the same element number.

name	dimension of array	type
sndlto	$ne_{ll,max} \times np_{sl}$	$integer$
sndrto	$ne_{lr,max} \times np_{sr}$	$integer$
sndlct	$np_{sl} \times 2$	$integer$
sndrct	np_{sr}	$integer$
lsent	$ne_{ll,ip} + ne_{lr,ip}$	$logical$

Table 4.3.1: Length and type of arrays for refinement cascade.

In table 4.3.1 $ne_{ll,ip}$ and $ne_{lr,ip}$ are the numbers of elements in the left and right overlap of processor ip, $ne_{ll,max}$ and $ne_{lr,max}$ are the maximum numbers of elements in the left and right overlap (maximum over all processors):

$$\begin{aligned} ne_{ll,max} &= \max_{ip=1,\dots,np} ne_{ll,ip} \\ ne_{lr,max} &= \max_{ip=1,\dots,np} ne_{lr,ip}. \end{aligned} \tag{4.3.9}$$

np_{sl} and np_{sr} are the numbers of overlap processors on the left and right side to which we may have to send information.

When we determine the refinement elements that have to be refined because of the error we have to take a look at those elements that are in the left overlap. As we explained above these elements can only be refined by their own processor. So for each processor in the left overlap we have one column in the array *sndlto*, from column 1 for the leftmost overlap processor to column np_{sl} for the direct left neighbour processor, see table 4.3.3. For each element we know which processor ip_{own} is its own processor, this information is stored in the element information **array *nenr* of integer type**, where we store in the first column the global element number, in the second column the number of the subdomain the element belongs to, from the third column we know if an element is a dividing line or sliding dividing line element, and in the fourth column we store the owning processor of the elements. The contents of array *nenr* is illustrated in table 4.3.2.

We determine the relative position ip_{rel} of processor ip_{own} to processor ip by

$$ip_{rel} = ip_{own} - ip. \tag{4.3.10}$$

local elem.	$nenr(elem, k)$ $k = 1$	$k = 2$	$k = 3$	$k = 4$
1 ⋮ ne	global element number	sub-domain number	number of coupling subdomains	number of owning processor

Table 4.3.2: Illustration of the contents of array *nenr*.

Note that ip_{rel} is negative for left neighbour processors. As the difference between the processor number ip and the processor number of the leftmost neighbour processor is np_{sl} the corresponding column ip_{col} in *sndlto* can be computed by

$$ip_{col} = ip_{rel} + 1 + np_{sl}. \tag{4.3.11}$$

In column ip_{col} in *sndlto* the element number has to be inserted. With our construction we always have the data of the leftmost processor to send data to in the first column. This is only to save memory because otherwise we would always have the data that has to be sent to processor 1 in column 1 of *sndlto* although only few processors really need to send data to this processor so that on most of the processors there would be no entries and therefore we would waste memory.

As we will see later, we must distinguish between the refinement elements in the left overlap that have to be refined because of the error and those of which the refinement results from the refinement cascade. This is the reason why we have two columns in the counter array *sndlct*. In the first column of *sndlct* we only count the elements that have to be refined because of the error, in the second column we count all refinement elements including the refinement elements because of the refinement cascade and because of the error.

Before the refinement cascade is started for the first time, we insert the refinement elements because of the error into *sndlto*, so if a refinement element is in the left overlap we increase the overlap element counter *sndlct* for this column by one:

$$sndlct(ip_{col}, 1) = sndlct(ip_{col}, 1) + 1. \tag{4.3.12}$$

This counts the position in *sndlto* at the same time, so we store the global element number $nr_{el,g}$ that we get from the first column of the element information array *nenr* in *sndlto*:

$$sndlto(sndlct(ip_{col}, 1), ip_{col}) = nr_{el,g}. \tag{4.3.13}$$

Furthermore, we enter a ***true*** into the logical array *lsent*. This array only comprises the elements of the left and right overlap, so the ***true*** for the local element $nr_{el,l}$ is entered at position $nr_{el,l} - ne_l$:

$$lsent(nr_{el,l} - ne_l) = true. \tag{4.3.14}$$

The array *lsent* is to guarantee that the same element information is not sent more than once to the concerned processor and is initialized with *false*. For example, if an element in the left overlap must be refined because of the error, its number is sent to the left. At the same time, the processor receives some refinement element numbers from the left and starts the local refinement cascade again. By this cascade the element must be refined again, this time because of the refinement cascade. So we would send its number to the left again. Therefore we enter a *true* into *lsent* for the elements we insert into *sndlto* and *sndrto*. Before we insert an element number into these arrays when executing the local refinement cascade we check if its entry in *lsent* is already set to *true*. The code for the inserting of the element numbers into *sndlto* of those elements that have to be refined because of the error is shown in listing 8.

a)

i	$sndlto(i,j)$ $j=1$	$j=2$	$j=3$
1	103	111	113
2	104	112	115
3	0	114	116
4	0	0	118
5	0	0	0
⋮	⋮	⋮	⋮
$ne_{ll,max}$	0	0	0
send to proc.	$ip-3$	$ip-2$	$ip-1$

j	$sndlct(j,k)$ $k=1$	$k=2$
1	0	2
2	2	3
3	4	4

b)

i	$sndrto(i,j)$ $j=1$	$j=2$	$j=3$
1	207	208	214
2	210	211	222
3	212	216	0
4	213	0	0
5	0	0	0
⋮	⋮	⋮	⋮
$ne_{lr,max}$	0	0	0
send to proc.	$ip+1$	$ip+2$	$ip+3$

j	$sndrct$ (j)
1	4
2	3
3	2

Table 4.3.3: Illustration of the arrays a) *sndlto* and *sndlct*, b) *sndrto* and *sndrct* for processor ip, i is number of entry, j identifies number of processor to send data to.

When we search for the neighbour elements that are of lower refinement stage, i.e. for the elements that have to be refined because of the refinement cascade, and find one that is an overlap element in the left overlap, we also have to enter the global element number into *sndlto*, i.e. the array with the refinement element numbers in the left overlap. As the second column of *sndlct* counts all refinement elements in the left overlap, we set the

```
! compute auxiliary value for ip_col
ip_hlp = myproc-1-np_sl

! for each element in the left overlap
do l = ne_l+1,ne_l+ne_ll
  ! check if element is ref. element
  if (refel(l,refst(l))) then
    ! compute relative proc. number to send data to
    ip_col = nenr(l,4)-ip_hlp
    ! increase counter and insert global elem. nr. into sndlto
    sndlct(ip_col,1) = sndlct(ip_col,1)+1
    sndlto(sndlct(ip_col,1),ip_col) = nenr(l,1)
    ! insert true into sent for current element
    lsent(l-ne_l) = .true.
  end if
end do
```

Listing 8: Code for inserting data into arrays *sndlto*, *sndlct* and *lsent*.

second column equal to the first one before we start the refinement cascade:

$$sndlct(ip_{col}, 2) = sndlct(ip_{col}, 1) \quad \text{for} \quad ip_{col} = 1, \ldots, np_{sl}, \tag{4.3.15}$$

and if we find a new refinement element during the execution of the refinement cascade that has to be sent to processor ip_{own}, we compute the corresponding column in *sndlto* by (4.3.10) and (4.3.11) and increase the counter in the second column of *sndlct* by one:

$$sndlct(ip_{col}, 2) = sndlct(ip_{col}, 2) + 1. \tag{4.3.16}$$

At the end of the local refinement cascade the counters in the second column of *sndlct* comprise all refinement elements that have to be sent to the corresponding processors—refinement elements because of the error and because of the cascade—whereas in the first column only the numbers of refinement elements because of the error are stored. In *sndlto* we do not make any difference between the stored elements, we only know by the counter *sndlct* which is the first refinement element because of the refinement cascade. The counters for the refinement elements because of the error in the first column of *sndlct* are only greater than zero for the first step of the refinement cascade, i.e. during the first determination of the elements to be refined because of the cascade condition, as we store all refinement elements because of the error before we start the cascade. In later steps we only have refinement elements because of the refinement cascade. We will see at the end of this subsection why we have to distinguish between the origin of the refinement elements.

If we search for the cascade find an element of lower refinement stage that is in the right overlap, we have to enter the element number into *sndrto*. The arrays *sndrto* and *sndrct*

have the same meaning as *sndlto* and *sndlct*, but there the refinement elements of the right overlap are stored. Note that there are not any refinement elements in the right overlap because of the error, but only because of the refinement cascade! This is because an element can be a refinement element because of the error only if one of its nodes is a refinement node. But as an element is owned by the processor the leftmost node belongs to (rule 4), the element would be owned by the refinement processor itself if one of its nodes is owned by this processor, but it can never be in the right overlap. This explains why the array *sndrct* needs no second column as *sndlct* (see table 4.3.1).

Now for the first time of the refinement process we need communication between the processors. The arrays *sndlto* and *sndrto* must be exchanged. If there are not any refinement elements in the overlap of any processor, here nothing has to be done and the preparing step of the mesh refinement is done. Otherwise, we first send the refinement element array *sndrto* to the right. The receiving processor has no information about the length of the message it will receive. So we split up the communication as described in subsection 4.1.4 and first send the number of elements stored in *sndrto* to the right in $np-1$ cycles.

Then it is known on each processor how many element numbers it will receive and so the real element data can be sent. As the exchange of data can only be made by global element numbers, the receiving processors first must determine the corresponding local element numbers. This is done by a binary search in the **integer array *nenrs***—*nenrs* is the inverted first column of the element information array *nenr*, i.e. we store the corresponding local element number for a global one in *nenrs*—that is sorted by ascending global element number. For each element we enter a *true* into the logical array *refel*, i.e. the array for the refinement elements, in the column of its refinement stage. After the execution of each local refinement cascade we transform the *refel* array to the integer arrays *indrel* and *narpl* and afterwards initialize *refel* with *false* so that there are only *true* entries for the element numbers the processor just received.

After all processors have sent, received and processed the data to the right, we continue with the element information that has to be sent to the left. Here we proceed in the same way, i.e. we first send the message lengths stored in *sndlct* to the left in $np-1$ cycles and afterwards send the real element data to the concerned processors, in this case we send the array *sndlto* with the global element numbers of the refinement elements in the left overlap. The receiving processors insert the data into their *refel* array in the correct column. For the entering of the elements into *refel* the reason for the refinement of an element is of no importance so we make no difference between refinement elements because of the error and refinement elements because of the refinement cascade.

A processor may receive the numbers of refinement elements from an overlap processor that it already has to refine himself by its own information and therefore already has inserted

indrel: integer array for refinement element numbers
refel: logical array for refinement elements
refpt: integer array for refinement node numbers

B1 Determine refinement nodes because of the error.

B2 Determine refinement elements because of the error.

B3 Determine refinement elements because of the refinement cascade.

B4 Transform logical array *refel* to integer array *indrel*.

B5 Check if there are any refinement elements in the left or right overlap of any processor. If not, exit.

B6 Re-initialize *refel* by *false*.

B7 Pass message lengths for refinement elements in the right overlap to the own processors of these elements.

B8 Pass global refinement element numbers in the right overlap to the own processor of these elements.

B9 Receiving processors:

- **a)** Transform received global element numbers to local element numbers.
- **b)** Enter *true* into *refel* for each received refinement element.

B10 Pass message lengths for refinement elements in the left overlap to the own processors of these elements.

B11 Pass global refinement element numbers in the left overlap to the own processors of these elements.

B12 Receiving processors:

- **a)** Transform received global element numbers to local element numbers.
- **b)** Enter *true* into *refel* for each received refinement element.

B13 Determine new refinement elements because of the refinement cascade.

B14 Insert <u>new</u> refinement elements into array *indrel*.

B15 Continue with step B5 (local refinement cascade may lead to new refinement elements in the overlap).

Algorithm B: Algorithm for the preparation step of the mesh refinement on a distributed memory parallel computer.

the element numbers into its *indrel* array, i.e. the integer array for the refinement element numbers. We must avoid that the same element number is inserted twice into *indrel*, and we want to have all refinement elements of a stage stored directly one after the other. So we make a local copy of the *indrel* array which is denoted by ***indcp*** and update *indrel* during the refinement cascade. The local refinement cascade consists of several steps, and in each step we check the neighbour elements of the refinement elements of the refinement stage that corresponds to the current step number. At the beginning of each step of the refinement cascade we get the element numbers of those elements from the local copy *indcp* that must already be refined and of which the refinement stage corresponds to the current step number of the refinement cascade. These element numbers are copied back into the *indrel* array. Furthermore, we enter a *false* into *refel*, i.e. the logical array for the refinement elements, for these elements (for *refel* and *narpl* see figure 4.3.5). In step s of the refinement cascade we set:

$$refel\,(indcp(i), s) = false \quad \text{for} \quad i = el_{first}, \ldots, el_{last} \tag{4.3.17}$$

$$\begin{aligned} \text{with} \quad el_{first} &= narpl(s+1,1) \quad \text{and} \\ el_{last} &= narpl(s+1,1) + narpl(s+1,2) - 1. \end{aligned}$$

Afterwards only those received elements still have the value *true* in *refel* of which we did not yet enter the element number into *indrel*, so we append their numbers to the *indrel* array after we checked if they evoke the refinement of another element. The *narpl* array is adapted accordingly. Then we continue with the refinement elements of the next refinement stage.

During the execution of the refinement cascade we may again come across refinement elements that belong to the overlap. So this procedure is repeated until there are not any new refinement elements in the overlap any more. The counters in the first column of *sndlct* are only greater than zero for the first step of the refinement cascade as we store all refinement elements because of the error before we start the cascade. In later steps we only have refinement elements because of the refinement cascade.

After the last step of the refinement cascade we have stored the numbers of all elements that have to be refined in the array *indrel*. The refinement can start now and during the refinement process no new refinement elements are added. The whole preparation step of the mesh refinement algorithm is shown in algorithm B.

RECAPITULATION: By the means of rules 3 and 4 we get a clear assignment of the elements to the processors and we define the refinement processor for the elements. In one step of the refinement cascade the refinement element numbers of the overlap are compiled in the arrays *sndlto* and *sndrto* for the left and right overlap and afterwards the columns of these arrays are sent to the corresponding processors that insert the element information in the array *refel*. Then the own refinement element numbers are

deleted from this array and the current column of *refel* is transformed into an integer array that is appended to the array *indrel*. Afterwards we continue with the elements of the next refinement stage.

After the execution of the refinement cascade we check if we will exceed the limit for the number of elements if we refine the mesh as desired. The maximum number of elements ne_{max} that is precsribed by the user is a local length on a distributed memory parallel computer, so we may store ne_{max} elements per processor. If this number is exceeded we must stop the computation and the user must increase this value. So we must

$$\text{check if } \; ne_n + el_{new} \cdot \sum_{rs=0}^{rs_{max}} narpl(rs+1,2) > ne_{max}$$

on each processor and if this inequality is fulfilled on one or more processors, we print out an error message and stop the computation.

In the refinement process itself we start with the elements of refinement stage 0 and continue up to refinement stage rs_{max}. If we have completed the refinement of one stage we have to update the information on the other processors where the refinement cascade has caused the refinement of elements of the actual stage. This is because these evoking elements that will be refined in the next step have the old information about their neighbours (that have been refined in the meantime). So the information which elements have been created from the old ones must be exchanged. This is necessary because from the future refinement of the elements new nodes will be created that also have to be stored in the neighbour elements.

Therefore we have to store the refinement elements in the overlap that result from the refinement cascade in two **integer arrays *rcvl*** and ***rcvr*** that both have the length ne_l (as we do not yet knowhow many local elements will be concerned) and the width 3. All elements that have to be refined because of a refinement cascade on a left neighbour processor are inserted in *rcvl*. In the first column we enter the local element number—which must be transformed into the global element number for the data exchange—, in the second column the starting address of the new element numbers—this is necessary to get a relation between the refined old element and the newly-created elements—and in the third column the own processor of the evoking element—so that it is known where the information has to be sent to later. The same holds for *rcvr* for the right neighbour processors. As we only want to store the refinement elements that have to be refined because of the refinement cascade in *rcvr*, we have to distinguish between the elements that have to be refined because of the error and those that have to be refined because of the refinement cascade when we send *sndlto* to the left side as described on page 81. Note that in the right overlap we have refinement only because of the error.

If we have coupled domains with dividing lines all that we just described is the same. In the (locally executed) refinement cascade we also have to look for the neighbour elements on the other side of the dividing lines as explained on pages 66ff. On a distributed memory parallel computer the length of the *dloteadr* array is equal to ne_n, that of the *dlote* array is equal to l_{dlote} where l_{dlote} is computed like for a single processor, see (4.3.1) on page 70.

4.4 Mesh refinement on a single processor

We first consider the algorithm of the mesh refinement for a single processor in 2-D and explain the differences in 3-D and for dividing lines and sliding dividing lines afterwards. Nevertheless, it is always the same program, i.e. we have integrated all versions into one program and control the program flow by global input parameters.

4.4.1 Mesh refinement in 2-D

Important notations:

bnod: integer array for external boundary nodes
indrel: integer array for refinement element numbers
infpol: double precision array for evaluated influence polynomials
ipadd: double precision array for starting addresses in *infpol*
lnpl: logical array for edge identification
narpl: integer array for information about storage of data in *indrel*
nb: integer array for neighbour element numbers
nbadd: integer array for starting addresses in *nb*
nek: integer array for node numbers of the elements
nekinv: integer array for elements a node belongs to
nenr: integer array for element information
nnr: integer array for node information
nnrs: integer array for corresponding local number for a global node number
rtl: integer array for refinement edge information
u: double precision array for function values

SUMMARY: We describe one refinement step, i.e. the refinement of the elements of the lowest refinement stage, and mention the differences to the following steps (refinement of higher refinement stages) at the end of this subsection. As the nodes that are generated in this refinement step do not only have to be inserted into the refinement elements themselves but also into their neighbour elements, we have to determine these neighbour elements first. We obtain the new nodes by halving the edges of the refinement elements. The refinement of an element means bisection of its edges. So we insert all necessary information about the refinement edges into an array and afterwards generate

the new nodes which means node number, coordinates, consistency order and function value. We insert the new nodes in the elements they belong to and check if they are external boundary nodes. Then we separate the old elements into four new elements.

The mesh refinement begins with the refinement of the largest elements, and we continue with the next refinement stages up to stage rs_{max}. As the largest elements have refinement stage 0 we refine the elements of refinement stage $rs - 1$ in step rs creating elements of stage rs. At the beginning of each refinement step we have to create the inverted *nek* array *nekinv*, where we store the numbers of the elements a node belongs to, as described on page 24 because we changed the array *nek* in the step before. The *nekinv* array is needed for the search for the neighbour elements of the refinement elements of the current refinement stage.

This neighbour search is done next. Here we only search for elements of the same refinement stage because we also must enter the numbers of the new nodes into the array *nek* for the neighbour elements. We go through the array *indrel* with the refinement element numbers and search for each edge of the stored elements for elements that contain both end points. These elements are stored in an **integer array *nb*** (the refinement element itself is excluded). This two-dimensional array is allocated in every refinement step rs and has the length ne_{re} which is the number of refinement elements in the current step:

$$ne_{re} = narpl(rs + 1, 2). \tag{4.4.1}$$

The width is equal to n_{edge} because n_{edge} is the number of edges of an element and an element may have at most one neighbour element per edge in 2-D. Therefore *nb* has three columns for the neighbours of the three edges. As the neighbour element information for each edge is written one after the other into the same row, we need an additional information array for the starting address of each edge. This is the **integer array *nbadd*** and has the same length as *nb* and width equal to $n_{edge} + 1$ which we initialize by 0. These two arrays are illustrated in figure 4.4.1 for the arbitrarily chosen element 36. The first edge always starts in entry 1, of course, the neighbour element is element 35. As the second edge is an external boundary edge, there is not any neighbour element for this edge, and the neighbour element for edge 3 with the number 37 is stored in entry 2 of *nb*. In the last column of *nbadd* we store the end address of the neighbour elements of the last edge, increased by one, so that the numbers of the neighbour elements of element el at edge k are stored in the entries $nbadd(el, k)$ to $nbadd(el, k + 1) - 1$ for each edge.

In listing 9 you can see the code for the neighbour search of the elements. In contrast to the refinement cascade we now search for elements of the same refinement stage, so the examined edge must not have a mid-point yet. Then we go through the row in the *nekinv* array where we stored the element numbers the first end point of the edge belongs to. For each element we look in the row of the *nekinv* array where the element numbers of the second

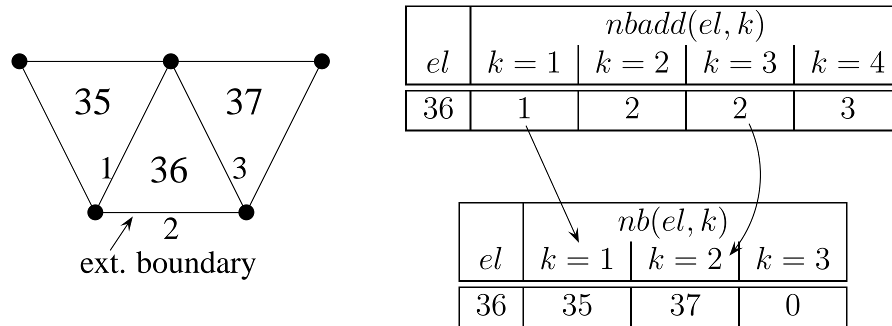


Figure 4.4.1: Illustration of the arrays *nb* and *nbadd* for element 36 (2-D).

end point are stored for the same element number. The elements that occur in both rows are stored in *nb* (again the examined element itself is excluded) and the counter *nbadd* is increased by the number of found elements. In 2-D we always find at most one neighbour element.

An example is illustrated in figure 4.4.2. We search for the neighbour element of element 95 for the edge with the end points 52 and 44 by associative comparison in a double loop. The performance could be improved by sorting the element numbers in ascending order and then execute a binary search. When we go through the corresponding rows in the *nekinv* array we find out that elements 60 and 95 occur in both rows 44 and 52. So element 60 is the desired neighbour element.

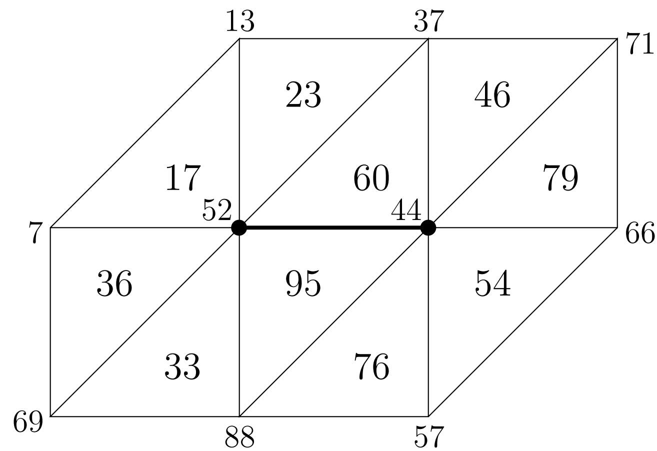
The refinement of an element is reduced to halving its edges followed by grouping the old and new nodes to get the new elements. So the main task is the refinement—which means bisection—of the edges of the refinement elements. Therefore we have to transform the array *indrel* with the numbers of the refinement elements into an array where we store all necessary edge information. This comprises the number of the new node, the numbers of the end points of the refinement edge, the refinement element number, the position where the new node number has to be entered in the corresponding row of the *nek* array (see figure 4.4.3), i.e. the array for the node numbers in an element, the number of neighbour elements and the neighbour element numbers. As we already mentioned, an element may have at most one neighbour element at an edge in 2-D, so we have to store seven integer numbers per refinement edge. The one-dimensional **integer array** to store the data is denoted by ***rtl*** and has got the length $7 \cdot ne \cdot n_{edge}$, see table 4.4.1. We enter the edge information described in this table into the array *rtl* only if two conditions are fulfilled: firstly, the edge must not have a mid-point yet because otherwise, the edge has not to be halved, and secondly, the edge does not yet occur in the array.

Of course, new nodes are only generated on edges that are not already halved. So we insert

```
do i = 1,narpl(rs+1,2)     ! initialize address array nbadr
  nbadr(i,1) = 1
end do

fst = narpl(rs+1,1)        ! compute first/last ref. elem.
lst = fst+narpl(rs+1,2)-1  ! of current ref. stage

do j = 1,n_edge            ! for each edge
  coli = hlpnek(j,1)       ! set column nrs. for current edge
  colj = hlpnek(j,2)
  do i = fst,lst           ! for each ref. element
    curr = i-fst+1         ! curr: nr. of current elem.
    cnt  = nbadr(curr,j)   ! set current counter cnt
    elnr = indrel(i)       ! elnr: current ref. elem.

    ! check if mid-point is nonexistent
    if (nek(elnr,j+elpt) == 0) then
      nod_1 = nek(elnr,coli) ! set end points of current edge
      nod_2 = nek(elnr,colj)
      cntold = cnt           ! store cnt

      ! for each element nod_1 belongs to
      do k = 1,inne(nod_1)
        ! for each element nod_2 belongs to
        do l = 1,inne(nod_2)
          ! check if element numbers are equal
          if ((nekinv(nod_1,k) == nekinv(nod_2,l)).and.
 &            (nekinv(nod_2,k) /= elnr)) then
            ! insert element into nb and increase counter
            nb(curr,cnt) = nekinv(nod_1,k)
            cnt = cnt+1
          end if
        end do
      end do

      ! compute new nnb_max
      nnb_max = max0(nnb_max,cnt-cntold)
    end if

    ! store new counter cnt in nbadr
    nbadr(curr,j+1) = cnt
  end do
end do
```

Listing 9: Code for arrays *nb* and *nbadd*.

the end points of a refinement edge into a logical array and for each investigated edge we look in this array if the combination of end points has already been inserted. Below we



$node$	$nekinv(node, k)$					
	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$
44	46	54	60	76	79	95
52	17	23	33	36	60	95

Figure 4.4.2: Illustration of the search for neighbour elements of the same refinement stage.

describe in detail how this is done. If yes, the edge is skipped. For the other edges we store in *rtl* the information that we specified above and that is shown in table 4.4.1 where r_{cnt} is the starting address of the information of edge j in refinement element el. By pos we denote the position of the node in the refinement element, see figure 4.4.3. In 2-D each element may have at most one neighbour element at an edge. When we give the numbers to the new nodes later, we go through the *rtl* array sequentially and fill in the first entry of each refinement edge.

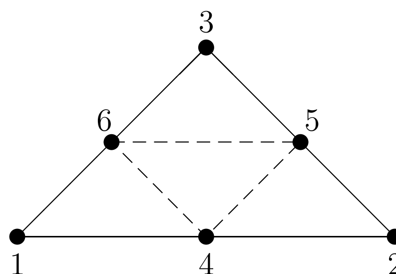


Figure 4.4.3: Local numbering of the nodes in an element (2-D).

As the same edge may be halved by two different refinement elements, i.e. by its two neighbour elements in 2-D or more elements in 3-D, we must assure that the mid-point is

generated only once. Therefore we have a **logical array *lnpl*** (logical neighbour point list) with length equal to n and width equal to non_{max} where non_{max} is the maximum number of neighbour nodes of a node (that we get from the nearest neighbour ring). This array is initialized with *false*. We have one row for each node and the kth column is for the kth neighbour node of the nodes. The next neighbour ring is stored in *fstring*, see on page 24. So if nod_1 and nod_2 are the end points of the edge, we first determine which neighbour nod_2 is of nod_1 and which neighbour nod_1 is of nod_2 in the next neighbour ring. These numbers are denoted by nb_1 and nb_2. For example, in figure 4.4.4 node B is the fifth neighbour of node A and node A is the second neighbour of node B, so it holds $nb_1 = 5$ and $nb_2 = 2$.

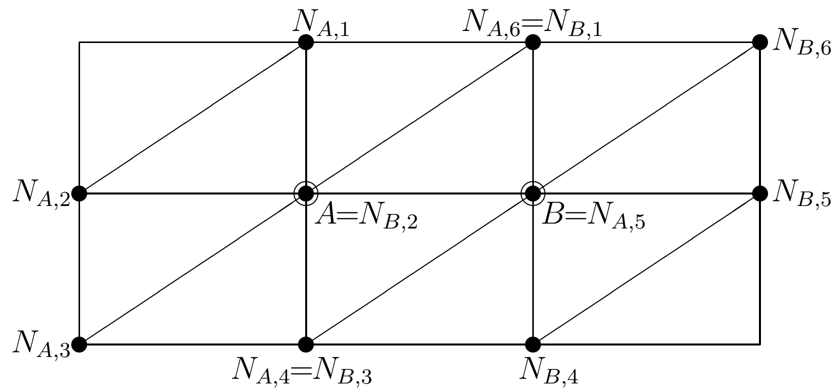


Figure 4.4.4: Illustration of the neighbourhood of two nodes.

When the numbers nb_1 and nb_2 are known we can insert an edge into *lnpl*, i.e. we change the nb_1^{th} entry in row nod_1 and the nb_2^{th} entry in row nod_2 to *true*:

$$\begin{aligned} lnpl(nod_1, nb_1) &= true \\ lnpl(nod_2, nb_2) &= true. \end{aligned} \tag{4.4.2}$$

But before we insert this information into *lnpl*—and afterwards the refinement edge information into *rtl*—we must be sure that the edge did not occur yet to avoid multiple numbering of the same new node, so we

$$\text{check if } \; lnpl(nod_1, nb_1) = true$$

and if yes, this means that there already occurred a refinement element that also contains this edge and therefore the current edge is skipped. Otherwise, we insert the edge as seen in (4.4.2) and store the information of table 4.4.1 in the array for the refinement edges, i.e. the *rtl* array. As we store the same information twice in *lnpl* we could save storage if we only stored the information in the row of the node with the smaller local number. But this

i	$rtl(i)$	description
⋮	⋮	⋮
{ preceding edge }		
$r_{cnt}+1$		left free
$r_{cnt}+2$	nod_1	local number of end point 1 of edge j
$r_{cnt}+3$	nod_2	local number of end point 2 of edge j
$r_{cnt}+4$	el	local refinement element number
$r_{cnt}+5$	pos	local node number in element (see figure 4.4.3)
$r_{cnt}+6$	1	number of neighbour elements (= 1 in 2-D)
$r_{cnt}+7$	el_{nb}	local number of the neighbour element
{ following edge }		
⋮	⋮	⋮

Table 4.4.1: Illustration of the contents of array *rtl* in 2-D (edge information).

meant an additional comparison of the node numbers of the end points of an edge and thus additional time consumption. As our primary objective is performance on a distributed memory parallel computer we give away storage for that. After the edge information is stored, we increase r_{cnt} by 7.

RECAPITULATION: We have stored in the array *rtl* all necessary information we need for the current refinement step of the mesh refinement process. An edge is identified by its end points as the edges have not got any numbers and therefore cannot be stored easily in an array. We use a two-dimensional logical array where we set for one end point all entries to *true* that correspond to that neighbour nodes that form a refinement edge together with the first end point. The numbering of the edges would be another measure to improve the performance of the mesh refinement algorithm again at the price of additional storage.

SUMMARY: In *rtl*, the array where we store the refinement edge information, we have stored all element numbers where we have to insert the node numbers of the newly-created nodes. By this information we are able to generate the new nodes which means issuing of the new local and global node numbers, computing of the new coordinates, the consistency order and the function values and checking of the boundary node property. Moreover, we are able to insert the node numbers into the refinement elements and their neighbour elements.

After we inserted the data for all edges of all refinement elements into the *rtl* array we have

to check if we would exceed the maximum number of nodes by generating the new nodes of the current refinement step. The user prescribes the maximum number of nodes and elements, n_{max} and ne_{max}, respectively, that determine the length of all arrays for the node and element data. If these numbers are exceeded during the mesh refinement process we must stop the computation and the user has to increase these values. So we

$$\text{check if } \; n + \frac{r_{cnt}}{7} > n_{max}$$

and if yes, we print out an error message and stop the computation (note that we already checked if we exceed the limit for the elements after the execution of the refinement cascade). Otherwise, we enter the new local node numbers of the generated nodes into the first entry of the data for each edge that has been left free before in array *rtl*, see table 4.4.1. For each refinement edge in *rtl* a new mid-point is generated. The last old node has got the number n, and we continue counting the $r_{cnt}/7$ new nodes by the next natural numbers, so that the new nodes get the local numbers

$$\begin{array}{rl} \text{from} & n_{first} = n + 1 \\ \text{to} & n_{last} \;\, = n + \frac{r_{cnt}}{7}. \end{array}$$

So we have for the new nodes:

$$rtl\,(\,7\cdot(i-1)+1\,) = n+i \quad \text{for} \quad i = 1, \ldots, \tfrac{r_{cnt}}{7}. \tag{4.4.3}$$

For the node information we introduce an array just like we did for the elements (see on page 78, array *nenr*). In this **integer array *nnr*** we store the global node number in the first column and the number of the subdomain a node belongs to in the second column. In the third one we store the number of subdomains the node couples to, and in the fourth column we store the number of the own processor of the nodes. The contents of array *nnr* is illustrated in table 4.4.2.

local	$nnr(node,k)$			
node	$k=1$	$k=2$	$k=3$	$k=4$
1 ⋮ n	global node number	sub- domain number	number of coupling subdomains	number of owning processor

Table 4.4.2: Illustration of the contents of array *nnr*.

Then we enter the corresponding global node numbers into the node information array *nnr*.

For a single processor we have for the refinement nodes

$$\left.\begin{array}{rcl} nnr(n+i,1) & = & n+i \\ nnr(n+i,2) & = & 1 \\ nnr(n+i,3) & = & 0 \\ nnr(n+i,4) & = & 1 \end{array}\right\} \quad \text{for} \quad i = 1,\ldots,\tfrac{r_{cnt}}{7}. \tag{4.4.4}$$

For a single processor the global node number is equal to $n+i$ for the ith node. For a domain without dividing lines or sliding dividing lines we have to enter 1 in the second and 0 in the third column for each new node. The owning processor is 1 for each node for a single processor because there is only processor 1. Then we invert the first column of *nnr* by the procedure `HEAPSORTIX` (see subsection 4.1.3), so that we have for each global node number the corresponding local node number. This inverted **(integer) array** is called ***nnrs***.

We set

$$r_{cnt} \Leftarrow \frac{r_{cnt}}{7} \tag{4.4.5}$$

so that in the following $\boldsymbol{r_{cnt}}$ **is the number of refinement edges** which is equal to the number of new nodes, and then the new nodes get assigned their coordinates

$$\left.\begin{array}{l} x_{n+i} = \frac{x_{nod_{1,i}} + x_{nod_{2,i}}}{2} \\ y_{n+i} = \frac{y_{nod_{1,i}} + y_{nod_{2,i}}}{2} \end{array}\right\} \quad \text{for} \quad i = 1,\ldots,r_{cnt} \tag{4.4.6}$$

and their consistency order which is the lesser of the two orders of the end points of the edge:

$$q_{n+i} = \min\left(q_{nod_{1,i}}, q_{nod_{2,i}}\right) \quad \text{for} \quad i = 1,\ldots,r_{cnt}. \tag{4.4.7}$$

The computation of the new function values is more difficult. Here we have to evaluate the influence polynomials of the two end points of the edge nod_1 and nod_2 for the coordinates of the new mid-point. We have two possibilities to evaluate the influence polynomials. We can get the coefficients of the influence polynomials in the same way we got the difference formulas (because these are the partial derivatives of the influence polynomials) and evaluate them for the mid-point of the edge, i.e. we recompute the influence polynomials. The second way is to evaluate the influence polynomials as early as we generate the difference formulas and store the evaluated influence polynomials in order to use them for the interpolation (these are the coefficients of an interpolation formula for the central node of the difference star). The first method saves memory, the second one computation time. We decided on the second method although we do not know which edges have to be refined when we generate the difference formulas. So we have to evaluate the influence polynomials (3.3.3) for the mid-point of each edge of which a node is the end point. The evaluated

influence polynomials are stored in a one-dimensional **double precision array** denoted by ***infpol***. In 2-D, the length l_{infpol} of this array is computed by the formula

$$l_{infpol} = 6 \cdot \sum_{i=1}^{n} non_i \tag{4.4.8}$$

where n is the number of nodes before the mesh refinement process and non_i is the number of neighbour nodes of node i. The interpolation of the solution is always done with order $q = 2$ so we have 6 influence polynomials for an interpolation polynomial in 2-D. The starting addresses of the evaluated influence polynomials of a node in *infpol* are stored in a one-dimensional **integer array *ipadd*** with length equal to n. There $ipadd(i)$ points to the first influence polynomial of the first neighbour node of node i in *infpol*. After the 6 influence polynomials of the first neighbour node follow the influence polynomials of the second node and so on, see figure 4.4.5.

If we choose the consistency orders $q = 4$ or $q = 6$ for the solution of the system of PDEs, but for the interpolation of the function values for the new nodes we use order 2, the solution gets “dirty” by the mesh refinement. But this only holds for the initial solution of the following Newton iteration of the next computation cycle. The solution again is computed with consistency order $q = 4$ or $q = 6$, respectively, and the error from the mesh refinement will vanish, i.e. the solution is “cleaned” by the Newton iteration.

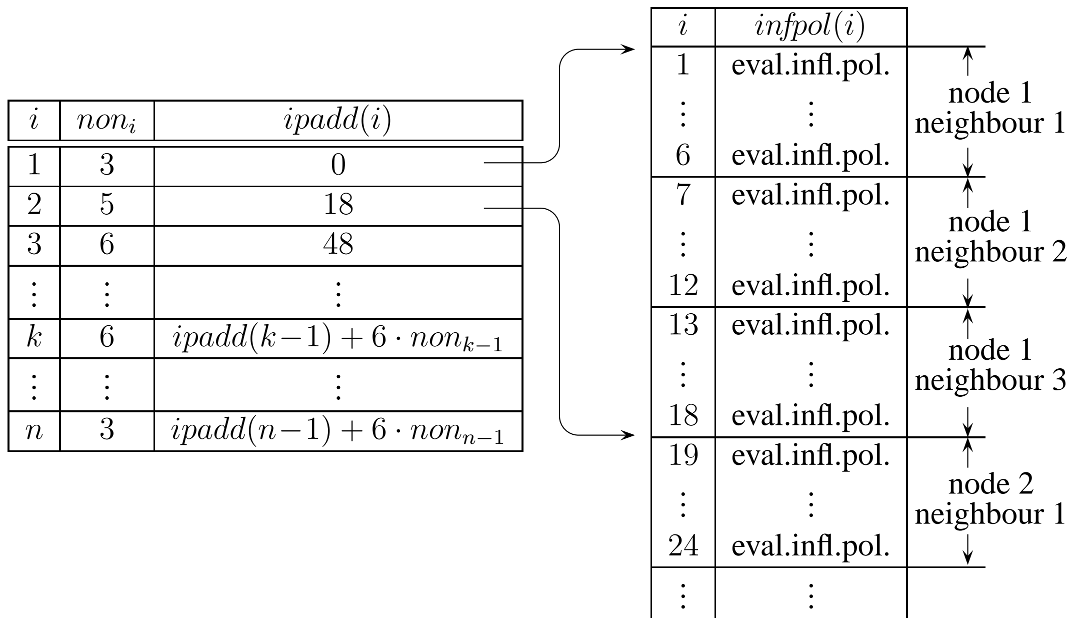


Figure 4.4.5: Illustration of the array *infpol* and *ipadd*.

To get a better performance on a vector computer we do the interpolation of the solution for all new nodes at the same time, so all necessary data must be available then. First we copy the two difference stars of the end points of each edge into an **integer array *ngrid*** with length equal to 6, which is the number of nodes in a difference star of consistency order $q = 2$, and width equal to $2 \cdot r_{cnt}$. Here the data for nod_1 is stored in the odd columns and the data for nod_2 is stored in the even columns in each case:

$$\left.\begin{array}{lcl} ngrid(j, 2{\cdot}i{-}1) & = & ngrid_2(nod_1, j) \\ ngrid(j, 2{\cdot}i) & = & ngrid_2(nod_2, j) \end{array}\right\} \quad \text{for} \quad j = 1, \ldots, 6 \tag{4.4.9}$$

for $i = 1, \ldots, r_{cnt}$. Again we need the information which neighbour nod_1 is of nod_2 and the other way round in order to determine the starting address of the influence polynomials in *infpol*. These starting addresses of all end points are stored in a one-dimensional **integer array *ia*** with length equal to $2 \cdot r_{cnt}$. Again the data for nod_1 is stored in the odd rows and that for nod_2 in the even rows.

$$\left.\begin{array}{lcl} ia(2{\cdot}i{-}1) & = & nb_{1,i} \\ ia(2{\cdot}i) & = & nb_{2,i} \end{array}\right\} \quad \text{for} \quad i = 1, \ldots, r_{cnt}. \tag{4.4.10}$$

Then the interpolation of the solution is done by

$$\begin{array}{lcl} u_{n+i,1} & = & \displaystyle\sum_{j=1}^{6} \big(\mathit{infpol}(ia(2{\cdot}i{-}1){+}j) \cdot u_{ngrid(j,2\cdot i-1)} \big) \\ u_{n+i,2} & = & \displaystyle\sum_{j=1}^{6} \big(\mathit{infpol}(ia(2{\cdot}i){+}j) \cdot u_{ngrid(j,2\cdot i)} \big) \end{array} \tag{4.4.11}$$

for $i = 1, \ldots, r_{cnt}$. The new function value is the mean value of the two interpolations for the end points of the edge:

$$u_{n+i} = \frac{u_{n+i,1} + u_{n+i,2}}{2} \quad \text{for} \quad i = 1, \ldots, r_{cnt}. \tag{4.4.12}$$

This interpolation is done for each of the l components, and the new function values are stored in the solution array *u* with length equal to n_{max} and width equal to l.

Now the new node numbers are entered into the array for the node numbers of an element, i.e. the *nek* array, for the refinement elements. In *rtl*, the array for the refinement edges (see table 4.4.1), we have stored for each edge (and thus for each new node i) the number of the element el_i and the local node number pos_i in element el_i, therefore we set

$$nek(el_i, pos_i) = n + i \quad \text{for} \quad i = 1, \ldots, r_{cnt}, \tag{4.4.13}$$

```
! for all refinement edges
do j = 1,r_cnt
  ! saddr = starting addr. of curr. edge in rtl
  saddr = (j-1)*nbm_6

  ! get new node number, element number
  ! and local position of node from rtl
  newn = rtl(saddr+1)
  el   = rtl(saddr+4)
  pos  = rtl(saddr+5)

  ! insert new node number into nek
  nek(el,pos) = newn
end do
```

Listing 10: Code for the update of the *nek* array for refinement elements.

see listing 10. Recall that the positions 4 to 6 in the *nek* array are for the mid-points of the corresponding edges.

Then we enter the new node numbers into the *nek* array for the neighbour elements of the refinement elements. Here we only stored the numbers of the neighbour elements but not the local node number of the new nodes in these elements. So we must determine the right position by an "`if ... then ... else ... end if`" construction. We go through the refinement edges stored in *rtl* and compare the two end points of the current refinement edge with the end points of the first edge of the current neighbour element. If they match, the new node number is stored in column 4 in the row for the neighbour element in array *nek*, otherwise we compare the end points with those of the second edge of the neighbour element. If they match, the node gets position 5, otherwise position 6. We store the new node numbers the same way as in (4.4.13), see listing 11.

If the 6th entry of an edge in *rtl* is zero (see table 4.4.1), i.e. there are not any neighbour elements, we have to check if the new node becomes an external boundary node. In 2-D this is very simple because the boundary edges are the only ones that have not got any neighbour elements. But we have to check to which boundary the new node belongs to. A new node is only then a boundary node if both end points of the edge are boundary nodes (on the same boundary). So we first determine the maximum number of external boundaries bd_{max} a boundary node belongs to. Then we need an **integer array *bdnr*** with length equal to the number of boundary nodes and width equal to $bd_{max} + 1$ where we store the **integer array *bnod***, where the numbers of the boundary nodes are stored in the first column, sorted in ascending order (by the external procedure `HEAPSORTIX`, see subsection 4.1.3). In the next columns we store the boundary numbers of the boundaries to which each node belongs. For the number of external boundaries a node belongs to we introduce an **integer**

```
! for all refinement edges
do j = 1,r_cnt
  ! saddr: starting addr. of curr. edge in rtl
  saddr = (j-1)*nbm_6

  ! get new node number, element number
  ! and end points of the edge from rtl
  newn  = rtl(saddr+1)
  nod_1 = rtl(saddr+2)
  nod_2 = rtl(saddr+3)
  el    = rtl(saddr+nbm_6)

  ! nodx: corner node x of cuurent element el
  nod1 = nek(el,1)
  nod2 = nek(el,2)
  nod3 = nek(el,3)

  ! set logical variables for edge search
  lnod11 = nod1==nod_1
  lnod12 = nod1==nod_2
  lnod21 = nod2==nod_1
  lnod22 = nod2==nod_2
  lnod31 = nod3==nod_1
  lnod32 = nod3==nod_2

  ! if edge is not a boundary edge search for
  ! local position of new node in element el
  if (el > 0) then
    if ((lnod11.and.lnod22).or.(lnod12.and.lnod21)) then
      pos = 4
    else if ((lnod21.and.lnod32).or.(lnod22.and.lnod31)) then
      pos = 5
    else
      pos = 6
    end if
    ! insert new node into nek
    nek(el,pos) = newn
  end if
end do
```

Listing 11: Code for the update of the *nek* array for neighbour elements.

array b_{cnt} with length equal to the number of external boundary nodes.

In the sorted array of the boundary nodes *bdnr* we can search for the two end points of an edge much more efficiently by a binary search than in the unsorted array *bnod* where we had to search linearly. If we find two end points that are boundary nodes, i.e. they both

occur in the array *bdnr*, we check to which boundary both of them belong and store this boundary number together with the number of the new boundary node (that is stored in the first entry of the refinement edge in *rtl*, see table 4.4.1), in an **integer array *newbn*** with length equal to r_{cnt} and width equal to 2. The code by which we determine if the two end points belong to the same boundary is shown in listing 12.

```
! increase number of boundary nodes
nblc = nblc+1
! insert new node number into newbn
newbn(nblc,1) = newn

bnri = bcnt(ibnd)          ! bnri/bnrj: nr. of boundaries
bnrj = bcnt(jbnd)          ! for node ibnd/jbnd
doi: do i = 1,bnri         ! for each boundary (ibnd)
  bi = bdnr(ibnd,1+i)      ! bi: number of boundary (ibnd)
  do j = 1,bnrj            ! for each boundary (jbnd)
    bj = bdnr(jbnd,1+j)    ! bj: number of boundary (jbnd)
    if (bi == bj) then     ! check if bd. numbers are equal
      bd = bi              ! bd: boundary number for new node
      exit doi             ! exit if boundary is found
    end if
  end do
end do doi

! insert boundary number for new node into newbn
newbn(nblc,2) = bd
! only used for sliding dividing line nodes
newbn(nblc,3) = nod_1
```

Listing 12: Code for the determination of new external boundary nodes and sliding dividing line nodes in 2-D.

RECAPITULATION: We finished the generation of the new nodes now. A new node is always created in the middle of a refinement edge and it gets the lesser of the consistency orders of the end points. The function value for a new node is interpolated by the combination of the function values and the evaluated influence polynomials for the nodes in the difference stars of the two end points of the edge. From these function values interpolated from the left and from the right end point we compute a mean value that finally represents the new function value of the node. The evaluated influence polynomials have been computed and stored during the generation of the difference formulas. In 2-D a node is then and only then an external boundary node if both end points of the edge belong to the same external boundary and if there is not a neighbour element at this edge. The node numbers are inserted into the *nek* array where we store the node numbers for each element, for the refinement element itself we have stored the

position in the *rtl* array, for the neighbour elements we must check this by a conditional if-construct.

We have to insert the new external boundary nodes into the *bnod* array. Like for the new nodes and elements we have to check if we exceeded the limit for the boundary nodes nb_{max} that is prescribed by the user. So we

$$\text{check if} \;\; neb + neb_{new} > nb_{max}$$

where neb_{new} is the number of new boundary nodes in the current refinement step. If yes, we print out an error message and stop the computation. Otherwise we append the contents of the *newbn* array to the information array for the external boundary nodes, i.e. the *bnod* array.

SUMMARY: At the end of a refinement step the new elements are generated. We first issue the new local and global element numbers and afterwards separate the old elements into four elements of the higher refinement stage in 2-D. Here we choose for each of the four new elements three nodes out of the six nodes of the refined element that represent the corner nodes of the new triangle.

The next point is the generation of the new elements out of the refinement elements. Although we generate four elements out of the old one in 2-D, there are only three new element numbers as one element gets the old number of the refinement element. The last old element has got the number ne and we continue counting the new elements by the following natural numbers, so the new elements get the local (and because of a single processor also global) element numbers

$$\begin{aligned} \text{from} \quad & ne_{first} = ne + 1 \\ \text{to} \quad & ne_{last} \;\, = ne + 3 \cdot ne_{re}. \end{aligned}$$

For the refinement elements the rows in the *nek* array, the array for the node numbers of an element, are full, i.e. we have stored a node number in each of the six columns (three corner nodes and three mid nodes) so that we can generate four new elements, see figure 4.4.3. The element that contains the first local node gets the old element number of the refinement element, but it is the last of the new elements that is generated because otherwise we would overwrite the element information for the other elements. The selection of the nodes for the four new elements is described in the next paragraph. First we store the information in the element information array *nenr* (see on page 78) for the three elements which get new numbers. This information is for each element:

$$\left.\begin{aligned} nenr(ne+i,1) &= ne+i \\ nenr(ne+i,2) &= 1 \\ nenr(ne+i,3) &= 0 \\ nenr(ne+i,4) &= 1 \end{aligned}\right\} \quad \text{for} \quad i = 1, \ldots, 3 \cdot ne_{re}. \tag{4.4.14}$$

In the second column of array *nenr* we store the number of the subdomain the new elements belong to which is 1 if the domain is not divided into several subdomains by dividing lines or sliding dividing lines and the third column we set equal to zero for all elements. This column also is only of account for domains with dividing lines or sliding dividing lines. We also store the number of the own processor of the new elements in the fourth column of *nenr* which is 1 for a single processor.

For the element with the old element number this is done the same way. Then the *nek* array is updated as shown in table 4.4.3 and figure 4.4.6. For example, the 4th new element consists of the 4th, 5th and 6th node of the old element el_{old}. This information is stored in an **integer array** $\boldsymbol{el_{hlp}}$ where we store in the jth row the local node numbers for the jth new element el_j. If we denote by el_{loc} the row in the array *nek* that corresponds to the local number of the new element, we set

$$nek(el_{loc}, k) = nek\left(el_{old}, el_{hlp}(el_j, k)\right) \quad \text{for} \quad k = 1, \ldots, 3 \tag{4.4.15}$$

for the three elements that get new element numbers, i.e. for $j = 2, \ldots, 4$.

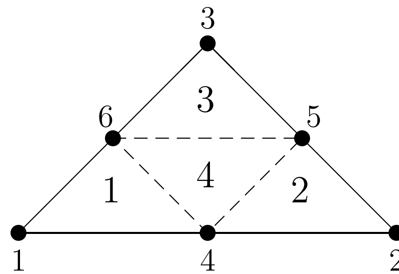


Figure 4.4.6: Illustration of array el_{hlp}.

For the example above, we yield by substituting the array el_{hlp}

$$\begin{aligned} nek(el_{loc}, 1) &= nek(el_{old}, 4), \\ nek(el_{loc}, 2) &= nek(el_{old}, 5), \\ nek(el_{loc}, 3) &= nek(el_{old}, 6). \end{aligned} \tag{4.4.16}$$

new element el_j	$el_{hlp}(el_j, k)$		
	$k=1$	$k=2$	$k=3$
$j=1$	1	4	6
$j=2$	2	5	4
$j=3$	3	6	5
$j=4$	4	5	6

Table 4.4.3: Update of the *nek* array information in 2-D for figure 4.4.6.

For the element that gets the old element number, i.e. for $j = 1$, we set

$$nek(el_{old}, k) = nek\,(el_{old}, el_{hlp}(el_1, k)) \quad \text{for} \quad k = 1, \ldots, 3. \tag{4.4.17}$$

As a newly-created element cannot have any mid-point on its edges the entries 4 to 6 are set equal to zero:

$$\begin{aligned} nek(el_{loc}, 4) &= 0, & nek(el_{old}, 4) &= 0, \\ nek(el_{loc}, 5) &= 0, & nek(el_{old}, 5) &= 0, \\ nek(el_{loc}, 6) &= 0, & nek(el_{old}, 6) &= 0. \end{aligned} \tag{4.4.18}$$

The refinement of a triangular element in 2-D is illustrated in figure 4.4.7. In figure 4.4.7a) you see the original refinement element, in figure 4.4.7b) we have inserted the newly-created nodes and the edges that connect them. Finally, in figure 4.4.7c) we have separated the four elements that are generated from the old refinement element.

The refinement stage of the four new elements is $rs + 1$, i.e. it is increased by one in comparison with the refinement stage rs of the old element. We insert the new value of the refinement stage for the old element el_{old} and the new elements el_{loc} into the **integer array *refst*** of length ne_{max}:

$$refst(el_{old}) = rs + 1, \quad refst(el_{loc}) = rs + 1. \tag{4.4.19}$$

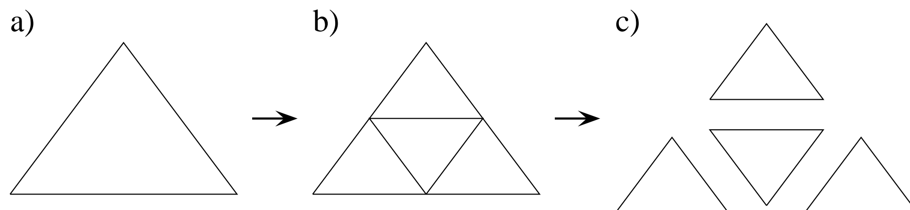


Figure 4.4.7: Illustration of the refinement of a triangle: a) original element, b) original element with new nodes and elements, c) divide element into four new ones.

At the end of each refinement step we compute the value in_{max} which is the maximum number of elements a node belongs to. This is only for the preparation of the next refinement step where we have to determine the inverted *nek* array that we need for the neighbour element search and therefore need the value in_{max} to allocate the array *nekinv* with the correct width.

By n_{rs} we denote the number of nodes and by ne_{rs} the number of elements that are stored on a processor before refinement step rs, i.e. it holds

$$\begin{aligned} n_0 &= n \\ ne_0 &= ne. \end{aligned} \tag{4.4.20}$$

We increase the number of nodes n_{rs} and the number of elements ne_{rs} that existed before the current refinement step by the number of generated new nodes r_{cnt} and by the number of generated new elements $3 \cdot ne_{re}$ in refinement step rs:

$$\begin{aligned} n_{rs+1} &= n_{rs} + r_{cnt} \\ ne_{rs+1} &= ne_{rs} + 3 \cdot ne_{re}. \end{aligned} \tag{4.4.21}$$

As we have subdivided the refinement elements of the current refinement step into smaller elements of a higher refinement stage, these new elements have the same refinement stage as those that have to be refined in the next refinement step. So if the refinement of an element of the current refinement step has been caused by the refinement cascade, the newly-created elements and the refinement element of the next refinement step that has caused the refinement cascade have the same refinement stage and therefore we are able to generate the mid-points of the refinement edges in the next refinement step.

There are not any differences to the first refinement step for the following refinement steps we have to pay attention to. So we can summarize a refinement step of the mesh refinement algorithm on a single processor by algorithm C on page 122. After the refinement of the refinement elements of all stages the sorting of the nodes by x-coordinate takes place and afterwards the next computing cycle can start, i.e. then the solution of the system of PDEs is computed on the new grid.

4.4.2 Extension to 3-D

Important notations:

bnod: integer array for external boundary nodes
infpol: double precision array for evaluated influence polynomials
lbnd: logical array for computing 3-D external boundary nodes
nek: integer array for node numbers of the elements
rtl: integer array for refinement edge information

There are not many differences for the mesh refinement in 3-D on a single processor. An element may have more than one neighbour at an edge, so that

$$nnb_{max} > 1 \quad \text{in 3-D}$$

holds where nnb_{max} is the maximum number of neighbour elements per edge of a refinement element in the current refinement step. Accordingly, the maximum number of neighbour elements per element is the product of nnb_{max} and the number of edges per element, $n_{edge} = 6$. So we had to allocate the array *nb* where we store the neighbour elements for each edge of a refinement element, with length ne_{re} and width $n_{edge} \cdot nnb_{max}$. Unfortunately, the value nnb_{max} is not known until we have inserted the neighbour element information into the arrays *nb* and *nbadd*. So we have to choose a value greater or equal to nnb_{max} for the determination of the width of *nb*. Another possibility was to proceed like for the array *nekinv* where we first go through the elements and only determine the number of elements per node without inserting them into *nekinv*. As the array *nb* is comparatively small, we accept the loss of storage place to improve the performance. The condition is fulfilled for in_{max}, the maximum number of elements per node, so that the width of *nb* is set to $n_{edge} \cdot in_{max}$.

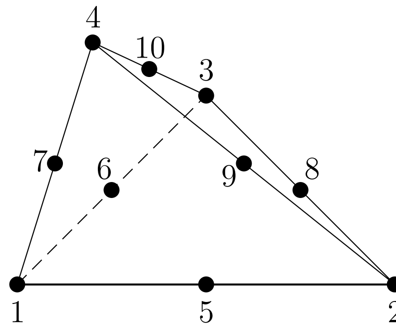


Figure 4.4.8: Local numbering of the nodes in an element (3-D).

We have to allocate the *rtl* array where we store the refinement edge information with length $nbm_6 \cdot ne \cdot n_{edge}$ with

$$nbm_6 = nnb_{max} + 6. \tag{4.4.22}$$

For this purpose we go through the refinement elements of the current refinement stage and for each of these elements we go through all edges and enter the refinement edge information into *rtl* for those edges that are not yet halved and that we did not enter into the *rtl* array before. The “6” in (4.4.22) comes from the fact that we have to store (like in 2-D) the number of the new node and of the two end points of the edge, the element number, the local position of the new node in the element and the number of neighbour elements

in the *rtl* array. By pos we denote the position of the node in the refinement element, see figure 4.4.8. All the neighbour elements are inserted into the *rtl* array as indicated in table 4.4.4, the number of neighbour elements is stored in entry $r_{cnt} + 6$. This information is easily obtained from the *nekinv* array that gives for each node the elements it belongs to. To find the neighbour elements of an edge we proceed like described for 2-D on page 87.

i	$rtl(i)$	description
$\vdots$	$\vdots$	$\vdots$
	{ preceding edge }	
$r_{cnt} + 1$	left free	
$r_{cnt} + 2$	nod_1	local number of end point 1 of edge j
$r_{cnt} + 3$	nod_2	local number of end point 2 of edge j
$r_{cnt} + 4$	el	local refinement element number
$r_{cnt} + 5$	pos	local node number in element (see text)
$r_{cnt} + 6$	k	number of neighbour elements
$r_{cnt} + 7$	$el_{nb,1}$	local number of neighbour element 1
$\vdots$	$\vdots$	$\vdots$
$r_{cnt} + 6 + k$	$el_{nb,k}$	local number of neighbour element k
$\vdots$	$\vdots$	$\vdots$
$r_{cnt} + nbm_6$	$el_{nb,nnb_{max}}$	local number of neighbour element nnb_{max} (= 0 if $k < nnb_{max}$, k = number of neighbour elements)
	{ following edge }	
$\vdots$	$\vdots$	$\vdots$

Table 4.4.4: Illustration of the contents of array *rtl* in 3-D (edge information).

For the new nodes we also have to compute a new z-coordinate like we compute the x- and y-coordinate in (4.4.6):

$$\left.\begin{aligned} x_{n+i} &= \tfrac{x_{nod_{1,i}}+x_{nod_{2,i}}}{2} \\ y_{n+i} &= \tfrac{y_{nod_{1,i}}+y_{nod_{2,i}}}{2} \\ z_{n+i} &= \tfrac{z_{nod_{1,i}}+z_{nod_{2,i}}}{2} \end{aligned}\right\} \quad \text{for} \quad i = 1, \ldots, \frac{r_{cnt}}{nbm_6}. \tag{4.4.23}$$

Note that we have to divide r_{cnt} by nbm_6 now because in the following r_{cnt} represents the number of refinement edges:

$$r_{cnt} \Leftarrow \frac{r_{cnt}}{nbm_6}. \tag{4.4.24}$$

This also holds for (4.4.4) and (4.4.7). For the interpolation of the function values of the new nodes with a polynomial of order $q = 2$ we need $m = 10$ nodes and therefore we have 10 evaluated influence polynomials for each neighbour node. So the "6" in equations (4.4.9) and (4.4.11) is replaced by "10", therefore (4.4.9) becomes

$$\left.\begin{array}{lcl} ngrid(j, 2{\cdot}i{-}1) & = & ngrid_2(nod_1, j) \\ ngrid(j, 2{\cdot}i) & = & ngrid_2(nod_2, j) \end{array}\right\} \quad \text{for} \quad j = 1, \ldots, 10 \tag{4.4.25}$$

for $i = 1, \ldots, r_{cnt}$ and (4.4.11) is changed into

$$\begin{array}{lcl} u_{n+i,1} & = & \sum_{j=1}^{10} \big(\mathit{infpol}(ia(2{\cdot}i{-}1){+}j) \cdot u_{ngrid(j,2\cdot i-1)} \big) \\ u_{n+i,2} & = & \sum_{j=1}^{10} \big(\mathit{infpol}(ia(2{\cdot}i){+}j) \cdot u_{ngrid(j,2\cdot i)} \big) \end{array} \tag{4.4.26}$$

for $i = 1, \ldots, r_{cnt}$ and for each of the l components. According to (4.4.8), the length of the array *infpol* where we store the evaluated influence polynomials during the generation of the difference and error formulas is

$$l_{\mathit{infpol}} = 10 \cdot \sum_{i=1}^{n} non_i \tag{4.4.27}$$

where n here denotes the number of nodes before the mesh refinement process and non_i is the number of neighbour nodes of node i.

A point that is more difficult than in 2-D is the checking of the boundary node property. In 2-D we only have to check the edges that have not got any neighbour element, in 3-D we must check all edges. In 3-D it is not sufficient to check if the two end points of the edge are boundary nodes. A new node may be a boundary node although there are neighbour elements of the refinement element on the edge concerned. As we have tetrahedrons in 3-D, there are four nodes in each element that shares this edge, so we have two nodes that are not end points of the edge under consideration. We have to check if each of these "free" nodes belongs to exactly two elements that share the edge so that the concerned elements form a kind of ring around the edge, see the description below. Then the edge is not a part of the boundary and therefore the new node is not a boundary node, see figure 4.4.9b). For the edge in figure 4.4.9a) we have nodes that do not occur twice in the surrounding elements of the edge (more precisely, node 3 in element 1 and node 7 in element 4 do not occur in a second element) and therefore the new node is a boundary node. We use a one-dimensional **logical array *lbnd*** of length n (that is initialized with *true*) where we set

$$lbnd(nod) = \neg\, lbnd(nod)$$

for a node nod each time this node is found in an element that shares the edge, i.e. in the refinement element and its neighbour elements. When we have treated all these elements and there is at least one entry in *lbnd* with the value *false* the new node is a boundary node. We explain this by table 4.4.5 where table 4.4.5a) corresponds to figure 4.4.9a) and table 4.4.5b) corresponds to figure 4.4.9b). Both tables have a dynamic component, i.e. after having inserted the nodes of element 1 into *lbnd* the array looks like it is shown in row 1, after having inserted the nodes of element 2 *lbnd* looks like shown in row 2 and so on. At the end we have in row 4 of table 4.4.5a) two entries left with the value *true*. They belong to the nodes 3 and 7 which both occur only once in figure 4.4.9a), the new node becomes a boundary node. In the last row of table 4.4.5b) all entries of *lbnd* have the value *false* and therefore each node occurs twice in the elements that share the edge, so that the new node is an inner node. The code for the determination of the new boundary nodes in 3-D is shown in listing 13.

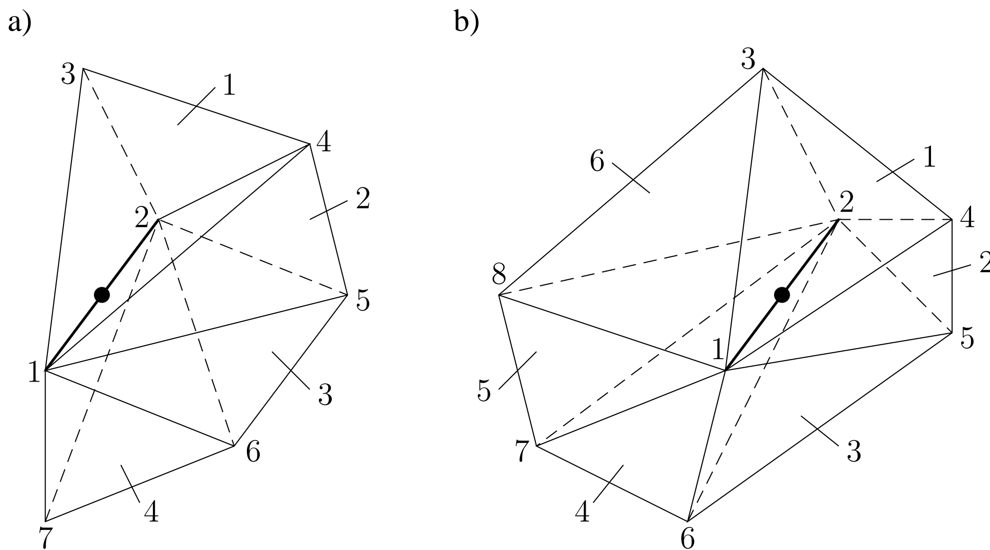


Figure 4.4.9: Illustration of the boundary node property (3-D): new node on refinement edge (bold) is a) a boundary node, b) not a boundary node.

The generation of the new elements is done like in 2-D, it is even used the same code. Only the number of new elements that is stored in el_{new} is seven (there result eight elements, one gets the old number, so seven new numbers are created) instead of three in 2-D so that the new element numbers go

$$\begin{aligned} \text{from} \quad ne_{first} &= ne + 1 \\ \text{to} \quad ne_{last} &= ne + 7 \cdot ne_{re}. \end{aligned}$$

The value of el_{new} also changes in the equations (4.4.14) and (4.4.21). The array el_{hlp} where we store the information which nodes of the old element form the new elements is

a)

node / el.	*lbnd*(k) 1	2	3	4	5	6	7
1	F	F	T	T	F	F	F
2	F	F	T	F	T	F	F
3	F	F	T	F	F	T	F
4	F	F	T	F	F	F	T

b)

node / el.	*lbnd*(k) 1	2	3	4	5	6	7	8
1	F	F	T	T	F	F	F	F
2	F	F	T	F	T	F	F	F
3	F	F	T	F	F	T	F	F
4	F	F	T	F	F	F	T	F
5	F	F	T	F	F	F	F	T
6	F	F	F	F	F	F	F	F

Table 4.4.5: Use of array *lbnd* for assignment of boundary node property (3-D): new node is a) a boundary node, b) not a boundary node, see figure 4.4.9.

also different. This is shown in table 4.4.6, see also figure 4.4.8.

In the jth row of el_{hlp} the local node numbers for the jth new element el_j are stored. If we denote by el_{loc} the row in the array *nek*, the array where we store the node numbers for the elements, that corresponds to the local number of the new element, we set

$$nek(el_{loc}, k) = nek\left(el_{old}, el_{hlp}(el_j, k)\right) \quad \text{for} \quad k = 1, \ldots, 4 \tag{4.4.28}$$

for the seven elements that get new numbers, i.e. for $j = 2, \ldots, 8$. For $j = 1$, i.e. for the element that gets the old element number, we have

$$nek(el_{old}, k) = nek\left(el_{old}, el_{hlp}(el_1, k)\right) \quad \text{for} \quad k = 1, \ldots, 4. \tag{4.4.29}$$

As a newly-created element cannot have any mid-point on its edges the entries 5 to 10 for

new element el_j	$el_{hlp}(el_j, k)$ $k=1$	$k=2$	$k=3$	$k=4$
$j=1$	1	5	6	7
$j=2$	5	2	8	9
$j=3$	6	3	8	10
$j=4$	7	4	10	9
$j=5$	5	6	8	10
$j=6$	5	6	7	10
$j=7$	5	7	9	10
$j=8$	5	8	9	10

Table 4.4.6: Update of the *nek* array information in 3-D.

```
      lbnd = .false.              ! initialize lbnd

      do i = 1,noel               ! for each neighbour element
        el = rtl(nbm_6*(j-1)+6+i) ! el: number of neighbour elem.
        do k = 1,elpt             ! for each corner node of el
          node = nek(el,k)        ! node: number of corner node
          ! if node is not an end point of the
          ! current edge negate entry in lbnd
          if ((node /= nod_1).and.(node /= nod_2))
     &      lbnd(node) = .not.lbnd(node)
        end do
      end do

      el = rtl(nbm_6*(j-1)+4)     ! el: number of ref. element
      do k = 1,elpt               ! same loop as above
        node = nek(el,k)
        if ((node /= nod_1).and.(node /= nod_2))
     &    lbnd(node) = .not.lbnd(node)
      end do

      ! check if there are true-entries in lbnd
      if (any(lbnd)) then
        bnri = bcnt(ibnd)         ! bnri/bnrj: nr. of boundaries
        bnrj = bcnt(jbnd)         ! for node ibnd/jbnd
        doi: do i = 1,bnri        ! for each boundary (ibnd)
          bi = bdnr(ibnd,1+i)     ! bi: number of boundary (ibnd)
          do k = 1,bnrj           ! for each boundary (jbnd)
            bj = bdnr(jbnd,1+k)   ! bj: number of boundary (jbnd)
            if (bi == bj) then    ! check if bd. numbers are equal
              ! increase nr. of new bd. nodes and insert
              ! boundary node information into newbn
              nblc = nblc+1
              newbn(nblc,1) = newn
              newbn(nblc,2) = bi
              ! only used for sliding dividing line nodes
              newbn(nblc,3) = nod_1
            end if
          end do
        end do doi
      end if
```

Listing 13: Code for the determination of new external boundary nodes and sliding dividing line nodes in 3-D.

the mid-points are set equal to zero:

$$\left.\begin{array}{rcl} nek(el_{loc},k) & = & 0 \\ nek(el_{old},k) & = & 0 \end{array}\right\} \quad \text{for} \quad k = 5,\ldots,10. \tag{4.4.30}$$

The refinement of a 3-D tetrahedron is illustrated in figure 4.4.10 where you can see the original refinement element in figure 4.4.10a). In figure 4.4.10b) we inserted the mid-points of the old edges and connected them so that we have got the edges of the new elements. In figure 4.4.10c) we take away the corner elements, i.e. we create the first four elements. There is left a kernel that consists of eight triangular surfaces. This kernel is divided into another four elements, see figure 4.4.10d).

The refinement stage of the four new elements is $rs + 1$, i.e. it is increased by one in comparison with the refinement stage rs of the old element. The current refinement stage is inserted into array *refst* just like for 2-D, see (4.4.19).

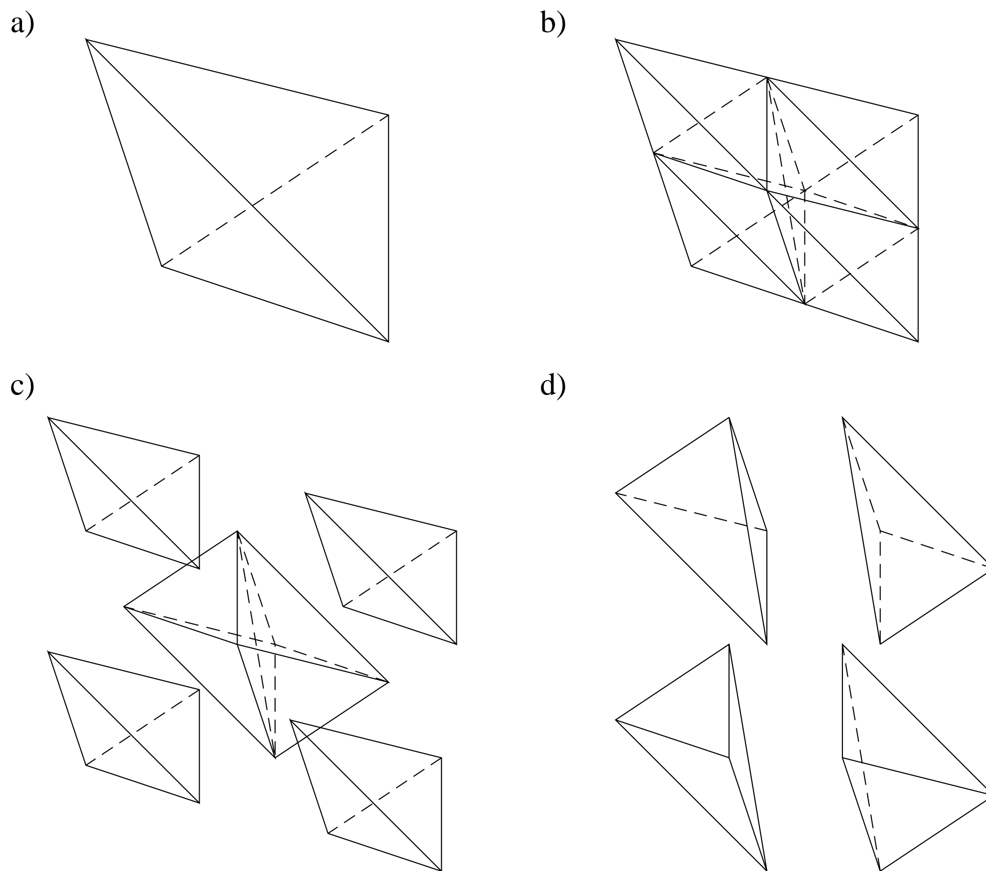


Figure 4.4.10: Illustration of the refinement of a tetrahedron: a) original element, b) original element with new nodes and edges, c) take away the four corner elements, d) divide kernel into four elements.

4.4.3 Mesh refinement with dividing lines in 2-D and 3-D

SUMMARY: First we explain the differences of the neighbour element search in comparison to problems with only one domain and afterwards we add the missing entry in the array *dlote* for the twin nodes on the other side of a dividing line. To identify refinement edges that are dividing line edges at the same time, we have to label them by a key number. For 3-D problems we have to take care of the determination of the new external boundary nodes.

Important notations:

dlote:	integer array for twin node information of dividing line edges
dloteadr:	integer array for starting addresses of dividing line elements in *dlote*
logarr:	logical array for computation of new dividing line elements
nenr:	integer array for element information
nnr:	integer array for node information
rtl:	integer array for refinement edge information
snod:	integer array for sliding dividing line nodes
tnod:	integer array for dividing line nodes

If we have dividing lines in the solution domain we also have to note some differences in the refinement process. If the refinement element is a dividing line element and we search for the neighbour elements of a refinement element at the beginning of a refinement step, we also have to search for elements on the other side of the dividing line. The preparation of this neighbour search has been described in section 4.3 where we explained how we store the information about the twin nodes on the other side of a dividing line in the arrays *dlote* and *dloteadr*—in *dloteadr* we store the starting address of the dividing line element information in *dlote* and in *dlote* we store the information for each dividing line edge of the dividing line elements, see figure 4.3.4. With the help of these two arrays we can now easily search for neighbour elements on the other side of the dividing line and store the element numbers in *nb*, the array for the neighbour element numbers of the edges of an element. The width of *nbadd*, i.e. the array for the starting addresses of the edges in *nb*, is $2 \cdot n_{edge} + 1$ because we do not only store the starting address of the neighbour elements on this side of the dividing line in *nb* but also the starting address of the neighbour elements on the other side of the dividing line. Again we store the end address of the last edge in the last column.

To illustrate the shape of the two arrays *nb* and *nbadd* have a look at figure 4.4.11. Here we only show the row that corresponds to the arbitrarily chosen element 42 forgoing the element numbers of the neighbour elements, and as we have at most one neighbour element in 2-D we illustrate the much more difficult 3-D situation. In *nbadd* we have two entries for each edge plus one entry to mark the end of the last edge. In the first entry of an

el	$nbadd(el,k)$												
	1	2	3	4	5	6	7	8	9	10	11	12	13
42	1	5	8	11	16	20	20	25	25	28	33	38	38

el	$nb(el,k)$
42	0 $\cdots$ 0

e_{11} e_{12} e_{21} e_{22} e_{31} e_{41} e_{51} e_{52} e_{61}

e_{ij}: edge i, side j

$n_{edge} \cdot in_{max}$

Figure 4.4.11: Illustration of the arrays *nb* and *nbadd* for (dividing line) element 42 (3-D).

edge we insert the column number in *nb* where the neighbour elements on this side (side 1) of the dividing line start and in the second entry we insert the column number where the neighbour elements on the other side (side 2) of the dividing line start. If an edge is not a dividing line edge the second entry points to the same column as the first entry of the following edge because there are not any neighbour elements on the other side of a dividing line, and therefore the column number must not be increased.

Let the edges 1, 2 and 5 of element 42 in figure 4.4.11 be dividing line edges. The starting address of the neighbour elements of the first edge on this side of the dividing line is always one, and the first element has 4 neighbour elements that share this edge. So the neighbour elements on the other side of the dividing line begin in entry 5 of row 42 of *nb*. There we have 3 neighbour elements. For edge 2 we have 3 neighbour elements on this side of the dividing line and 5 on the other side. So the 4 neighbour elements for edge 3 start at entry 16. The starting address of the neighbour elements on the other side of the dividing line of the 3rd edge therefore is 20, but as edge 3 is not a dividing line edge, there is no other side of the dividing line and the first neighbour element of the 4th edge is stored in entry 20 of *nb*. After the 5 neighbour elements of edge 4 follow the 3 neighbour elements of the 5th edge on this side and the 5 neighbour elements on the other side of the dividing line. Finally, 5 elements share edge 6.

In *dlote* we store the number of coupling subdomains and the twin node numbers on the other side of each dividing line for each dividing line edge of the dividing line elements as explained on page 71. Furthermore, we store in *dlote* the number of neighbour elements nr_{el} on the other side of the dividing line in the entries we left free when we executed the refinement cascade, see on page 71. The neighbour elements on the other side of the dividing line are those elements that contain both twin nodes on the other side of the

dividing line that are already stored in *dlote*. According to the notation there we set

$$dlote(addr_{edge\,j} + 3 \cdot k - 2) = nr_{el} \tag{4.4.31}$$

for edge j and the kth coupling subdomain.

After the search for the neighbour elements we sort the array *tnod* with the dividing line nodes by ascending node number (by the external procedure `HEAPSORTIX`, see subsection 4.1.3) and store the information, to which dividing lines a node belongs to, in an **integer array *dl*** with length equal to nib_n and width equal to $2 \cdot n_{inb} + 1$ where n_{inb} is the number of dividing lines and nib_n is the number of dividing line nodes including the overlap. In the first column we store the number of dividing lines a node belongs to and in the other columns we store the numbers of the dividing lines. As it is impossible that any node belongs to all dividing lines we fill the rest of the columns where we have no entries with zeros.

In 2-D, each element may have at most one neighbour element. For a dividing line edge, there must be—because of matching grids—a neighbour element, and this element is always on the other side of the dividing line, and there are not any neighbour elements on this side of the dividing line, i.e. in the subdomain the refinement element belongs to. So when we store the information for the new nodes in the refinement edge array *rtl* (see table 4.4.1 for 2-D, table 4.4.4 for 3-D), we can use the last entry for a new node in *rtl* to store a key number to mark dividing line nodes. This key number is -1, it means that the edge information for a twin node is following next. The edge information for the new twin node on the other side is stored directly behind the information for the new node on this side of the dividing line. For this node the last entry is set to 0, see figure 4.4.12a). In 3-D, where we also have neighbour elements on this side of the dividing line, we have to set the length of *rtl* to $(nnb_{max} + 7) \cdot ne \cdot n_{edge}$ so that we have an additional entry for the key number -1. Thus it holds

$$nbm_6 = nnb_{max} + 7. \tag{4.4.32}$$

There may be more than one twin node in 3-D and we set the last entry to -1 for each of them except the last one where the entry is set to 0:

$$\left.\begin{aligned} rtl(r_{cnt} + 7) &= -1 \\ rtl(r_{cnt} + 14) &= 0 \end{aligned}\right\} \text{ in 2-D}$$

$$\left.\begin{aligned} rtl(r_{cnt} + k \cdot nbm_6) &= -1 \quad \text{for} \quad k = 1, \ldots, n_{sect} - 1 \\ rtl(r_{cnt} + n_{sect} \cdot nbm_6) &= 0 \end{aligned}\right\} \text{ in 3-D}$$

(see figure 4.4.12b) for a quadruple point).

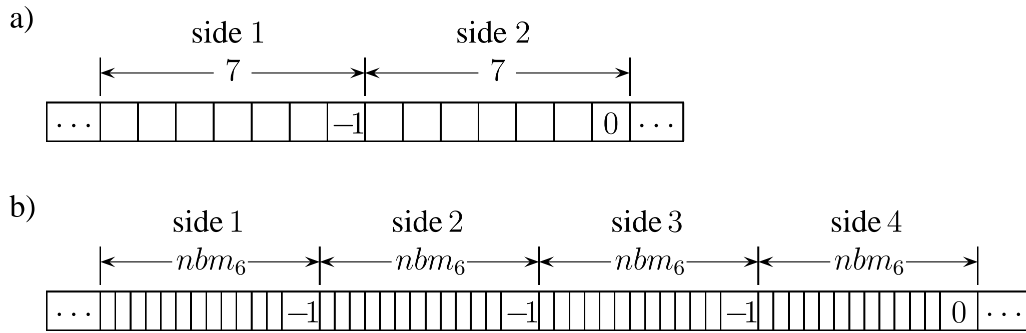


Figure 4.4.12: Illustration of array *rtl* for a dividing line edge a) in 2-D, b) in 3-D.

When we enter the new nodes into the node information array *nnr*, the entries for the second and third column are different from (4.4.4). We set

$$\left.\begin{array}{lcl} nnr(n+i,1) & = & n+i \\ nnr(n+i,2) & = & nnr(nod_1,2) \\ nnr(n+i,3) & = & n_{sect,cpl} \\ nnr(n+i,4) & = & 1 \end{array}\right\} \quad \text{for} \quad i=1,\ldots,\tfrac{r_{cnt}}{7}. \tag{4.4.33}$$

where $nnr(nod_1,2)$ is the number of the subdomain the first end point of the edge belongs to. Both end points of an edge always belong to the same subdomain, so we could have chosen the second end point, too. The new mid-point belongs to the same subdomain as the two end points of an edge. $n_{sect,cpl}$ is the number of subdomains the node couples to.

Another critical point is the determination of the new boundary nodes in 3-D. In 2-D there is no difference to the determination of the boundary node property in 2-D without dividing lines because an external boundary node may never be a dividing line node at the same time. When a dividing line meets an external boundary the intersection consists of only one node in 2-D, but in 3-D the intersection of a dividing surface and an external boundary consists of a bunch of edges. Like for 3-D without dividing lines we have to check if each of the non-edge nodes of an element that shares the edge occurs twice as described on page 106 for 3-D problems without dividing lines. For 3-D with dividing lines we have the exception that we do not have to check this for the dividing line nodes because we know that they must have a twin node on the other side of the dividing line.

SUMMARY: Of course, completely new is the determination of new dividing line nodes. Each new dividing line node creates as many rows in the array *tnod* for the dividing line nodes as it occurs on different dividing lines. After the node itself we must also create the rows for all its twin nodes. Each new dividing line node is assigned a dividing line

```
! posinod: position of node nod_1 in sorted array tnodst
posinod = BINSCH(tnodst(1,1),index(1,1),ctr,nod_1)

nrdl = dl(posinod,1)      ! dl: number of DLs nod_1 belongs to
snr  = tnodst(posinod,3) ! snr: subdomain nr. nod_1 belongs to
do j = 1,nrdl            ! for each DL
  ! insert node nr., DL and subdomain number into tnod
  tnod(nib_l+j,1) = pt
  tnod(nib_l+j,2) = dl(posinod,1+j)
  tnod(nib_l+j,3) = snr
end do
! nrdlsm: number of rows inserted into tnod so far
nrdlsm = nrdl

! initialize sect and insert new node number
sect = 0
sect(snr) = pt
```

Listing 14: Code for entering of the new dividing line node data into array *tnod* (first node).

number and a subdomain number. At the end the information to which subdomains the coupling nodes belong is inserted in all rows of *tnod* we just created.

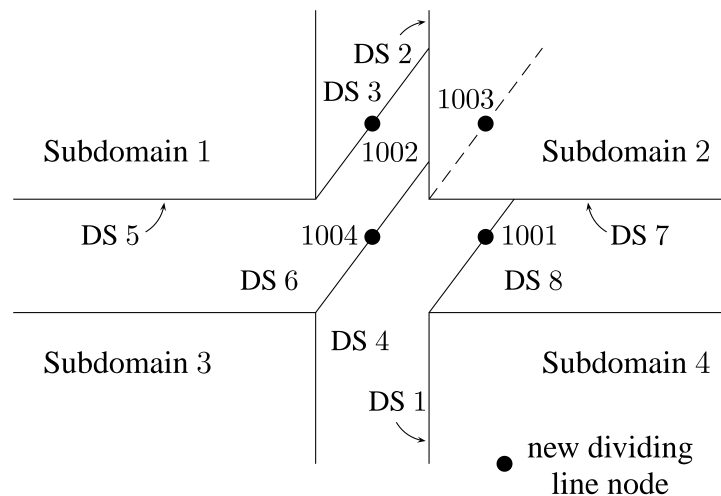
In order to determine the new dividing line nodes we go through the refinement edge array *rtl* (see table 4.4.1 for 2-D, table 4.4.4 for 3-D) and first check for each refinement element if it is a dividing line element, i.e. we check in *dloteadr* and *dlote* (see figure 4.3.4) if, for the current element and the current edge, the number of subdomains the edge couples to is greater than zero. As we store in *dloteadr* the starting addresses of the dividing line element information in *dlote*, only the entries for the dividing line elements are unequal to zero. For these elements we check if the entry for the number of subdomains for the current edge in *dlote* is unequal to zero. If not, the newly-created node is not a dividing line node because only the entries for the dividing line edges are unequal to zero. Otherwise, we have to determine the number of dividing lines the new node belongs to, and then we insert the new node into the information array for the dividing line nodes, i.e. the **integer array *tnod***, once for each dividing line. We store in the columns of *tnod* the number of the dividing line node, the numbers of the dividing line and the subdomain the node belongs to, and in the columns 4 to $n_{sect}+3$ we store the numbers of the coupling nodes for the n_{sect} subdomains, see table in figure 4.4.13. At first, we only enter the node number, the dividing line number and the number of the subdomain into *tnod*. As we did not yet generate the coupling nodes, we cannot enter their numbers yet. In a one-dimensional auxiliary **integer array *sect*** with length equal to n_{sect}, i.e. the number of subdomains, we store the node number at the position of the subdomain, e.g. if node 1 is in subdomain 2 we set $sect(2) = 1$. The code for the entering of the first new dividing line node into the array *tnod* is shown in listing 14.

```
do i = 1,nr                    ! for each coupling subdomain
  addr  = glob1(cnt)           ! addr: starting addr. for edge
  pt    = rtl(addr)            ! pt: new node number
  nod_1 = rtl(addr+1)          ! nod_1: first end point of edge
  nod_2 = rtl(addr+2)          ! nod_2: second end point of edge
  el    = rtl(addr+3)          ! el: number of ref. element
  k     = rtl(addr+4)-elpt     ! k: pos. of new node in elem. el

  ! posinod: position of node nod_1 in sorted array tnodst
  posinod = BINSCH(tnodst(1,1),index(1,1),ctr,nod_1)

  nrdl = dl(posinod,1)       ! dl: nr. of DLs nod_1 belongs to
  snr  = tnodst(posinod,3)   ! snr: subdom. nr. nod_1 belongs to
  do j = 1,nrdl              ! for each DL
    addr = n_ibl+nrdlsm+j    ! compute row in tnod
    ! insert node nr., DL and subdomain number into tnod
    tnod(addr,1) = pt
    tnod(addr,2) = dl(posinod,1+j)
    tnod(addr,3) = snr
  end do
  ! nrdlsm: number of rows inserted into tnod so far
  nrdlsm = nrdlsm+nrdl

  ! insert node nr. into sect
  sect(snr) = pt

  ! increase counter cnt
  cnt = cnt+1
end do

! insert node numbers from sect into each new row in tnod
do i = 1,nrdlsm
  do j = 1,n_sect
    tnod(nib_l+i,3+j) = sect(j)
  end do
end do
```

Listing 15: Code for entering of the new dividing line node data into *tnod* (twin nodes).

Afterwards we also have to insert into *tnod* all the twin nodes of the new node for all the dividing lines they belong to. As the data for the twin nodes is stored directly behind that of the new node and as we know how many twin nodes each dividing line node has got (because the number of twin nodes is equal to the number of subdomains the node couples to, which is stored in *dlote*) this can be done very easily in the same way as for the new node itself. Now we have stored in *sect* in which subdomains the twin nodes occur and we copy this array to the columns 4 to $n_{sect} + 3$ into all the rows in *tnod* we just created,

see figure 4.4.13 for a cross of eight dividing surfaces (DS) in 3-D. Seen geometrically, we have two crossing dividing surfaces that have been transformed to eight logical dividing surfaces when we created the new logical dividing surface nodes, see subsection 3.5.1. In 2-D it is not possible that a new dividing line node couples to more than one subdomain. The code for the entering of the data for the twin nodes of the first dividing line node into the array *tnod* is shown in listing 15.



i	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
	$tnod(nib_n+i,k)$						
	node	DS	sub-domain	twin node 1	twin node 2	twin node 3	twin node 4
1	1001	1	4	1002	1003	1004	1001
2	1001	8	4	1002	1003	1004	1001
3	1002	3	1	1002	1003	1004	1001
4	1002	5	1	1002	1003	1004	1001
5	1003	2	2	1002	1003	1004	1001
6	1003	7	2	1002	1003	1004	1001
7	1004	4	3	1002	1003	1004	1001
8	1004	6	3	1002	1003	1004	1001

Figure 4.4.13: Illustration of *tnod* for new dividing line nodes, 3-D crossing of 8 dividing surfaces (DS).

The example for the crossing of eight dividing surfaces in figure 4.4.13 should illustrate this proceeding. The node that results from the refinement of an element gets the first number, here 1001. The other nodes in the subdomains 1, 2 and 3 result from the “matching grids”-condition. Each of the 4 nodes belongs to 2 dividing surfaces, so we need $2 \cdot 4 = 8$ rows

in the array *tnod* for the dividing surface information. Node 1001 is in subdomain 4 and belongs to dividing surface 1 and dividing surface 8, node 1002 is in subdomain 1 and belongs to the dividing surfaces 3 and 5, and so on. At the end, the auxiliary array *sect* therefore looks like this:

$$\begin{aligned} sect(1) &= 1002, \\ sect(2) &= 1003, \\ sect(3) &= 1004, \\ sect(4) &= 1001. \end{aligned} \tag{4.4.34}$$

This array is copied to the columns 4 to $n_{sect} + 3$ of array *tnod* for each of the 8 rows we just created.

SUMMARY: When we refine dividing line elements we have to check which of the new elements of the higher refinement stage are dividing line elements. These are all elements that have got at least two dividing line nodes that belong to the same dividing line. So we determine those edges of the refinement elements that are dividing line edges first and then we are able to determine the new dividing line edges and therefore the new dividing line elements. In 3-D we also must regard combinations of two old edges to get all new dividing line edges, because not only the bisectioned parts of the old dividing line edges may become dividing line edges but also those edges that are the connection of two newly-created nodes, because in 3-D we have dividing surfaces in exact terminology. If both end points of a newly-created edge belong to the same dividing surface, the edge becomes a dividing line edge.

The generation of the new elements is done the same way as without dividing lines. In the element information array *nenr* (see on page 78) we set

$$\begin{aligned} nenr(ne+i,1) &= ne+i \\ nenr(ne+i,2) &= nenr(el_{old},2) \\ nenr(ne+i,3) &= \begin{cases} 0 & \text{for non-dividing line elements} \\ 1 & \text{for dividing line elements} \end{cases} \\ nenr(ne+i,4) &= 1 \end{aligned} \tag{4.4.35}$$

for $i = 1, \ldots, 3 \cdot ne_{re}$. In order to be able to fill in the third column we have to check which of the generated elements are dividing line elements, i.e. which of the generated elements have at least one edge on a dividing line. In a preparing step we check for each edge in each refinement element if both end points are dividing line nodes. If yes, we enter a *true* into a **logical array *logarr*** with length equal to the number ne_{re} of refinement elements in the current refinement step rs and width equal to n_{edge} (this array has been initialized with *false* before). After checking all elements we go through the element array again. This time we

assign the property dividing line element to the new elements. In 2-D this is quite easy: we check if edge k in refinement element el is a dividing line edge, i.e. we

$$\text{check if } \ logarr(el, k) = true$$

and if yes, the two new elements $el_{new,1}$ and $el_{new,2}$ that share this edge get the property "dividing line element", i.e. we set

$$\begin{aligned} nenr(el_{new,1}, 3) &= 1 \\ nenr(el_{new,2}, 3) &= 1. \end{aligned} \tag{4.4.36}$$

So in contrast to (4.4.14) we store in the third column of *nenr* the value 1 for dividing line elements.

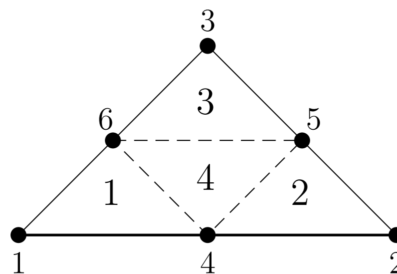


Figure 4.4.14: Illustration of the dividing line property in 2-D, edge 1 (bold) is dividing line edge.

In table 4.4.7 you can see which elements get the dividing line element property if a certain edge of the old element is a dividing line edge, in figure 4.4.14 we illustrate this for edge 1, the connection of nodes 1 and 2, of the old element having the dividing line property. We see clearly that the new elements 1 and 2 become dividing line elements as they both get one half of the bisectioned edge 1. From table 4.4.7 we learn that, if edge 2 was the dividing line edge, the elements 1 and 3 became dividing line elements, and for edge 3 being the dividing line edge, the elements 2 and 3 became dividing line elements.

DL	end point		new DL	
edge	1	2	elements	
1	1	2	1	2
2	2	3	2	3
3	1	3	1	3

Table 4.4.7: dividing line property for the new elements in 2-D.

In 3-D this algorithm is more complex because there are not only two elements that may become dividing line elements if an edge is a dividing line edge. If an edge is a dividing line edge this means that the newly-created mid-point of this edge is a dividing line node, too. So if there is a second edge in the same element that is also a dividing line edge all elements that share the edge between the two newly-created dividing line nodes on the two edges become dividing line elements. Like in 2-D an element is a dividing line element in 3-D if at least one of its edges belongs to a dividing surface (we say “line”). This means that there are at least two new dividing line elements if an edge is a dividing line edge, but there may be more (this depends on the other edges), see table 4.4.8 in connection with figure 4.4.8, whereas in 2-D we can determine the new dividing line elements without paying attention to other edges of the same refinement element. For example, if in table 4.4.8 edge 1 is a dividing line edge (first row), nodes 1 and 2 and with them the new node 5 that is the mid-point of the edge between them (see figure 4.4.8) are dividing line nodes. This means that elements 1 and 2 are dividing line elements because they contain the edge 1–5 and 2–5, respectively, see table 4.4.6. If the other edges of the element are not any dividing line edges, these are all new dividing line elements. If, for example, edge 2 is also a dividing line edge (second row of table 4.4.8), node 3 and—as it is the mid-point of the dividing line edge 2–3 (see figure 4.4.8)—the new node 6 is a dividing line node and therefore the edge between the dividing line nodes 5 and 6 is a dividing line edge. So all elements that share this edge are dividing line elements, too. These elements have the numbers 5 and 6 (see table 4.4.6, element 1 has already become a dividing line element). Element 3 is a dividing line element because nodes 3 and 6 are dividing line nodes and therefore the edge between them becomes a dividing line edge. In table 4.4.8 all multiple entries are omitted.

The algorithm of the mesh refinement on a single processor for problems with dividing lines can be seen on page 122, algorithm C.

For the mesh refinement with sliding dividing lines we do not have to worry about anything special. Of course, we must determine the new sliding dividing line nodes, but this is done almost exactly like for the external boundary nodes. We first determine the maximum number sd_{max} of sliding dividing lines a sliding dividing line node belongs to and allocate an **integer array *sdnr*** with length equal to the number of sliding dividing line nodes and width equal to $sd_{max}+1$. We sort the **integer array *snod*** with the information of the sliding dividing line nodes by ascending node number (by the external procedure HEAPSORTIX, see subsection 4.1.3) and store the sorted node numbers in the first column of *sdnr*. Analogously to *tnod* for the dividing lines, we store in the columns of *snod* the number of the sliding dividing line node, the numbers of the sliding dividing line and the subdomain the node belongs to, and in the columns 4 to $n_{sect}+3$ we store the numbers of the coupling nodes for the n_{sect} subdomains. In the next columns of *sdnr* we store the sliding dividing line numbers of the sliding dividing lines to which each node belongs. We determine the new sliding dividing line nodes exactly like for the external boundary nodes as described

DL edge	end point 1	end point 2	2nd DL edge	new DL elements		
1	1	2	–	1	2	
			2	3	5	6
			3	4	6	7
			4	3	5	8
			5	4	7	8
2	1	3	–	1	3	
			3	4	6	
			4	2	5	
			6	4	5	6
3	1	4	–	1	4	
			5	2	7	
			6	3	6	7
4	2	3	–	2	3	
			5	4	8	
			6	4	5	8
5	2	4	–	2	4	
			6	3	7	8
6	3	4	–	3	4	

Table 4.4.8: Illustration of the dividing line property in 3-D.

on page 97. The number of the new sliding dividing line node is stored together with the sliding dividing line number in an **integer array *newsn*** with length equal to r_{cnt} and width equal to 3.

The main difference between new external boundary nodes and new sliding dividing line nodes results from the fact that we need a fictitious opposite node for sliding dividing line nodes so that we have to assign such a node to each new sliding dividing line node. We could assign any node of the coupling sliding dividing line but in order to reduce the time for the search for an appropriate fictitious opposite node in the next computation cycle we try to assign a node with a distance as small as possible. And what could be a better fictitious opposite node than that ones of the end points of the edge of which the investigated new sliding dividing line node is the mid-point? We choose the first of the two end points nod_1 and store the number nod_1 in the third column of the array *newsn*. We use the same code for external boundary nodes and sliding dividing line nodes, the only difference is that the line

```
newbn(nblc,3) = nod_1
```

in listing 12 and 13 is only used for sliding dividing line nodes.

After having stored the information for all new sliding dividing line nodes in *newsn* we must insert the sliding dividing line node information into the *snod* array. For each new sliding dividing line node we copy the new node number and the sliding dividing line number from *newsn* to the end of *snod*. The subdomain number of the new node is the same as that of the end points of the edge. We have to determine the row in *snod* where the data for the chosen end point, i.e. the first end point, of the refinement edge is stored. As the node numbers in *snod* are only stored in ascending order for each sliding dividing line but not for the whole array we execute a binary search on the concerned sliding dividing line and copy the fictitious opposite nodes—one for each subdomain the node couples to—of the end point, that are stored in the columns 4 to $n_{sect} + 3$ in *snod*, into the row of the new sliding dividing line node in *snod*.

nek:	integer array for node numbers of the elements
nekinv:	integer array for elements a node belongs to
nnr:	integer array for node information
rtl:	integer array for refinement edge information

C1 Create the *nekinv* array by inverting the element array *nek*.

C2 Search for the neighbour elements of the same refinement stage for each edge of the refinement elements.

C3 Insert the necessary edge information into the *rtl* array.

C4 Update the *nnr* array with the data of the new nodes.

C5 Only for coupled domains with dividing lines: Check if the new nodes are dividing line nodes.

C6 Assign coordinates and order to the new nodes.

C7 Interpolate the solution for the new nodes.

C8 Insert the new nodes into the refinement elements and their neighbour elements in the *nek* array.

C9 Check if the new nodes are external boundary nodes.

C10 Only for coupled domains with sliding dividing lines: Check if the new nodes are sliding dividing line nodes.

C11 Generate four elements in 2-D and eight elements in 3-D out of each old refinement element.

C12 Prepare the next refinement step.

Algorithm C: Algorithm for one refinement step of the mesh refinement on a single processor.

Before we insert the new dividing line nodes into *tnod* and the new sliding dividing line nodes into *snod*, the arrays for the dividing line node and the sliding dividing line node information, we have to check if we exceeded the limit for the boundary nodes nb_{max}. We use the same value for all three kinds of boundary nodes, so the check is he same as for the external boundary nodes as described on page 97.

The algorithm of the mesh refinement on a single processor for problems with sliding dividing lines is algorithm C. This algorithm is valid for 2-D and 3-D for problems with or without dividing lines or sliding dividing lines for the mesh refinement on a single processor.

4.5 Mesh refinement on a distributed memory parallel computer

Again we first explain the 2-D algorithm of the mesh refinement for a distributed memory parallel computer, continue with the differences to 3-D and conclude with the peculiarities of dividing lines and sliding dividing lines.

4.5.1 Mesh refinement in 2-D

Important notations:

bnod: integer array for external boundary nodes
iglob: integer array for starting addresses of own ref. edges in *rtl*
infarr: integer array for sending update information to overlap processors
lglob: logical array for own refinement edges
lnpl: logical array for edge identification
lp: integer array for numbers of edges to receive from overlap processors
nbrs: integer array for numbers of nodes, elements and boundary/DL/SDL nodes
nek: integer array for node numbers of elements
nekinv: integer array for elements a node belongs to
nenr: integer array for element information
nenrs: integer array for corresponding local number for a global element number
nnr: integer array for node information
nnrs: integer array for corresponding local number for a global node number
p_{cnt}: integer array, counter for edges in *ptl*
ptl: integer array for refinement edge information (overlap edges)
rcvl: integer array for updating element information in left overlap
rcvlct: integer array for number and starting address of the elements in *rcvl* per proc.
rcvr: integer array for updating element information in right overlap
rcvrct: integer array for number and starting address of the elements in *rcvr* per proc.
refst: integer array for refinement stages of elements
rtl: integer array for refinement edge information (local edges)

The refinement process on a distributed memory parallel computer begins exactly like described in subsection 4.4.1 for a single processor. We start with the largest elements of stage 0 and deal with one stage after the other up to the smallest refinement elements of stage rs_{max}. The refinement for one stage is called a refinement step.

Again we first invert the array for the node numbers of the elements, i.e. the *nek* array, to get the inverted *nek* array *nekinv* where we store the element numbers a node belongs to, then we search for the neighbour elements of each refinement element. Note tht both the *nek* array and the *nekinv* array comprise the nodes and elements, respectively, that are owned by a processor and those belonging to the left and right overlap. Afterwards we have to set up the *rtl* array with the information about the new nodes that have to be generated. Here the first difference to the single processor version comes to light.

Rule 5
An edge is always owned by that processor its leftmost node is owned by.

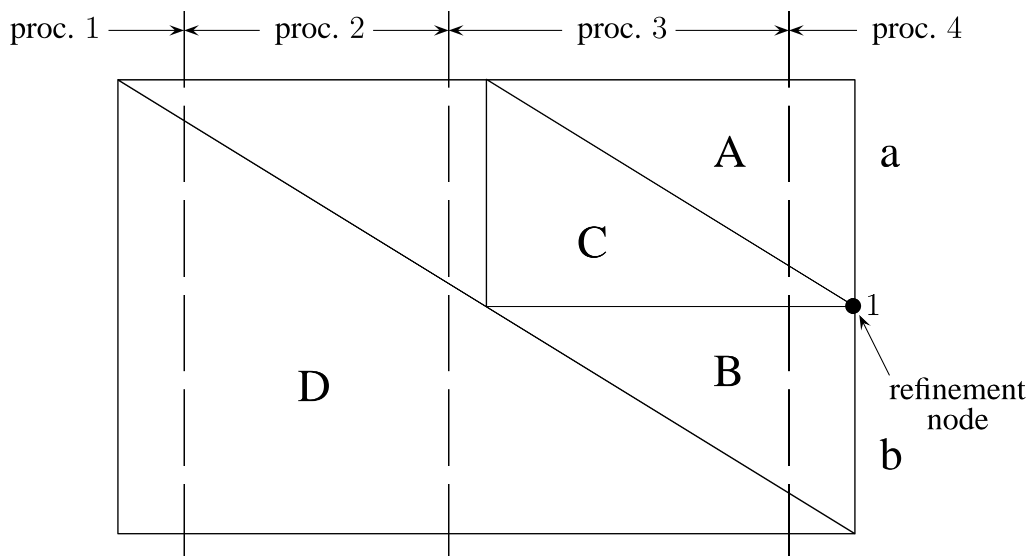


Figure 4.5.1: Illustration of the refinement process on $np = 4$ processors.

In figure 4.5.1 you see a similar situation as in section 4.3. There are edges that are not owned by the processor that owns the refinement element in figure 4.3.6 as well as in figure 4.5.1 (see rule 5). So during the refinement cascade process the information about the

refinement of an element must be sent to the left, and afterwards, during the refinement process itself, the information about an edge that has to be halved must be sent to the right to the processor that owns the edge and therefore has to generate the new mid-point (see rule 6). In figure 4.5.1 the triangles A, B and C are refinement elements because node 1 is a refinement node. As these triangles are owned by processor 3, the information about the refinement must be sent from processor 4 to processor 3 (and because of the refinement cascade processor 3 must send the information that triangle D is also a refinement element to processor 1, but this is of no account here). When the elements are really refined by halving the edges processor 3 that refines these elements must send the refinement edge information (see table 4.5.1) for the edges a and b to the right to processor 4 because this processor is the owner of the two edges, see rule 5.

Rule 6
An edge is always halved by that processor that owns the edge, i.e. this processor generates the mid-point and the new node number.

SUMMARY: A refinement element may have edges that belong to the right overlap, i.e. they are owned by an overlap processor and must therefore be halved by this processor. So we have to introduce a second array besides *rtl* to store the data for the refinement edges of the overlap. This array has got one column for each overlap processor on the right side, and after having inserted all refinement edge information the data of each column is sent to its corresponding overlap processor by the use of the communication pattern we introduced in subsection 4.1.4, i.e. first the processors only get knowledge of the message lengths they will receive in each communication cycle and in a second step the real edge data is passed to the target processors.

So for the mesh refinement on distributed memory parallel computers we only store the information of the own refinement edges in array *rtl*. When we create the *rtl* array, we must simultaneously create a second array for the edges that belong to the right overlap because an edge is always halved by the processor it belongs to, see rule 6. This means that the edge information has to be sent to the corresponding overlap processor and therefore we store the edge data in a second array. This **integer array** is ***ptl***, its length is equal to $nbm_6 \cdot notp$ and its width is equal to the number of overlap processors on the right side np_{sr} where $notp$ is the number of edges of own elements in the right overlap. This number has been computed during the processing of the refinement cascade. Moreover, it holds

$$nbm_6 = nnb_{max} + 6 = 7 \quad \text{in 2-D} \tag{4.5.1}$$

as an element may have at most one neighbour element at an edge ($nnb_{max} = 1$), so that we have the edge information shown in table 4.4.1 for *rtl* and in table 4.5.1 for *ptl* for each

refinement edge. Note that an overlap edge can never be in the left overlap as the corresponding refinement element would be owned by the same processor because of rule 4.

Now we go through the array *indrel* with the refinement element numbers of the current refinement stage and check for each edge of the current element that has not yet a mid-point if it is owned by the processor itself, which means—because of rule 5—if at least one of its end points is owned by the processor (see next but one paragraph). If yes, the information about the refinement edge is inserted into *rtl*, the array for the local refinement edges, otherwise we first determine the processor by which the edge is owned. This is, according to rule 5, the leftmost of the processors the end points belong to. On principle, then the edge information is inserted into *ptl*, the array for the overlap refinement edges, like it is shown in table 4.5.1. In contrast to array *rtl* (see table 4.4.1) we have to insert the global node and element numbers into *ptl* because the communication is only possible by global node and element numbers.

i	$ptl(i,k)$	description
⋮	⋮	⋮
{ preceding edge }		
$p_{cnt}(k)+1$	left free	
$p_{cnt}(k)+2$	nod_1	global number of end point 1 of edge j
$p_{cnt}(k)+3$	nod_2	global number of end point 2 of edge j
$p_{cnt}(k)+4$	el	global refinement element number
$p_{cnt}(k)+5$	pos	local node number of new node in element (see figure 4.4.3)
$p_{cnt}(k)+6$	1	number of neighbour elements ($=1$ in 2-D)
$p_{cnt}(k)+7$	el_{nb}	global number of the neighbour element
{ following edge }		
⋮	⋮	⋮

Table 4.5.1: Illustration of the contents of column k of array *ptl* in 2-D, k is the column to be sent to processor $ip+k$.

Each column has an own pointer to the current end of the column. These counters are stored in a one-dimensional **integer array p_{cnt}** with length equal to np_{sr}. So after having determined the processor ip_{own} the edge belongs to we compute the corresponding column col_{ptl} in *ptl* by simply subtracting the own processor number ip:

$$col_{ptl} = ip_{own} - ip \tag{4.5.2}$$

because we want to store the refinement edges owned by the direct neighbour processor $ip+1$ in column 1 of *ptl*, those owned by overlap processor $ip+2$ in column 2 etc. like

described for the array *sndrto* used during the refinement cascade.

If we are in a cycle where the elements have different refinement stages, the determination if an edge is in the right overlap is quite difficult. This is because we do not change the order of the nodes on a processor during the refinement process (we will see later that this statement is not quite right). At the beginning we have n_l own nodes on each processor followed by the nodes of the overlap (first the left overlap, then the right one), altogether n_n nodes. In the first refinement step we generate $n_{overl,1}$ new overlap nodes that are the mid-points of the refinement edges that are owned by overlap processors on the right side and that begin with the local number n_n+1. They are followed by the $n_{own,1}$ newly-created own nodes resulting from the bisection of the own refinement edges. At the end of a refinement step we have some overlap nodes again. After the last overlap node of refinement step 1 follow the first overlap nodes of refinement step 2 and so on. To know which nodes are own nodes we introduce an **integer array *nbrs*** with length equal to $3 \cdot (rs_{max} + 2)$ and width equal to 5 where

$$rs_{max} = n_{cyc} - 1 \tag{4.5.3}$$

as already mentioned on page 74. The length comes from the fact that we need 3 entries per refinement stage (and we have $rs_{max} + 1$ refinement stages as the largest elements have refinement stage 0) and 3 additional entries for the old nodes and elements that were stored before the refinement process on a processor. We need a width of 5 because we store in column 1 the numbers of nodes and we also use *nbrs* for the numbers of elements (column 2) and the numbers of external, internal and sliding boundary nodes (columns 3 to 5). At the beginning of the refinement process we insert into the first entry the number n_l of own nodes, into the second entry the number of the first node of the right overlap and into the third entry the number of the last node of the right overlap. At the end of each refinement step the information for the corresponding stage is inserted into *nbrs*. In refinement step rs we insert in the first column of *nbrs* into entry $3 \cdot rs + 1$ the number of new own nodes, into entry $3 \cdot rs + 2$ the local number of the first new overlap node on the right side and into entry $3 \cdot (rs + 1)$ the local number of the last new overlap node, see table 4.5.2. If there are not any refinement elements in the current refinement step we copy the values of the last refinement step into the appropriate entries in *nbrs*. The same we do for the numbers of elements that are entered into column 2. For the external boundary nodes, the dividing line nd sliding dividing line nodes it is of no importance if the overlap nodes are in the left or in the right overlap, so we only store the number of new own nodes and the local number of the last overlap node in the columns 3 to 5 of *nbrs*.

In the counter array p_{cnt} we want to store the number of refinement edges for which we have to send the edge information to the corresponding overlap processors on the right side. At the moment, we have stored the number of entries in the columns of *ptl*, so we have to divide all entries in p_{cnt} by nbm_6 after having created *ptl* and p_{cnt}, as one edge consists of

```
      ! for each refinement stage
      do l = 1,rs+1
        ! check if end points are in the right overlap
        if ((nod_1 >= nbrs(3*l-1,1)).and.
     &      (nod_1 <= nbrs(3*l,1)).and.
     &      (nod_2 >= nbrs(3*l-1,1)).and.
     &      (nod_2 <= nbrs(3*l,1))) then

!***      if both points belong to overlap then the
!***      new point also belongs to overlap proc.
!***      therefore insert edge information into ptl
          ! p_nod_1/p_nod_2: owning proc. of nod_1/nod_2
          p_nod_1 = nnr(nod_1,4)
          p_nod_2 = nnr(nod_2,4)

          ! proc: proc. new node is generated (relative to myproc)
          proc = min0(p_nod_1,p_nod_2)-myproc
          ! addr: starting address of edge in ptl
          addr = p_cnt(proc)

          ! insert end points, elem.nr. and pos. in elem. into ptl
          ptl(addr+2,proc) = nnr(nod_1,1)
          ptl(addr+3,proc) = nnr(nod_2,1)
          ptl(addr+4,proc) = nenr(indrel(j),1)
          ptl(addr+5,proc) = k

          ! insert neighbour elements into ptl
          ptlcnt = 0
          do ii = nbfst,nblst-1
            ptlcnt = ptlcnt+1
            ptl(addr+6+ptlcnt,proc) =
     &        nenr(nb(curr,nbfst+ptlcnt-1),1)
          end do
          ! insert nr. of neighbour elements into ptl
          ptl(addr+6,proc) = ptlcnt
          ! increase counter for next edge
          p_cnt(proc) = p_cnt(proc)+nbm_6

          ! found = true means new node is in overlap
          found = .true.
          ! no search in next ref. stages needed anymore
          exit

        end if
      end do
```

Listing 16: Code for inserting the edge information into *ptl*.

Ref. step	Entry k	Nodes	Elements	Boundary nodes ext.	Boundary nodes int.	Boundary nodes slid.
	1	number of own nodes/elements before refinement				
	2	local number of first node/element of right overlap before refinement		—	—	—
	3	local number of last node/element of overlap before refinement				
1	4	number of new own nodes/elements in ref. step 1				
	5	local number of first new node/element in right overlap in ref. step 1		—	—	—
	6	local number of last new node/element of overlap in ref. step 1				
⋮	⋮	⋮				
rs	$3rs{+}1$	number of new own nodes/elements in ref. step rs				
	$3rs{+}2$	local number of first new node/element in right overlap in ref. step rs		—	—	—
	$3rs{+}3$	local number of last new node/element of overlap in ref. step rs				

Table 4.5.2: Illustration of the array *nbrs*.

nbm_6 entries:

$$p_{cnt}(ip) \Leftarrow \frac{p_{cnt}(ip)}{nbm_6} \quad \text{for} \quad ip = 1, \ldots, np_{sr}. \tag{4.5.4}$$

This division is the analogue to (4.4.5) in 2-D or, more general, (4.4.24) in 3-D, respectively, for r_{cnt}, the counter for the array *rtl*.

On a distributed memory parallel computer we also have to check if we exceeded the limit for the nodes by the generation of the new nodes. As we have remarked for the maximum number of elements in section 4.3 similarly, the maximum number of nodes n_{max} is a local length on a distributed memory parallel computer. So we have to

$$\text{check if } \; n_n + \sum_{ip=1}^{np_{sr}} p_{cnt}(ip) + \frac{r_{cnt}}{nbm_6} > n_{max}$$

and if yes, we print out an error message and stop the computation.

As the last nodes stored on a processor before the refinement process are overlap nodes, we want to create the new overlap nodes first in order to have all overlap nodes one after the other. So each processor starts with the generation of the new nodes that are created

by halving those edges that are in the overlap of a processor. Here it is of no account if an edge must also be halved because an element on the processor that owns this edge has to be refined. We use the communication pattern we described in subsection 4.1.4 on page 55 and first only send the number of edges to the right so that it is known on each processor how much information it will receive and from which processors it will receive the data. Again, we omit sending messages with length equal to zero. At the beginning we copy the array p_{cnt} to a buffer *sndbuf* which is sent around in $np-1$ cycles. In each communication cycle i the processors

$$\text{from} \quad ip = i$$
$$\text{to} \quad ip = np - 1$$

send their current buffer *sndbuf* to their right neighbour processor. The processors

$$\text{from} \quad ip = i + 1$$
$$\text{to} \quad ip = np$$

receive the message in a buffer *rcvbuf* and store the relevant information in a one-dimensional **integer array *lp*** with length equal to np_{max}:

$$lp(i) = rcvbuf(i).$$

Afterwards we set

$$sndbuf = rcvbuf$$

because we want to send the p_{cnt} array we just received from processor $ip-1$ in *rcvbuf* to processor $ip+1$ in the following communication cycle with which we continue immediately. In table 4.5.3 the relation between p_{cnt} and *lp* is shown clearly for $np = 8$ processors. In the rows we have the processor numbers and in each column of p_{cnt} you can see the number of edges to send to the ith overlap processor, i.e. $i = 1$ is the direct neighbour processor on the right side, $i = 2$ is the second overlap processor and so on. Here i also indicates the number of the communication cycle in which the data will be sent. For *lp* the ith column represents the ith overlap processor on the left side from which a processor will receive data. For example, processor 1 will send the information for 10 edges to its second neighbour processor on the right side (see left table, row 1, column 2)—which is processor 3—in the second cycle, so it holds $lp(2) = 10$ on this processor (see right table, row 3, column 2). On processor 3 it is known that it will receive 10 edges from processor 1 because the 10 is stored in the second entry of *lp*. In general, it holds

$$lp_{ip+i}(i) = p_{cnt,ip}(i) \tag{4.5.5}$$

where ip is the number of the sending and $ip+i$ the number of the receiving processor, and i is the number of the communication cycle.

proc.	$p_{cnt}(i)$ $i=1$	$i=2$	$i=3$
1	20	10	2
2	18	12	0
3	19	0	0
⋮	⋮	⋮	⋮
7	17	0	0
8	0	0	0

proc.	$lp(i)$ $i=1$	$i=2$	$i=3$
1	0	0	0
2	20	0	0
3	18	10	0
4	19	12	2
⋮	⋮	⋮	⋮
8	17	0	0

Table 4.5.3: Illustration of the arrays p_{cnt} and *lp* for $np = 8$ processors.

By the array *lp* the number of edges for which processor ip will receive data is known on this processor and by multiplying this length with mem_{int}, the number of bytes for an integer variable, we easily get the message lengths. If $lp(i) = 0$ holds for an overlap processor there is no message to receive from this processor and we therefore continue with the communication to the next overlap processor.

Now we can send the real edge information that is stored in the array *ptl* because on each processor the lengths of the messages to receive are known. This second part of the communication consists of $np_{max,r}$ cycles. In each communication cycle i processor ip sends the data in the column of *ptl* for processor $ip + i$ to this processor if

$$ip \leq np - i \quad \wedge \quad p_{cnt,ip}(i) > 0 \tag{4.5.6}$$

holds, i.e. all processors from processor 1 to that processor that sends to the last processor np pass their data to processor $ip + i$ if there is any data to send at all. The message has a length of $nbm_6 \cdot p_{cnt,ip}(i) \cdot mem_{int}$ bytes. All processors with

$$ip > i \quad \wedge \quad lp_{ip}(i) > 0 \tag{4.5.7}$$

are ready to receive a message from processor $ip - i$. The message is received in the *ptl* array of the receiving processor $ip + i$, the starting address is $nbm_6 \cdot p_{cnt,ip+i}(i) + 1$, i.e. the received data is stored directly behind its own edge information in column i that the processor just sent to processor $ip + i$. In figure 4.5.2 we illustrate the array *ptl* on processor ip after it sent and received the refinement edge information in array *ptl*.

SUMMARY: The refinement edge information of those edges that belong to the overlap of the processors that own the corresponding refinement elements has been passed to that processors that are authorized to create the mid-points of these edges. For the present we only issue the global node numbers of the new nodes. As it must be known how many nodes the other processors have to create in order to be able to compute the

i	$p_{cnt}(i)$
1	7
2	5
3	3

i	$lp(i)$
1	5
2	3
3	2

i	$ptl(i,k)$ $k=1$	$k=2$	$k=3$
1	ref. edges sent to $ip+1$	ref. edges sent to $ip+2$	ref. edges sent to $ip+3$
$\vdots$			
$3 \cdot nbm_6$			
$\vdots$			ref. edges rec. from $ip-3$
$5 \cdot nbm_6$			
$\vdots$		ref. edges rec. from $ip-2$	0
$7 \cdot nbm_6$			
$\vdots$	ref. edges rec. from $ip-1$		
$8 \cdot nbm_6$			
$\vdots$		0	$\vdots$
$12 \cdot nbm_6$			
$12 \cdot nbm_6 + 1$	0	$\vdots$	
$\vdots$	$\vdots$		
$notp \cdot nbm_6$	0	0	0

Figure 4.5.2: Illustration of array *ptl* on processor ip after sending to the right for $np_{sr} = 3$.

own new global node numbers this needs communication again. After having finished this part the edge data we just completed by the new global node numbers is sent back to the source processors so that there the new node numbers are also known.

We go through the edge information arrays a processor received and first issue new local node numbers from 1 to n_{cnt} as we do not yet know the global numbers the new nodes will get at this time. Afterwards the receiving processors must issue the new global node numbers for the new nodes, i.e. the new global node numbers are determined by the processor that owns the refinement edge. The problem is that we only know the number of new nodes to be generated on each processor, but the new nodes are still numbered from 1 to n_{cnt}. If we want to give the new global node numbers, we must know the starting address of the global numbers of the new nodes on each processor first, i.e. it must be known on each processor how many new nodes the processors on the left side will generate. This needs communication again because each processor must send its local n_{cnt} to the right. Again, this is done by the communication pattern described on page 55 but this time it is only a scalar that is sent around in $np - 1$ cycles. In each cycle the receiving processor ip adds the received value to an integer npp_{ip} that has been initialized by 0. At the end npp_{ip} is the number of nodes that is generated by the processors on the left side of a processor ip. In

figure 4.5.3 the numbering of the new nodes is illustrated.

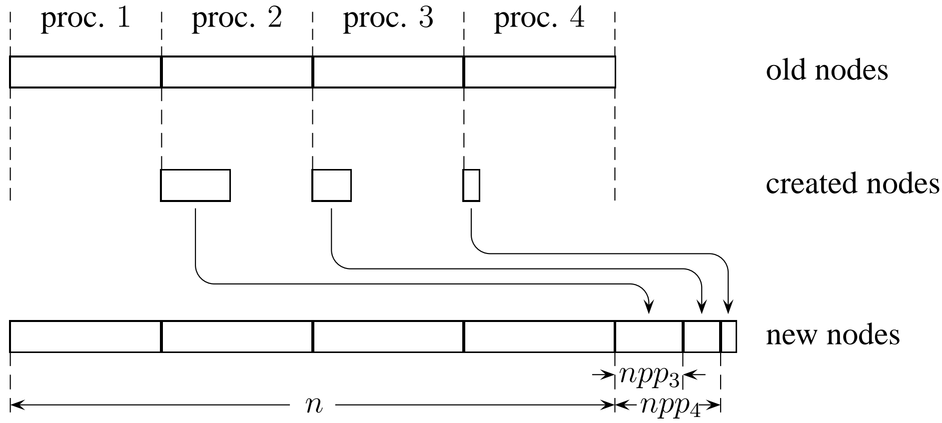


Figure 4.5.3: Illustration for global numbering of the new nodes originating from the *ptl* array for $np = 4$ processors.

Then the new global node numbers are issued. The new nodes on processor ip get the global numbers

$$\begin{aligned} \text{from} \quad & n + npp_{ip} + 1 \\ \text{to} \quad & n + npp_{ip} + n_{cnt}. \end{aligned}$$

As the new nodes already have got numbers from 1 to n_{cnt} we can very easily compute the new global node numbers by adding $n + npp_{ip}$. The total number of new nodes is n_{ptl} and is computed on the last processor np by

$$n_{ptl} = npp_{np} + n_{cnt,np}. \tag{4.5.8}$$

Afterwards this number is scattered onto the other processors.

Now the completed *ptl* array with the overlap refinement edge information and the new node numbers is sent back to the processors it originally came from. This is done in $np_{max,r}$ cycles where processor ip sends column i of *ptl* (from starting address $nbm_6 \cdot p_{cnt,ip}(i) + 1$) back to processor $ip - i$ in each cycle i if

$$ip > i \quad \wedge \quad lp_{ip}(i) > 0 \tag{4.5.9}$$

holds. All processors with

$$ip \leq np - i \quad \wedge \quad p_{cnt,ip}(i) > 0 \tag{4.5.10}$$

get ready to receive the message from processor $ip + i$. Here we know the message length on the receiving processors as they have sent the messages with the same length to the right

before. The received message is directly written over the old *ptl* array.

SUMMARY: After the processors received "their" *ptl* array from the overlap processors they append the edge information of the *ptl* array to their local *rtl* array and issue the local node numbers for the new nodes.

Although the nodes in the *ptl* array are overlap nodes, we must also assign coordinates, consistency order and function values to them and they must be inserted into the refinement elements and their neighbour elements. We want to have only one array for all edges with this characteristic and we choose for reasons of simplicity the array for the local refinement edges, i.e. the *rtl* array, so the edge information of all columns of the *ptl* array must be appended to the *rtl* array, i.e. after the local edge information follows the information for the edges of the first right neighbour processor, then the information for the edges of the second right neighbour processor and so on. As we have local node and element numbers in *rtl* we have to transform the global node and element numbers in *ptl* into local ones simultaneously.

We also insert the new global node number of a node and the processor that owns the node into the node information array *nnr*, see table 4.4.2. Therefore we must also issue new local node numbers and connect them with the global ones. As the last old node has got the number n_n and we continue counting the new local nodes by the following natural numbers as usual, the new local node numbers go

$$\begin{aligned} \text{from} \quad & n_n + 1 \\ \text{to} \quad & n_n + \sum_{i=1}^{np_{max,r}} p_{cnt}(i). \end{aligned}$$

Then we are able to store the node numbers in the *nnr* array:

$$\begin{aligned} nnr\left(n_n + \sum_{k=1}^{j-1} p_{cnt}(k) + i, 1\right) &= ptl\,(nbm_6 \cdot (i-1) + 1, j) \\ nnr\left(n_n + \sum_{i=1}^{j-1} p_{cnt}(k) + i, 4\right) &= ip + j \end{aligned} \tag{4.5.11}$$

for $i = 1, \ldots, p_{cnt}(j)$ and $j = 1, \ldots, np_{max,r}$. So we first insert the new nodes generated by the first overlap processor on the right side into *nnr*, followed by the new nodes of the second overlap processor and so on. In *ptl* we have stored the global node numbers of the new nodes that have been sent back from processor $ip+j$. These global numbers are stored in the first column of *nnr*. In the fourth column we store the number of the processor that owns the new node. Columns 2 and 3 of *nnr* are left free as the nodes are overlap nodes that will be deleted at the end of the refinement process and therefore the information is

only stored on the processor that owns the node.

We set

$$saddr_2 = \sum_{i=1}^{np_{max,r}} p_{cnt}(i) \qquad (4.5.12)$$

in order to have the starting address for the local numbers of the next nodes. When we append the data of a refinement edge to the *rtl* array we have to increase the integer counter r_{cnt} by nbm_6 so that

$$r_{cnt} \Leftarrow r_{cnt} + nbm_6 \cdot \sum_{i=1}^{np_{max,r}} p_{cnt}(i) \qquad (4.5.13)$$

holds after having appended the edge information of each edge in each column of array *ptl*. The code for this process is shown in listing 17 on page 136, the algorithm for the issuing of the new node numbers for the mid-points of the edges in the *ptl* array is algorithm D.

ptl: integer array for edges to be refined by overlap processors

D1 Pass number of edges in each column of *ptl* to the right.

D2 Pass the columns of *ptl* to the corresponding processors.

D3 Receiving processors:

- **a)** Go through all received edge arrays.
- **b)** Issue a new local node number to the mid-point of each edge.
- **c)** Pass the number of new nodes to the right.
- **d)** Issue the global numbers to the new nodes.
- **e)** Pass the received and completed edge arrays back to the original processors.

Algorithm D: Algorithm for the issuing of new node numbers to the mid-points of the edges in the *ptl* array (2-D).

SUMMARY: After having inserted the edge information of the edges in the right overlap into *rtl* we do the same for the edge information the processors just received from the left in the *ptl* array. As it is possible that there occur the same edges in the received *ptl* array and the local *rtl* array we must take care to append only new edges to *rtl*. Afterwards the processors issue the new local node numbers for the mid-points of the appended edges. Then we assign local and global node numbers to all new nodes that did not get one yet. These are the mid-points of those edges that have to be halved only by their owning processor.

```
      ! saddr2: local nr. of new nodes already generated
      saddr2 = 0

      ! for each right overlap processor
      do i = 1,np_maxr
        ! for each new node of the current overlap proc.
        do j = 1,p_cnt(i)
          ! saddr1: starting address of edge in ptl
          saddr1 = nbm_6*(j-1)

          ! nod_1/nod_2: local numbers of end points of the edge
          nod_1 = BINSCH(nnrs(1,1),nnrs(1,2),nnold,ptl(saddr1+2,i))
          nod_2 = BINSCH(nnrs(1,1),nnrs(1,2),nnold,ptl(saddr1+3,i))

          ! nb_1: neighbour number of nod_2 for nod_1
          nb_1 = NBSRCH(iinfo,nod_1,nod_2,ilev,idstar)
          ! nb_2: neighbour number of nod_1 for nod_2
          nb_2 = NBSRCH(iinfo,nod_2,nod_1,ilev,idstar)

          ! set lnpl = true for this pair of end points
          lnpl(nod_1,nb_1) = .true.
          lnpl(nod_2,nb_2) = .true.

          ! insert new local node info into nnr
          nnr(n_n+saddr2+j,1) = ptl(saddr1+1,i)
          nnr(n_n+saddr2+j,4) = myproc+i

          ! insert ref. edge info into rtl
          rtl(r_cnt+1) = n_n+saddr2+j
          rtl(r_cnt+2) = nod_1
          rtl(r_cnt+3) = nod_2
          rtl(r_cnt+4) =
     &      BINSCH(nenrs(1,1),nenrs(1,2),ne_nn,ptl(saddr1+4,i))
          rtl(r_cnt+5) = ptl(saddr1+5,i)
          noel = ptl(saddr1+6,i)
          rtl(r_cnt+6) = noel
          do ii = 1,noel
            rtl(r_cnt+6+ii) = BINSCH(nenrs(1,1),nenrs(1,2),
     &        ne_nn,ptl(saddr1+6+ii,i))
          end do
          ! increase r_cnt for next edge
          r_cnt = r_cnt+nbm_6
        end do
        ! increase saddr2 for next overlap processor
        saddr2 = saddr2+p_cnt(i)
      end do
```

Listing 17: Code for the processing of the *ptl* array that has been sent to the right and received back with the global node numbers of the mid-points.

The next thing to do is to look if there are edges in the *ptl* array, the array for the overlap refinement edge information, a processor received from the left that this processor that is the owner of these edges wants to halve itself, too. This is illustrated in figure 4.5.4 where element A on processor 1 and element B on processor 2 both are refinement elements.

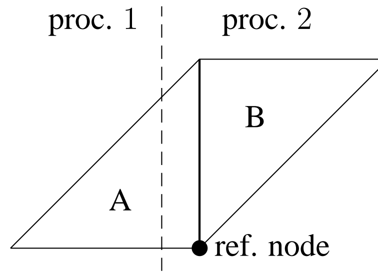


Figure 4.5.4: Illustration of a situation where the same refinement edge occurs in the received *ptl* array (received from processor 1) and the (local) *rtl* array of processor 2.

Then the information for these edges has already been inserted into the receiving processor's own *rtl* array for the local refinement edge information. So we go through all columns of the received *ptl* array and look if the value for the combination of end points of the current edge has already been set to *true* in the array *lnpl* (see on page 91), the array where we mark the end points of the edges we already regarded, i.e. if the second end point nod_2 is the nb_1^{th} neighbour node of the first end point nod_1 we

$$\text{check if } \; lnpl(nod_1, nb_1) = true.$$

Of course, if the same edge occurs several times in different columns of *ptl*, only the first occurence is examined. If the edge has not to be halved because of the necessary refinement of an own element, i.e. it holds

$$lnpl(nod_1, nb_1) = false,$$

we have to insert the edge data into *rtl* and therefore the processor appends the edge data to its own *rtl* array. We also have to increase the counter r_{cnt} that indicates the current end of the *rtl* array by nbm_6, the number of entries per edge. If the edge has also to be refined by its own processor, i.e. if

$$lnpl(nod_1, nb_1) = true,$$

we search in *rtl* for this edge that represents the same physical edge and insert the new local node number nod_l of the mid-point:

$$rtl(nbm_6 \cdot (l_{found} - 1) + 1) = nod_l \tag{4.5.14}$$

where l_{found} is the number of the matching edge in *rtl*. We introduce a one-dimensional **logical array *lglob*** with length equal to r_{cnt}/nbm_6 (here r_{cnt} still is the original value) which is the original number of edges in *rtl*. This array is initialized by *true* and must be set to *false* for all edges that already have got a mid-point because they are refinement edges on a left neighbour processor so that they do not get another mid-point by their own processor. So if l_{found} is the number of the edge in *rtl* that corresponds to the current edge in *ptl* we have to set

$$lglob(l_{found}) = \mathit{false}. \tag{4.5.15}$$

In both cases we insert the necessary information for the nodes into all four columns of the node information array *nnr*. With $saddr_2$ from (4.5.12) the first new local node gets the number

$$n_n + saddr_2 + 1$$

and for each new local node we increase this number by one. At the end $saddr_2$ is set to the last local node number on the processor. You can see the code for the inserting of the data in listing 18.

As each processor has inserted all necessary edge information into its local refinement edge array *rtl* we are able to issue the new global node numbers of those nodes we did not yet give a new number. These are the new mid-points of the edges that are not refinement edges on any left neighbour processor. We first transform the logical *lglob* array, where all the edges that still have to get a new global number for the mid-point have the value *true*, into a one-dimensional **integer array *iglob*** with length equal to r_{cnt} where we insert the starting addresses of the edges in *rtl*. The number of nodes to which we still must give a new global node number is denoted by m_{cnt}. In order to know the global number of the first new node we have to know how many nodes will be generated by the processors on the left side. So each processor sends its local m_{cnt} to the right in $np-1$ cycles exactly like we have done this before for the overlap nodes, see on page 132. In each cycle the processors add the received number to a counter npp_{ip} that indicates the number of nodes that will be generated on the left side at the end. This counter has been initialized by 0 at the beginning.

First we issue the new local node numbers and insert them into the first place of each edge in *rtl*. The new numbers go

$$\begin{aligned} \text{from} \quad & n_n + saddr_2 + 1 \\ \text{to} \quad & n_n + saddr_2 + m_{cnt}. \end{aligned}$$

As we have stored the starting addresses of the remaining refinement edges in *iglob* and in the first place of the edge information we have the local node number of the new mid-point, we set

$$rtl\,(iglob(i)) = n_n + saddr_2 + i \quad \text{for} \quad i = 1, \dots, m_{cnt}. \tag{4.5.16}$$

Then we can issue the new global node numbers. The new nodes get the numbers

$$\begin{aligned} \text{from} \quad & n + n_{ptl} + npp_{ip} + 1 \\ \text{to} \quad & n + n_{ptl} + npp_{ip} + m_{cnt}. \end{aligned}$$

In figure 4.5.5 the global numbering of the new nodes is illustrated.

Again, the total number of new nodes $n_{rs,new}$ is computed on the last processor np by

$$n_{rs,new} = n_{ptl} + npp_{np} + m_{cnt,np} \qquad (4.5.17)$$

```
      ! initialize lglob
      lglob  = .true.

      ! for each right overlap processor
      do i = 1,np_maxr
        ! initialize counter cnt
        cnt = 0
        ! for each new node generated by current overlap proc.
        do j = p_cnt(i)+1,p_cnt(i)+lp(i)

          ! check if first occurrence of the current node
          if (lpt(j-p_cnt(i),i)) then

            ! saddr1: starting address of current edge
            saddr1 = nbm_6*(j-1)
            ! increase counter cnt
            cnt = cnt+1

            ! nod_1/nod_2: local node nrs. of end pts. of the edge
            nod_1 = BINSCH(nnrs(1,1),nnrs(1,2),nnold,
     &        ptl(saddr1+2,i))
            nod_2 = BINSCH(nnrs(1,1),nnrs(1,2),nnold,
     &        ptl(saddr1+3,i))

            ! nb_1: neighbour number of nod_2 for nod_1
            nb_1 = NBSRCH(iinfo,nod_1,nod_2,ilev,idstar)
            ! nb_2: neighbour number of nod_1 for nod_2
            nb_2 = NBSRCH(iinfo,nod_2,nod_1,ilev,idstar)

            ! check if pair of end pts. is not yet ins. into lnpl
            if (.not.lnpl(nod_1,nb_1)) then
              ! insert node information into lnpl,
              ! nnr and rtl similar to listing 15
```

Listing 18: Code for the processing of the *ptl* array received from the left.

```
          else  ! pair is already inserted
            ! for each edge in rtl
            do l = 1,r_cnt
              ! saddr3: starting address of current edge in rtl
              saddr3 = nbm_6*(l-1)
              ! check if end points are indentical
              if (((rtl(saddr3+2) == nod_1).and.
   &                 (rtl(saddr3+3) == nod_2)).or.
   &                ((rtl(saddr3+2) == nod_2).and.
   &                 (rtl(saddr3+3) == nod_1))) then
                ! append new node to nnr array
                addr = n_n+saddr2+cnt
                nnr(addr,1)    = ptl(saddr1+1,i)
                nnr(addr,2)    = nnr(nod_1,2)
                nnr(addr,3)    = 1
                nnr(addr,4)    = myproc
                ! insert local node number into rtl
                rtl(saddr3+1) = n_n+saddr2+cnt
                ! lglob = false means node must not get
                ! new node number any more
                lglob(l)       = .false.
                ! corresp. edge found: no further search needed
                exit
              end if
            end do
          end if

        end if
      end do
      ! increase counter for local nodes
      saddr2 = saddr2+cnt
    end do
```

Listing 18: Code for the processing of the *ptl* array received from the left (continued).

and is scattered onto the other processors. At the end we connect the local node numbers with the global ones by inserting the data into the node information array *nnr*:

$$\left.\begin{array}{rcl} nnr(n_n + saddr_2 + i, 1) & = & n + n_{ptl} + npp_{ip} + i \\ nnr(n_n + saddr_2 + i, 2) & = & 1 \\ nnr(n_n + saddr_2 + i, 3) & = & 0 \\ nnr(n_n + saddr_2 + i, 4) & = & ip \end{array}\right\} \quad \text{for} \quad i = 1, \ldots, m_{cnt}. \tag{4.5.18}$$

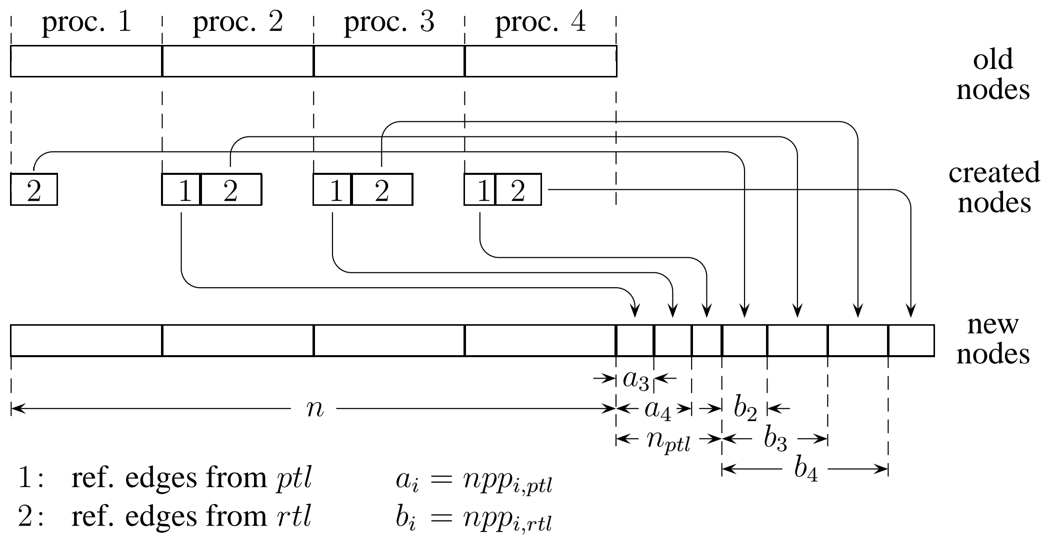


Figure 4.5.5: Illustration for global numbering of the new nodes originating from the *ptl* and the *rtl* array for $np = 4$ processors.

For the meaning of the columns see on page 94. We set

$$r_{cnt} \Leftarrow \frac{r_{cnt}}{nbm_6} \tag{4.5.19}$$

at the end, this value represents all edges that have been halved on this processor and therefore all new nodes that have been generated and got new local and global node numbers. In figure 4.5.6 we illustrate the local numbering of the new nodes on processor ip. The new overlap nodes result from the local ptl_{ip} array for the overlap refinement edge information, whereas the new own nodes result from the refinement edge information in *rtl*. ptl_{ov} is that part of *ptl* that has been received from the left overlap, therefore the new nodes resulting from this array are own nodes, too.

SUMMARY: We have generated all new nodes of the current refinement step but the nodes only consist of their numbers so far. We have to assign coordinates, consistency order and function values to them, and we have to insert the node numbers into the elements in which they occur.

Now the new nodes get their coordinates and order exactly as shown in (4.4.6) and (4.4.7). For the interpolation of the solution we have the problem that the difference stars for the overlap nodes are not at our disposal. So if one of the end points of a refinement edge is an overlap node, we just take the value that we interpolated by the evaluated influence polynomials of the other end point (that is owned by the processor) as function value for the new node. This does no harm as the function values are only starting values for the

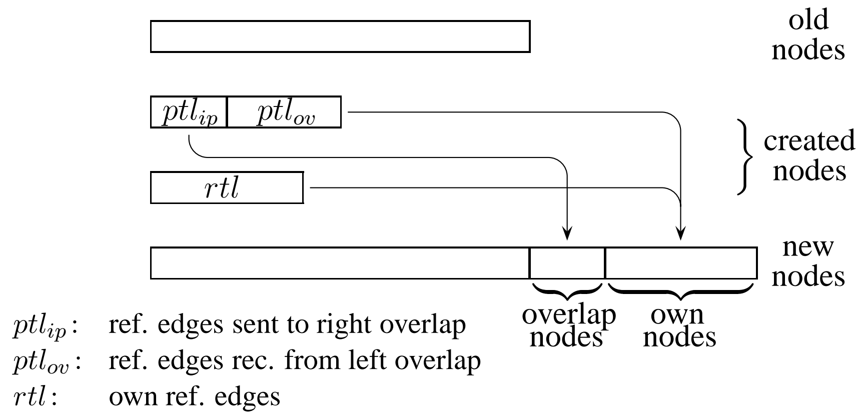


Figure 4.5.6: Illustration for local numbering of the new nodes originating from the *ptl* and the *rtl* array on processor ip.

Newton iteration in the following computation cycle.

There is no difference to the single processor run for the update of the element information in the *nek* array, i.e. the array where we store the local node numbers of the local elements. We described this update on page 96: we set the entry for the current element el_i and the current edge that has the local position pos_i in the element to the local node number that corresponds with them (see listing 10):

$$nek(el_i, pos_i) = rtl\,(nbm_6 \cdot (i-1)+1) \quad \text{for} \quad i = 1, \ldots, r_{cnt}. \tag{4.5.20}$$

Then we update the information of the neighbour elements of the refinement elements in the *nek* array as described on page 96. As already mentioned there we do not know the local edge number in the neighbour elements where we have to insert the new node numbers. Of course, the algorithm shown in listing 11 is not very efficient but somewhere we have to spend the time to identify the correct edge number anyway.

There is not any difference to the single processor run in the check if a new node is a boundary node so that the procedure on page 97 is also valid here. Only for the checking of the boundary node limit we have to take into account that nb_{max} is a local value on a distributed memory parallel computer and therefore

$$\text{check if} \;\; neb_n + nb_{new} > nb_{max}$$

where neb_{new} is the number of new external boundary nodes on a processor.

SUMMARY: The neighbour elements of an own refinement element of a processor may be owned by an overlap processor. So they belong to the overlap of the processor that owns the refinement element, and if we insert the new nodes into these overlap elements

we must ensure that they are also inserted into the elements on their own processors. So the node and element information must be passed to the overlap processors where the new node is inserted into the neighbour element and—if necessary—appended to the node list.

As seen in listing 11 on page 98 we updated the *nek* array for the neighbour elements of all refinement edges on the processor that owns and therefore refines the element by inserting the numbers of the new nodes into the rows in the *nek* array that correspond to the neighbour element numbers. But if the neighbour elements are not owned by the same processor, we did only update the information for overlap elements with that. We did not update the *nek* array on the processors that own the neighbour elements! So we go through the array for the local refinement edges, i.e. the *rtl* array, and look for each edge for neighbour elements that share this edge and that are owned by a different processor. First we look for elements in the left overlap, in a second step for elements in the right overlap. We explain the procedure for the elements in the left overlap, for the elements in the right overlap it works analogously. If we find an element as described above, we put all necessary information, which is global element number, local position of the mid-point in the element and the information that is stored in the node information array *nnr* for the new mid-point of the edge, into the corresponding column of a buffer *sndbuf*.

We have to determine the length of this buffer first. So we count the elements that have to be sent to each left overlap processor. This number is denoted by $ne_{snd,ip}$. We compute the maximum $ne_{snd,max,ip}$ of $ne_{snd,ip}$ over the overlap processors and then we compute the global maximum $ne_{snd,max,g}$ of this maximum over the processors:

$$\begin{aligned} ne_{snd,max,ip} &= \max_{ip=1}^{np_{max,l}} \; ne_{snd,ip}, \\ ne_{snd,max,g} &= \max_{ip=1}^{np} \; ne_{snd,max,ip} \end{aligned} \tag{4.5.21}$$

Therefore the buffer length is equal to $7 \cdot ne_{snd,max,g}$ and the width is equal to $np_{max,l}$, see table 4.5.4. The local position in the element is known because we already inserted the node in the neighbour element on the sending processor, so we do not need to search for this position on the target processor again. The corresponding column col_{rel} in the send buffer *sndbuf* is

$$col_{rel} = ip_{rel} + 1 + np_{sl} \tag{4.5.22}$$

with

$$ip_{rel} = ip_{own} - ip \tag{4.5.23}$$

where ip_{own} is the processor where the data has to be sent to and np_{sl} is the last processor on the left side to send data to. We already explained on page 79 in subsection 4.3.2 that we use this storage scheme to save memory.

k	$sndbuf(sndcnt(col_{rel}) + k, col_{rel})$
1	global element number
2	local position of new mid-point in the element
3	global node number of new mid-point of the current edge
4	number of subdomain new mid-point belongs to
5	number of coupling subdomains of new mid-point
6	owning processor of new mid-point
7	lr_{new} (see text)

Table 4.5.4: Storage scheme of array *sndbuf* for inserting data of an element of processor ip_{own} in column col_{rel}.

As the new mid-point we want to insert into the *nek* array on the receiving processor has just been generated by the sending processor, anything is known about this node on the receiving processor. So we also have to insert the necessary information into the node information array *nnr*, see table 4.4.2. In order to avoid multiple inserting of the same node (because of several neighbour elements that are owned by the same processor) we reserve the last entry of an edge for a value lr_{new} that indicates if the node should be inserted into the *nnr* array. We set

$$\begin{rcases} lr_{new} = 1 \\ lr_{new} = 0 \end{rcases} \text{ if the node should } \begin{Bmatrix} \text{be} \\ \text{not be} \end{Bmatrix} \text{ inserted.} \tag{4.5.24}$$

To determine the value of lr_{new} we have a **logical array *logsnd*** with r_{cnt} rows and np_{max} columns that is initialized by *true*. We go through the refinement edges in *rtl* and insert the neighbour elements of the current edge into *sndbuf* in the column col_{rel}. If a neighbour element is the first one for processor ip_{own}, lr_{new} is set to 1 and the array entry corresponding to the new node is set to *false* for this processor. For the following neighbour elements owned by the same processor the entry in the logical array is already *false*, so we set $lr_{new} = 0$.

A counter array *sndcnt* with length equal to np_{max} gives us the number of elements for which we have to send data to the neighbour processors. After inserting the information for all elements into the array *sndbuf* we send the counter array *sndcnt* to the left in $np - 1$ communication cycles. Again, we make use of the communication pattern presented in subsection 4.1.4. Afterwards each processor is ready to receive the message(s) with the element data. Here we only send data to neighbour processors where it is necessary as usual. We need $np_{max,l}$ communication cycles for the data exchange and in each cycle i the processors pass a message of length

$$\ell = 7 \cdot sndcnt(col_{target}) \cdot mem_{int} \tag{4.5.25}$$

to processor $ip_{target} = ip - i$ if

$$sndcnt(col_{target}) > 0 \tag{4.5.26}$$

where

$$col_{target} = 1 + np_{sl} - i \tag{4.5.27}$$

is the column in *sndbuf* where the data for processor ip_{target} is stored.

The target processor receives the message in the receive buffer *rcvbuf* which is a one-dimensional array of length $7 \cdot ne_{snd,max,g}$ as we always process the received data of the current communication cycle before the next cycle begins where the data in *rcvbuf* is overwritten. We go through the received data of the elements and first transform the global element number el_{global} into the corresponding local one (el_{local}) by the means of array *nenrs*, see on page 82. Then we insert the new nodes in the *nek* array, the array for the node numbers of the local elements:

$$el_{global} \quad \{ = rcvbuf(1) \ \} \overset{\text{nenrs}}{\Longrightarrow} el_{local},$$

$$nek(el_{local}, pos) = nod_{local} \tag{4.5.28}$$

where nod_{local} is set to the last local number we generated on a processor so far at the beginning and increase this value by one each time we have to insert a new node. This is necessary if $lr_{new} = 1$ holds, i.e. then we increase the number of local nodes by one and copy the node information from the receive buffer to the node information array *nnr*, see listing 19:

$$nnr(n_{l,new}, k) = rcvbuf(addr + 2 + k) \quad \text{for} \quad k = 1, \ldots, 4. \tag{4.5.29}$$

When we have finished the communication to the left, we do the same for those neighbour elements that are owned by processors in the right overlap. The target processor in communication cycle i is $ip_{target} = ip + i$ and the corresponding column col_{target} in *sndbuf* now is

$$col_{target} = ip_{target} - ip = i \tag{4.5.30}$$

because the processors where we do never send data to are on the right side and so we simply can reduce the width of the array *sndbuf*.

RECAPITULATION: By this step we have updated the element information of those elements that are not refinement elements themselves but they are only neighbour elements of a refinement element (thus get additional mid-point(s)) and furthermore are owned by a different processor than the refinement element. Then the processor that owns the refinement element must send the information that a new mid-point must be inserted into the neighbour element to the owning processor of this element and the

```
    ! for all received elements
    do i = 1,rcvcnt
      ! addr: starting address of current element
      addr = 7*(i-1)
      ! el: local number of current element
      el   = BINSCH(nenrs(1,1),nenrs(1,2),ne_n,rcvbuf(addr+1))
      ! pos: local position of mid-point in current element
      pos  = rcvbuf(addr+2)

      ! check if node must be inserted into nnr
      if (rcvbuf(7*i) == 1) then
        ! increase node counter and insert node info into nnr
        n_lnew = n_lnew+1
        nnr(n_lnew,1) = rcvbuf(addr+3)
        nnr(n_lnew,2) = rcvbuf(addr+4)
        nnr(n_lnew,3) = rcvbuf(addr+5)
        nnr(n_lnew,4) = rcvbuf(addr+6)
      end if

      ! insert node into nek array
      nek(el,pos) = n_lnew
    end do
```

Listing 19: Code for the processing of the received data for neighbour elements of refinement elements owned by an overlap processor.

owning processor of the neighbour element must insert the node information into its *nnr* array and insert the node number into its *nek* array.

SUMMARY: We divide the refinement elements of refinement stage rs into four elements of refinement stage $rs + 1$. Therefore we have to issue the new element numbers for three of them, one gets the old element number. Furthermore, we have to determine the owning processor of the newly-created elements according to rule 4. If a new element is not owned by the processor by which it has been created the information about this element must be sent to the owning processor.

The next thing in this refinement step to do is the generation of the new elements. As all necessary information is available on the processors this can be done purely local without communication. However, we must determine the number of elements that the processors on the left side generate in order to issue the new global element numbers becuse they are appended to the previous element numbers. This is done like described for the numbering of new nodes by m_{cnt} on page 138, in fact we send both numbers at the same time to the right to save communication time. In each cycle the received value is added to an integer $nepp_{ip}$ that has been initialized by 0. Then the global element numbers are issued. The new

elements on processor ip get the global numbers

$$\begin{array}{rl} \text{from} & ne + nepp_{ip} + 1 \\ \text{to} & ne + nepp_{ip} + 3 \cdot ne_{re} \end{array}$$

where $ne_{re} = narpl(rs + 1, 2)$ is the number of refinement elements in the current refinement step. At the end we connect the local element numbers with the global ones by inserting the data into the element information array *nenr* (see on page 78):

$$\left.\begin{array}{rcl} nenr(ne_n + i, 1) & = & ne + nepp_{ip} + i \\ nenr(ne_n + i, 2) & = & 1 \\ nenr(ne_n + i, 3) & = & 0 \\ nenr(ne_n + i, 4) & = & ip_{own,i} \end{array}\right\} \quad \text{for} \quad i = 1, \ldots, 3 \cdot ne_{re}. \qquad (4.5.31)$$

According to rule 4 the new elements belong to the processor that owns the leftmost node. So we have to determine this processor ip_{own} for each of the four new elements. This also holds for the element that gets the old element number, i.e. the elements with the old numbers may change the processor during a refinement step! But this is only possible under certain circumstances. Let us look at the example illustrated in figure 4.5.7: There are two old elements that both must be refined. The upper triangle is owned by processor ip as node 1 is owned by processor ip, so both edge 1 and 3 are owned by processor ip, and therefore the new nodes 4 and 6 are also owned by processor ip by our rules. As there are not any new elements neither node 4 nor node 6 belong to, each of the new elements has a node owned by processor ip, and therefore each new element is owned by processor ip. The difference between the lower and the upper element is that edge 1 in the lower triangle has already a mid-point when we refine the triangle. After the refinement the new node 6 is owned by processor ip, but the nodes 4 and 5 are owned by processor $ip + 1$. So all nodes contained in triangle 6 are owned by processor $ip + 1$, and therefore the triangle is also owned by this processor. We learn that there are two possibilities how a new element may change the owning processor, and the difference between the two possibilities is the number of corner nodes of the old element that are owned by processor ip. If only one of the corner nodes of the old element is owned by processor ip, there must already exist at least one of the mid-points of the two edges that share this node before the refinement process (for illustration see figure 4.5.7). If two corner nodes are owned by processor ip, both of the mid-points of the two edges that share the third node on processor $ip + 1$ or $ip + 2$ must already exist before the refinement process.

The element number of the ith old element $el_{old,i}$ is stored in the integer array *indrel* for the local refinement elements:

$$el_{old,i} = indrel\,(narpl(rs + 1, 1) + i - 1) \quad \text{for} \quad i = 1, \ldots, ne_{re}. \qquad (4.5.32)$$

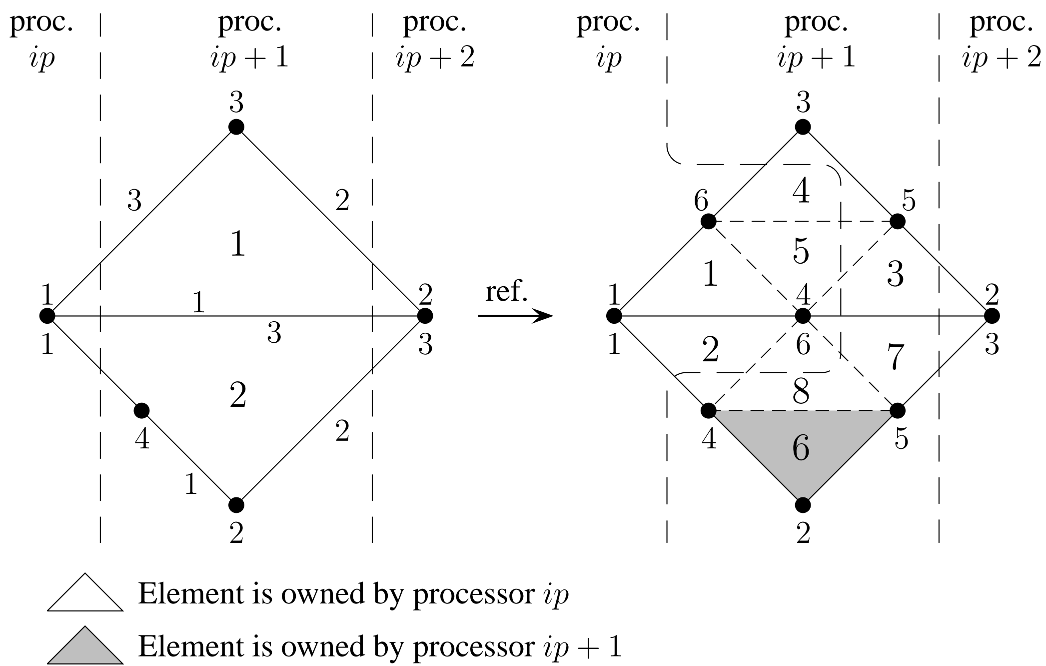


Figure 4.5.7: Illustration of the owning of the new elements in 2-D.

The element information of this old element $el_{old,i}$ is already stored in *nenr* so that we only have to update the number of the owning processor:

$$nenr(el_{old,i}, 4) = ip_{own,i} \quad \text{for} \quad i = 1, \ldots, ne_{re}. \tag{4.5.33}$$

If one of the four newly-created elements is not owned by the processor that generates this element we must send this information to the owning processor ip_{own}. This is because at the end of the refinement process, we throw away all overlap information and make a completely new sorting of the grid onto the processors again. So the processor that generates the element will delete its information as the element is in the overlap, and on the processor that owns the element anything would be known about its existence at all. Therefore the element information would get lost.

As we learned at the end of subsection 4.3.2 on page 85 we have got two integer arrays *rcvl* and *rcvr* for the storage of the numbers of refinement elements that have to be refined because of the refinement cascade. These arrays have to be sent back to the processors that own the evoking elements because we need the information about the refined elements for the refinement of the evoking elements (see below). So we check for each new element that belongs to the overlap if it already occurs in the array *rcvl* (if $ip_{own} < ip$) or in *rcvr* (if $ip_{own} > ip$) where ip_{own} is the owning processor of the element. If not, we just append it to the corresponding array.

SUMMARY: We generated all elements of the current refinement step and must now send the information about what happened to an old element to those overlap processors that own refinement elements of the next refinement stage that are neighbour elements of the old element because for their refinement we need the element information of the refined elements. We store all necessary information in a buffer that is passed to the overlap processor where the contents of the element information arrays is updated.

So the first refinement step is nearing completion now. If there are not any refinement elements of a higher refinement stage we have finished and can continue with the new sorting of the grid onto the processors. Otherwise, we have to update the element information for those elements on the neighbour processors that had to be refined because of the refinement cascade. This has already been explained at the end of subsection 4.3.2. The reason is that we need the element information of the refined elements on a processor in order to be able to execute the refinement of the evoking elements of the refinement cascade on the neighbour processor. If a refinement element el_{ip} of stage rs on processor ip has been refined because of the refinement cascade that has been caused by refinement element el_{ip-1} of stage $rs+1$ on processor $ip-1$ the information about the four new elements that have been developed from element el_{ip} must be sent to processor $ip-1$ so that this processor is also able to update its *nek* array with the node numbers of the local elements. Then the refinement of element el_{ip-1} can be carried out in the next refinement step.

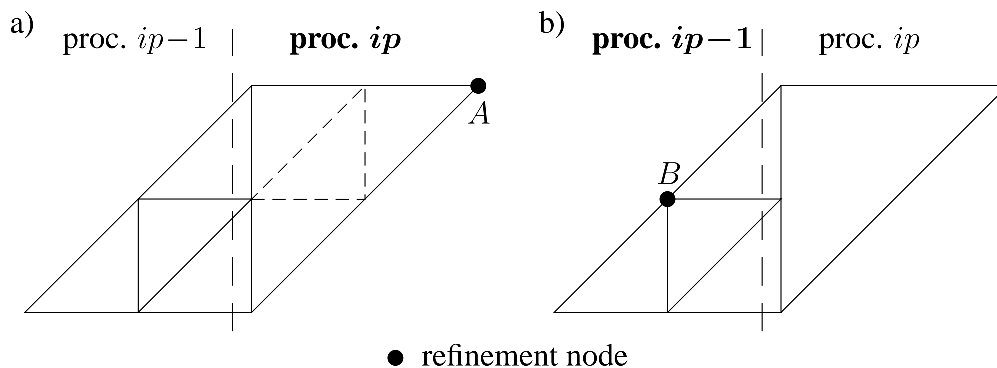


Figure 4.5.8: Illustration of the known information at the end of a refinement step a) on refinement processor ip, b) on overlap processor $ip-1$.

An example will illustrate the necessity of this update step between two refinement steps. In figure 4.5.8 the situation at the end of a refinement step is shown for two neighbour processors ip (figure 4.5.8a)) and $ip-1$ (figure 4.5.8b)). Note that you can see the same situation both in figure 4.5.8a) and b), the difference is the known information on the respective processor. As there is a refinement node (node A) in the right element on processor ip the right element has been refined in the current refinement step and the new elements have been generated on the right side (dashed lines). If we tried to refine the left upper element

of processor $ip-1$ in the next refinement step (because of the refinement node B), we would generate an additional node on the left vertical edge of the right element as the information of the refinement of this element is missing on processor $ip-1$. Note that the right element has not only to be refined because of the refinement node A in the current refinement step but also because of the refinement cascade evoked by refinement node B.

But the first thing to do is to check if there are any refinement elements at the processor borders that have to be refined because of the refinement cascade at all. So we introduce two **counters** $\boldsymbol{le_{cnt}}$ and $\boldsymbol{re_{cnt}}$ for elements of which we have to send the updated *nek* array to the left or right, respectively. If both counters are equal to zero on all processors there is nothing to update and we only have to add the data for the current refinement stage to the *nbrs* array, the array where we store the numbers of own and overlap nodes and elements, and continue with the elements of the next refinement stage.

If there is any element information that has to be passed to a neighbour processor we have to do the following. We explain the processing for the communication to the left side, the communictation to the right side works analogously. First of all we sort the elements in *rcvl* (see on page 85), i.e. the array where we stored the numbers of the elements of which we have to send the information to the left, by the processor number (in ascending order) they must be sent to. Therefore we need an **integer array** ***rcvlct*** of length np and width 2. Here we store the number of elements per processor in the first column and the starting address of the elements for a processor in *rcvl* in the second column. We determine the leftmost processor to send data to which is

$$ip_{left} = rcvl(1,3). \tag{4.5.34}$$

So the number of cycles within the communication is

$$ip_{max,l} = \max_{ip=1,\ldots,np} \left(ip - ip_{left,ip}\right). \tag{4.5.35}$$

As usual, only those processors send data that really must send data to the current target processor. So we first have to pass the number of elements, for which we have to send the information to the overlap processors and by which those overlap processors can compute the message lengths they will receive, to the left. Therefore we allocate a send buffer *sndbuf* with length equal to $ip_{max,l}$ where we set

$$sndbuf(i) = rcvlct(ip - ip_{left} + 1 - i, 1) \quad \text{for} \quad i = 1,\ldots, ip - ip_{left}. \tag{4.5.36}$$

By this storage we have the number of elements for the direct left neighbour processor stored in the first entry of *sndbuf*, followed by the second left neighbour processor etc. just like we did for the array *sndlto* that we used during the refinement cascade, see on page 79. This buffer is sent to the left in $np-1$ cycles by the means of the well-known

communication pattern presented in subsection 4.1.4. Afterwards we send the messages with the real data to the target processors in $ip_{max,l}$ cycles. Within a communication cycle i the number of elements ne_s that must be sent to processor $ip - i$ is

$$ne_s = rcvlct(ip, 1). \tag{4.5.37}$$

ne_s determines the length l_{infarr} of the message that is sent to the neighbour processor which is computed by

$$\begin{aligned} l_{infarr} &= 2 \cdot n_{edge} \cdot ne_s + 4 \cdot (el_{new} + 1) \cdot ne_s + nolnod \cdot (el_{new} + 1) \cdot ne_s + 1 \\ &= (2 \cdot n_{edge} + (4 + nolnod) \cdot (el_{new} + 1)) \cdot ne_s + 1 \qquad (4.5.38) \\ &= 46 \cdot ne_s + 1 \end{aligned}$$

with

$$n_{edge} = 3, \quad el_{new} = 3, \quad nolnod = 6 \text{ in 2-D.}$$

So the information we have to send to the left consists of 46 integer numbers for each element in 2-D! This comes from the fact that we have to send the information of the element information array *nenr* ($= 4 \cdot (el_{new} + 1)$) and of the *nek* array ($= nolnod \cdot (el_{new} + 1)$). Additionally, we have to send the information of newly-created nodes in the element that are not owned by the receiving processor because these nodes are not yet known on this processor. Here we only need to send the global node number and the processor that owns the node ($= 2 \cdot n_{edge}$ at most) because these nodes are only overlap nodes that must be inserted into the *nek* list on the neighbour processor. The processor number is needed to determine the owning processor of the newly-created elements. In the first entry of the one-dimensional **integer array *infarr*** with length equal to l_{infarr} that we use for the storage of the data we store the value n_s which is the number of nodes we have to send to processor $ip - i$. One could suppose that we could have sent this number together with the number of elements ne_s in the preparatory step to the left but in contrast to ne_s the value for n_s is not known at the beginning. So it is less expensive to increase the message length by one integer value than to go through the elements twice, first checking how many nodes have to be sent to the left before really inserting them the second time. The storage scheme of array *infarr* is shown in figure 4.5.9.

As we have stored in *rcvl* the numbers of all elements in the left overlap that have to be refined because of the refinement cascade without paying attention to their refinement stage, and we only want to send the data for the refinement elements of the current refinement step we must exclude all elements that are of a higher refinement stage (there are not any elements of a lower refinement stage as we will see in the next paragraph). So we have to check if for the refinement stage rs_{el} of an element el holds

$$rs_{el} = rs - 1 \tag{4.5.39}$$

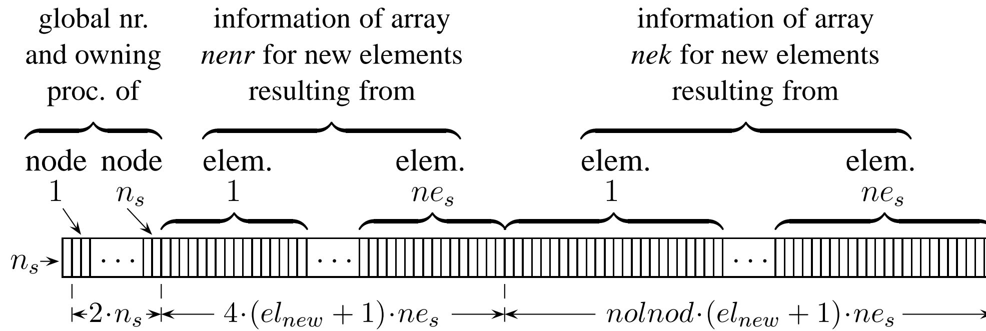


Figure 4.5.9: Storage scheme of array *infarr*.

where rs is the number of the current refinement step. If (4.5.39) holds, the element information is inserted into the array *infarr*, and if the node numbers of the new mid-points of this element are not yet stored in *infarr*, we make up leeway now.

After a processor sent the information of the refined elements of which the refinement results from the refinement cascade to those processors that own the evoking elements, we do not need the information that the information of this element must be sent to the overlap processor anymore because this element will not be refined once more during the same refinement cycle. So we set

$$rcvl(el, 1) = 0 \tag{4.5.40}$$

after having inserted all data for element el into *infarr*. When each receiving processor updated its *nek* and its *nenr* array for the elements and its *nnr* array for the nodes, we delete all entries from *rcvl* with a zero in the first column. This means that the elements of the refinement stage we just refined are deleted, so that we only have got elements of a higher refinement stage in the *rcvl* array. This is done to save time in the following refinement step when we again have to check the refinement stage of the elements by (4.5.39) in the update step. Otherwise we would check in each update step all elements that have already been updated in the preceding refinement steps and therefore do not have to be updated again, which makes no sense and therefore wastes time.

When we have stored the data of all elements in *infarr* in communication cycle i this array is passed to processor $ip - i$ where the message is received in a buffer *infbuf* that has also the length l_{infarr} (note that this value can be computed on the target processor because the number of elements for which the target processor will receive data is already known).

The receiving processors set

$$\begin{aligned} n_{r,ip-i} &= infbuf(1) = n_{s,ip} \quad \text{and} \\ ne_{r,ip-i} &= ne_{s,ip} \end{aligned}$$

where $n_{s,ip}$ is the number of nodes and ne_s is the number of elements for which we have stored the information in *infarr* or *infbuf*, respectively. We first check if the maximum number of nodes n_{max} or elements ne_{max} would be exceeded by the new nodes and elements. If yes, we print out an error message and stop the communication. Otherwise, we first append the new nodes to the node information array *nnr*. As the data exchange is only possible by global node and element numbers, we have to transform the (global) node numbers of the received elements into local node numbers. The processor just received new nodes that occur in the received elements, so we have to invert the first column of the *nnr* array with the global node numbers in order to get the updated *nnrs* array with the local node number for each global node that belongs to the processor. The next thing to do is the update of the old element, i.e. we overwrite the information for the old element stored in the array *nek* by the information the processor received from its neighbour processor. To get knowledge of the local number of this element on the receiving processor we have to transform the received global element number into the corresponding local one by a binary search in the *nenrs* array:

$$infbuf\ (2\cdot n_r+4\cdot(i-1)\cdot(el_{new}+1)+2) = el_{global} \overset{nenrs}{\Longrightarrow} el_{local}$$

for $i = 1, \ldots, ne_s$. Then we overwrite the row for this element in the array for the node numbers of each local element, i.e. the *nek* array, by the new node numbers for this element. Here we simultaneously transform the received global into local node numbers by a binary search in the updated *nnrs* array:

$$\begin{aligned} infbuf\ (2\cdot n_r+4\cdot ne_r\cdot(el_{new}+1)+ & \\ nolnod\cdot(k-1)\cdot(el_{new}+1)+2) &= nod_{global} \overset{\text{nnrs}}{\Longrightarrow} nod_{local}, \\ nek(el_{local}, k) &= nod_{local} \end{aligned} \tag{4.5.41}$$

for $k = 1, \ldots, nolnod$. The refinement stage of the updated old element is increased by one. The information of the three newly-created elements (by the subdivision) is appended to the array *nek* and the element information array *nenr*, respectively, i.e. we transform the global node numbers of the three received new elements into local numbers and insert them at the end of the *nek* array, furthermore we insert the global element number, domain number, dividing line property and own processor at the end of *nenr*. The refinement stage of the new elements is set to the same value as that one of the updated old element. At the end, we invert the *nenr* array to get the new *nenrs* array, i.e. the array for the local element number of each global element that belongs to the processor, and afterwards the communication with the next processor will start.

After having finished the $ip_{max,l}$ cycles for the communication to the left, the same takes place for the communicaton to the right. The number of communication cycles is

$$ip_{max,r} = \max_{ip=1,\ldots,np} (ip_{right} - ip) \tag{4.5.42}$$

with

$$ip_{right} = rcvr(re_{cnt}, 3). \tag{4.5.43}$$

For the communication to the right we allocate an integer array *rcvrct* where we store the number of elements per processor and the starting addresses of the elements for a processor in the sorted array *rcvr*. For the execution of the preparatory step of the data exchange we must update the send buffer *sndbuf* by setting

$$sndbuf(i) = rcvrct(ip - ip_{left} + i) \quad \text{for} \quad i = 1, \ldots, ip_{right} - ip \tag{4.5.44}$$

and send this buffer to the right in $np - 1$ cycles. During the following $ip_{max,r}$ cycles the element data is sent to processor $ip + i$ in cycle i where the message itself is arranged the same way as for the communication to the left and there is no difference in the work of the target processors.

RECAPITULATION: If we refine elements in a refinement step that have neighbour elements on an overlap processor which must be refined in the next refinement step, we first must send the information what happened to the larger elements in the current refinement step to this overlap processor. Otherwise, we would create an additional (fourth) node on an edge of the (then unchanged) larger element on the overlap processor by the refinement of the smaller element in the following refinement step. The concerned larger elements are exactly those that have to be refined because of the refinement cascade. So the information about the generated new elements and the mid-points that have been created on the edges of these elements must be sent to those neighbour processors that must refine the neighbour elements, that evoked the refinement, in the following refinement step. On the target processors the information is appended to the node and element lists.

Before the refinement process we had stored the old own elements from the original mesh at the beginning of the element arrays *nek* and *nenr*, followed by the old overlap elements. With the refinement of the largest elements we have generated new own elements that have been appended to the element arrays, and by the update step we received new overlap elements from the neighbour processors that we also appended to the element arrays. We want to group all own and all overlap elements that have been added during the current refinement step to the element arrays, because this grouping makes it easier to differentiate between these two types of elements, becuse then we only need two rows per refinement stage/step in the *nbrs* array for the numbers of own and overlap nodes and elements.

First the elements that have been newly-created by a processor in the current refinement step and that have been stored in the element arrays *nek* and *nenr* on a processor before the beginning of the update step are sorted by processor number where we start with the processors 1 to $ip - 1$ (overlap elements), continue with the processors $ip + 1$ to np (overlap elements) and finish with the elements owned by processor ip (own elements). The

elements in the overlap result from the fact that the newly-created elements have not to be owned by the same processor as the old refinement element, see on page 148. So the last old elements that have already been stored on processor ip before the refinement process are overlap elements as mentioned above, and by the re-sorting the first new elements are overlap elements again. The number of new overlap elements of this sorting is denoted by $ne_{overl,1}$, the number of new own elements by $ne_{own,1}$. Then we sort the elements that have been added during the update step by processor number. These elements are identified by their starting address which is the local number of the last element stored before the update step started, increased by one. This time we start with the elements owned by processor ip (own elements), continue with the processors 1 to $ip - 1$ (overlap elements) and finish with the processors $ip + 1$ to np (overlap elements). This is because then the new own elements are all stored one after the other in the element arrays as the first sorting of the elements ends with own elements of the processor and at the beginning of the second sorting of the elements we have stored own elements, too. The number of new own elements in the update step is denoted by $ne_{own,2}$, the number of new overlap elements in the update step by $ne_{overl,2}$. Figure 4.5.10a) illustrates the storage scheme of the elements on a processor at the end of a refinement step. Again, from the first column of the element information array *nenr* the inverse *nenrs* is computed where we store the local element number for each global element that belongs to the processor.

a)

$$ne_{n,new} \begin{cases} ne_n & \text{number of elements before current refinement step} \\ ne_{overl,1} & \text{number of new overlap elements by ref. of own elements} \\ ne_{own,1} & \text{number of new own elements by ref. of own elements} \end{cases}$$

$$ne_{upd} \begin{cases} ne_{own,2} & \text{number of new own elements by update step} \\ ne_{overl,2} & \text{number of new overlap elements by update step} \end{cases}$$

b)

$$n_{n,new} \begin{cases} n_n & \text{number of nodes before current refinement step} \\ n_{overl,1} & \text{number of new overlap nodes by ref. of edges in } ptl \\ n_{own} & \text{number of new own nodes by ref. of edges in } rtl \end{cases}$$

$$n_{overl,l} \begin{cases} n_{overl,2,l} & \text{number of new left overlap nodes (page 143ff.)} \\ n_{overl,3,l} & \text{number of new left overlap nodes by update step} \end{cases}$$

$$n_{overl,r} \begin{cases} n_{overl,2,r} & \text{number of new right overlap nodess (page 143ff.)} \\ n_{overl,3,r} & \text{number of new right overlap nodes by update step} \end{cases}$$

Figure 4.5.10: Illustration of a) the numbers of elements stored in the element arrays *nek* and *nenr*, b) the numbers of nodes stored in the node arrays *nnr*, *x*, *y*, *u* and *q* at the end of a refinement step.

The re-sorting of the nodes is necessary for the same reason as the re-sorting of the ele-

ments. Before the refinement process we had stored the old nodes from the original mesh at the beginning of the node arrays *nnr*, *x*, *y*, *u* and *q* (for general information, coordinates, solution and order of the nodes), followed by the old overlap nodes. With the refinement of the largest elements we have generated $n_{overl,1}$ new overlap nodes from the *ptl* array and $n_{own,1}$ new own nodes from the *rtl* array that have been appended to the node arrays. From the updating of the *nek* array for neighbour elements of refinement elements that are owned by overlap processors as described on page 143ff., we have got $n_{overl,2,l}$ overlap nodes in the left overlap and $n_{overl,2,r}$ overlap nodes in the right overlap. In the update step a processor receives $n_{overl,3,l}$ overlap nodes from the left and $n_{overl,3,r}$ overlap nodes from the right. The reason for the differentiation between overlap nodes received from the left and from the right is the following: When we introduced the *nbrs* array on page 127 we separated the overlap nodes into the nodes belonging to the left overlap and those nodes belonging to the right overlap because otherwise we could not find out which edges are in the right overlap which is necessary for the generation of the new mid-points of the refinement edges. So we re-sort the overlap nodes, i.e. we change the order of the $n_{overl,2,r}$ right overlap nodes from the process described on page 143ff. and the $n_{overl,3,l}$ left overlap nodes from the update step, see figure 4.5.10b).

The last necessity in the update step is the updating of the *nbrs* array for the node and element data, i.e. the array where we store the number of own and overlap nodes and elements that we introduced on page 127. For the elements we group the old overlap elements and the $ne_{overl,1}$ elements resulting from the refinement of the own elements. They form the new "old" overlap. The new own elements resulting from the refinement of the own refinement elements and from the update step comprise the new own elements of the current refinement step. After them, the new overlap elements follow. We insert the data for the elements into the *nbrs* array, cf. table 4.5.2 and corresponding text on page 127:

$$\begin{aligned} nbrs(3 \cdot rs, 2) &= ne_n + ne_{overl,1} \\ nbrs(3 \cdot rs + 1, 2) &= ne_{own,1} + ne_{own,2} \\ nbrs(3 \cdot rs + 3, 2) &= ne_{n,new} + ne_{upd} \end{aligned} \tag{4.5.45}$$

with the notations from figure 4.5.10a). The updating of the *nbrs* array for the nodes is quite difficult. Like for the elements we group the old overlap nodes and the $n_{overl,1}$ new overlap nodes resulting from the refinement of the edges in the own array for the overlap refinement edge information, i.e. the own *ptl* array. The left overlap of the current refinement step is formed by the $n_{overl,l} = n_{overl,2,l} + n_{overl,3,l}$ left overlap nodes, the right overlap by the $n_{overl,l} = n_{overl,2,r} + n_{overl,3,r}$ right overlap nodes that result from the updating of the *nek* array for neighbour elements of refinement elements that are owned by overlap processors as described on page 143 and from the data exchange during the update step,

see figure 4.5.10b). So we insert the node data into the *nbrs* array as following:

$$\begin{aligned} nbrs(3 \cdot rs, 1) &= n_n + n_{overl,1} \\ nbrs(3 \cdot rs + 1, 1) &= n_{own} \\ nbrs(3 \cdot rs + 2, 1) &= n_{new} + n_{overl,l} + 1 \\ nbrs(3 \cdot rs + 3, 1) &= n_{new} + n_{overl,l} + n_{overl,r} \end{aligned} \tag{4.5.46}$$

with the notations from figure 4.5.10b). The algorithm for the update step is shown in algorithm E.

infarr: integer array for sending update information
nbrs: integer array for numbers of nodes and elements during ref. step
nnr: integer array for node information
rcvlct: integer array for number and starting address of the elements in *rcvl* per processor

E1 Check if update is necessary at all. If not, continue with step E6

E2 Compute array *rcvlct* (see page 150).

E3 For each left overlap processor:

- **a)** Compute message length l_{infarr} for messages to send to the left.
- **b)** Insert node and element information into *infarr* array.
- **c)** Pass message to the current overlap processor.
- **d)** Receiving processors:
  1. Check if maximum numbers of nodes or elements would be exceeded by inserting the new nodes and elements. If yes, exit and stop.
  2. Insert node information into *nnr* array.
  3. Update element information for old elements in element arrays.
  4. Append element information for new elements to element arrays.

E4 Repeat steps E3a) to d) for right overlap processors.

E5 Re-sort elements.

E6 Update *nbrs* array.

Algorithm E: Algorithm for updating the element information of elements to be refined because of the refinement cascade on overlap processors.

As the nodes that a processor receives during the update step are only overlap nodes, the columns in *nbrs* for the three kinds of boundary nodes—external boundary nodes, dividing

line and sliding dividing line nodes—can be updated as soon as we check the boundary node property. We do not store any new boundary nodes of the overlap on a procesor during the mesh refinement process. We need the boundary nodes of the overlap only once during a refinement step, namely when we determine the new boundary nodes at a processor border if one of the two end points of an edge is in the overlap. But as all refinement elements and thus all refinement edges are fixed before the refinement process is started, the newly-created boundary nodes in the overlap cannot be an end point of a refinement edge and therefore we do not have to store them. So we set

$$\left.\begin{array}{rcl} nbrs(3 \cdot rs + 1, j) & = & nb_{own} \\ nbrs(3 \cdot rs + 3, j) & = & nbrs(3 \cdot rs, j) \end{array}\right\} \quad \text{for} \quad j = 3, \ldots, 5 \tag{4.5.47}$$

where nb_{own} is the number of new own boundary nodes in the current refinement step.

At the beginning of each refinement step we invert the array with the node numbers for each local element, i.e. the *nek* array, in order to get the *nekinv* array with the element numbers each local node belongs to. This array *nekinv* that we need for the neighbour element search is allocated with width equal to the maximum number of elements in_{max} a node belongs to. By the refinement of the elements of one stage the number of elements a node belongs to may change and therefore the maximum number of elements in_{max} any node belongs to must be recomputed at the end of a refinement step. Here ends the discussion of the first refinement step.

For the following refinement steps we have to pay attention to some differences to the first refinement step. We will specify these differences in the following: We increase the total number of nodes n_{rs} and the total number of elements ne_{rs} that existed before the current refinement step by the number of generated new nodes $n_{rs,new}$ (4.5.17) and by the number of generated elements in refinement step rs in order to be able to issue the global node and element numbers in the following refinement step:

$$\begin{array}{rcl} n_{rs+1} & = & n_{rs} + n_{rs,new} \\ ne_{rs+1} & = & ne_{rs} + 3 \cdot ne_{re} \end{array} \tag{4.5.48}$$

because for each refined element three additional elements are created. We also need to update the values for the number of local nodes n_n and elements ne_n including the overlap to issue the local node and element numbers in the following refinement step:

$$\begin{array}{rcl} n_{n,rs+1} & = & n_{new,2} + n_{upd} \\ ne_{n,rs+1} & = & ne_{n,new} + ne_{upd}, \end{array} \tag{4.5.49}$$

see equations (4.5.45) and (4.5.46) as well as figure 4.5.10. The algorithm for one refinement step of the mesh refinement on a distributed memory parallel computer is presented

in algorithm H on page 174 that also contains the peculiarities that occur in 3-D and for dividing lines and sliding dividing lines we describe in the following subsections.

After the refinement of the refinement elements of all refinement stages has taken place, i.e. after all refinement steps, we have to change the numbers of the external boundary nodes from local numbering to global numbering because now we have to sort the grid by x-coordinate again as described on page 45. Furthermore, we have to delete the overlap data for the nodes, elements and external boundary nodes. For the nodes we must do this for the node information array *nnr*, the coordinates, the consistency order and the solution array. For the elements we must eliminate the overlap data from the element information array *nenr*, the *nek* array with the node numbers of the elements and the array *refst* for the refinement stages of the elements. We illustrate this elimination for the *nnr* array in listing 20, for the elements we have to take the numbers stored in the second column of *nbrs*.

```
! initialize node counter
p = 0
! for each refinement stage
do k = 1,rs_max+1
  ! increase p by nr. of new nodes in last ref. stage
  p = p+nbrs(3*k-2,1)
  ! q: starting addr. of new nodes in curr. ref. stage
  q = nbrs(3*k,1)
  ! r: nr. of new nodes in current ref. stage
  r = nbrs(3*k+1,1)
  ! for each column of nnr
  do j = 1,nnrdim
    ! for each new node
    do i = 1,r
      ! copy nnr info
      nnr(p+i,j) = nnr(q+i,j)
    end do
  end do
end do
```

Listing 20: Code for the elimination of the overlap data for the *nnr* array.

In the array *bnod* with the information for the external boundary nodes we additionally have to set

$$bnod(i,1) = nnr(bnod(i,1),1) \quad \text{for each node } i. \tag{4.5.50}$$

which is shown in listing 21. This is because after the nodes have been re-sorted on the processors, we must also re-sort the external boundary nodes, and we can only do this efficiently by sending the array *bnod* in a ring shift to the processors in np communication cycles, and in each cycle the processors take out the boundary nodes they own. Therefore

the boundary nodes must be numbered by their global numbers.

```
! p: nr. of ext. bd. nodes before refinement
p = nbrs(1,3)
! for each old node
do i = 1,p
  ! transform local bd. node numbers into global ones
  bnod(i,1) = nnr(bnod(i,1),1)
end do

! initialize boundary node counter
p = 0
! for each refinement stage
do k = 1,rs_max+1
  ! increase p by nr. of new bd. nodes in last ref. stage
  p = p+nbrs(3*k-2,3)
  ! q: starting addr. of new bd. nodes in curr. ref. stage
  q = nbrs(3*k,3)
  ! r: nr. of new bd. nodes in current ref. stage
  r = nbrs(3*k+1,3)
  ! for each new bd. node
  do i = 1,r
    ! transform local bd. node numbers into global ones
    bnod(p+i,1) = nnr(bnod(q+i,1),1)
    ! copy boundary info
    bnod(p+i,2) = bnod(q+i,2)
  end do
end do
```

Listing 21: Code for the elimination of the overlap data for the *bnod* array.

4.5.2 Extension to 3-D

Important notations:

e_{cnt}: integer array, counter for edges in *etl*
etl: integer array for refinement edge information to be sent to owning processors of neighbour elements of the ref. element
lnpl/
$lnpl_2$: logical array for edge identification
lprocs: logical array for processors *etl* must be sent to
nenr: integer array for element information
nnr: integer array for node information
ptl: integer array for refinement edge information (overlap edges)
rtl: integer array for refinement edge information (local edges)

There is quite a number of differences between 2-D and 3-D on a distributed memory parallel computer. We put up the two arrays for the refinement edge information—*rtl* for the own refinement edges and *ptl* for the refinement edges in the right overlap—as described for 2-D and shown in table 4.4.4 for *rtl*. As in 3-D more than two elements may share an edge, it holds

$$nbm_6 = nnb_{max} + 6 > 7 \tag{4.5.51}$$

where nnb_{max} is the maximum number of neighbour elements per edge of a refinement element in the current refinement step. After putting up the two arrays we pass the *ptl* array to the right as in 2-D and the right overlap processors receive the edges they own and issue the new global node numbers for the new nodes. But there occurs another problem in 3-D, see figure 4.5.11: If two different processors ip_1 and ip_2 own elements that share the same edge that belongs to a right overlap processor ip_3, both processors send their information about this edge to processor ip_3 requesting to generate a mid-point on this edge, but there the information is received in different communication cycles and it is stored in different columns of the *ptl* array. As the same edge is concerned the mid-point must not get two different node numbers. Therefore we must detect identical refinement edges and forbid that the same physical node gets two different node numbers.

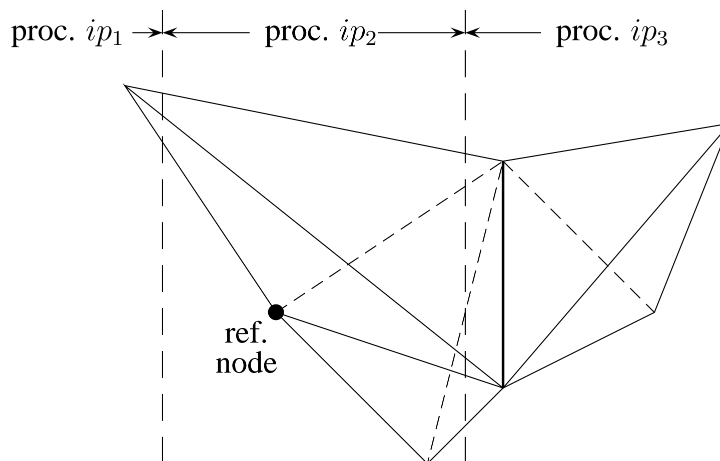


Figure 4.5.11: Illustration of a refinement edge (bold) of which the refinement is caused by two different overlap processors.

So we go through that part of the columns of the refinement edge information array *ptl* the processor received and first transform the global numbers of the end points of an edge into local ones by a binary search in the *nnrs* array, the array with the local node number for each global node that belongs to the processor. We need a **logical array *lnpl*$_2$** that serves the same purpose as the array *lnpl* we used to create the *ptl* and *rtl* array, see on page 91.

This array is initialized by *false*. Again, we have to determine in which neighbour relation the two end points of an edge are and insert the value *true* into $lnpl_2$ if we did not find this edge before. Then the new mid-point must (later) really get a new local node number. If the entry in $lnpl_2$ already has the value *true* for this edge we must find out which of the edges we already regarded is the same one. There we search linearly in the columns 1 to $i-1$ (if the edge occurs in the ith column of *ptl*) and compare the node numbers of the end points of the edges with those of the current edge. When we find the matching edge we store the node number that we issued for the mid-point of this edge also for that one of the current edge. As we do not know at this time the global numbers the new nodes will get, we give them local numbers. Therefore we have a **node counter n_{cnt}** that is initialized by zero and increased by one every time we find a new refinement edge.

The global numbers for the new nodes are issued as described on page 132 for 2-D and then each column of the *ptl* array is sent back to its original processor. The 3-D algorithm for the issuing of the new global node numbers for the mid-points of the edges in the *ptl* array is algorithm F.

Each processor appends the edge information of the *ptl* array at its own *rtl* array and issues the new local node numbers just like in 2-D. Then we check if there are any edges in the *ptl* arrays received from the left that also occur in the own *rtl* array and handle them as described for 2-D on page 137f.

ptl: integer array for edges to be refined by overlap processors

F1 Pass number of edges in each column of *ptl* to the right.

F2 Pass the columns of *ptl* to the corresponding processors.

F3 Receiving processors:

- **a)** Go through all received edge arrays.
- **b)** Issue a new local node number to the mid-point of each edge not yet considered.
- **c)** For an already considered edge search for the matching first occurrence and issue the same number to the mid-point of the current edge.
- **d)** Pass the number of new nodes to the right.
- **e)** Issue the global numbers to the new nodes.
- **f)** Pass the received and completed edge arrays back to the original processors.

Algorithm F: Algorithm for the issuing of new node numbers to the mid-points of the edges in the *ptl* array (3-D).

SUMMARY: If a refinement element has got several neighbour elements that share one of its edges and these neighbour elements are owned by different (overlap) processors and are not refinement elements themselves their owning processors do not get knowledge of the newly-created mid-points of the edge. Therefore we must send the refinement edge information to all processors that own neighbour elements that share an edge of the refinement element after the new node number has been issued.

In 3-D there may be, in contrast to 2-D, more than two elements that share an edge, see figure 4.5.12. So let us look at the following situation: Let there be a processor that wants to refine an element with an edge that is owned by a right neighbour processor. So it puts the edge information into the corresponding column of the array for the overlap refinement edge information, i.e. the *ptl* array, passes this column to the right neighbour processor and gets back the new global node number of the mid-point of the edge which is entered into the *nek* array for the refinement element. But what about the other elements that share this edge? If they are owned by the same processor there is no problem, as the new node number is available and can be entered into the *nek* array for this neighbour element. If they are owned by a different processor and are also refinement elements there is also no problem because then this processor has done the same as the other processor and therefore all necessary information is available. But if an element is not a refinement element and it is not owned by a processor that also owns at least one refinement element that shares the edge, the edge information is missing and we have to make it available.

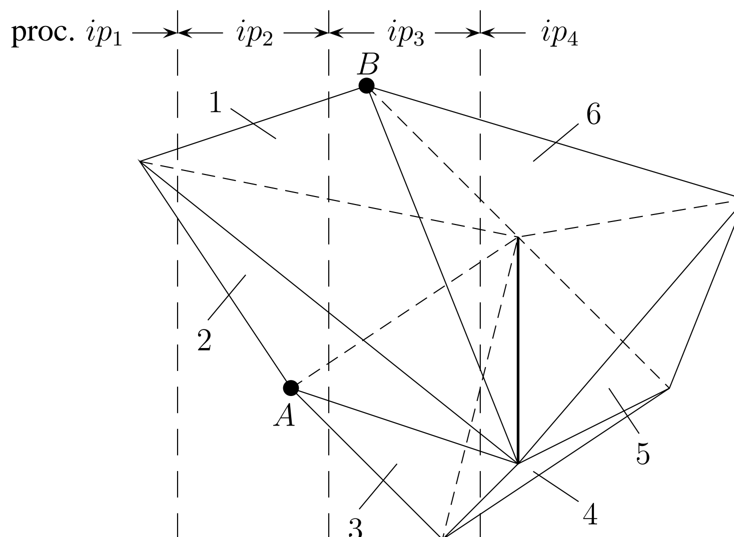


Figure 4.5.12: Illustration of a refinement edge (bold) where the neighbour elements are owned by different overlap processors.

In figure 4.5.12 we depict an exemplary situation. On processor ip_2 the refinement node A is located in the elements 2 and 3. Element 2 is owned by processor ip_1, element 3 by processor ip_2. As the right edge of these elements that is drawn bold in figure 4.5.12 is owned by processor ip_4 both processors must send the refinement edge information to this processor, and afterwards they receive the new global number of the mid-point of the edge. By this, the processors ip_1, ip_2 and ip_4 get knowledge of the new node number and may insert the new node into the elements 1, 2, 3 and 5, but processor ip_3 has not been involved and therefore has no information about the new node. So processor ip_2 has to send the refinement edge information to processor ip_3 after it has received the number of the mid-point from processor ip_4. Then processors ip_3 is capable of inserting the new node into the elements 4 and 6.
If not only node A was a refinement node, but also node B, element 6 on processor ip_3 became a refinement element. Therefore it also has to send the edge information of the bold edge to processor ip_4 and gets back the global node number of the mid-point that can be inserted into the elements 4 and 6. In this example, processor ip_1, ip_2 and ip_3 own refinement elements that share the bold edge, and therefore each of the three processors has got the number of the mid-point that must be inserted into the *nek* array for the elements that contain the refinement edge. Nevertheless, as the processors have not any knowledge of the fact that the other two processors already got the new node number, each of the three processors passes the edge information to the other two processors.

As we have just seen, it is always the processor that owns the refinement element that passes the edge information to the processors that own the neighbour elements that share the refinement edge, not the processor that owns the refinement edge and therefore issues the new node number of the mid-point of the edge. The reason is that these neighbour elements are overlap elements of the processor that owns the refinement element (this is the processor where we have searched and found the neighbour elements, so they must be stored on this processor), but they have not to be in the overlap of the processor that owns the refinement edge so that these elements are completely unknown on this processor and the edge information cannot be passed to the owning processor of the concerned neighbour elements.

To realize this task we go through the columns of the *ptl* array and first determine for all neighbour elements of a refinement element the processor that owns this element. The owning processor ip_{own} of an element is stored in $nenr(el, 4)$ (see on page 78) and as usual, we want to have a processor number relative to ip that gives us the number of the communication cycle in which we have to send the data to processor ip_{own}. So we compute

$$ip_{own} = nenr(el, 4) \tag{4.5.52}$$

and

$$ip_{rel} = ip_{own} - ip. \tag{4.5.53}$$

Although in practical problems it is a rare event, it may happen that some neighbour elements are owned by processors in the left overlap. For example, in figure 4.5.12 processor ip_2 must pass the information for the bold edge to processor ip_3 but also to processor ip_1 although it was not necessary here. Note that the edge information is passed to each processor that owns at least one neighbour element that shares the edge except the processor that issued the new node number of the mid-point as this processor must already have the information. We must proceed like this because the processors work independently and the refinement elements are only stored locally on a processor so that it is not known on a processor if a neighbour element that shares an edge is a refinement element if it is owned by an overlap processor.

If the neighbour element is in the left overlap the result of equation (4.5.53) may become negative. In this case we take the absolute value of ip_{rel} but make clear that the information must be sent to the left. This is done by an integer variable rl. We set

$$\left.\begin{array}{lcl} rl & = & 1 \\ rl & = & 2 \end{array}\right\} \begin{array}{c}\text{if the neighbour}\\ \text{element is in the}\end{array} \left\{\begin{array}{c}\text{right}\\ \text{left}\end{array}\right\} \text{overlap.} \tag{4.5.54}$$

In order to know to which overlap processors we must send any edge information at all, we introduce a **logical array *lprocs*** with length equal to np and width equal to 2 which is initialized by *false*. At the end of this preparatory step we set

$$lprocs(ip_{rel}, rl) = true, \tag{4.5.55}$$

see listing 22.

After having inserted a *true* into *lprocs* for the owning processors of all neighbour elements that share the current refinement edge, all processors where we have to send the edge information are known. For these processors we copy the edge information into the corresponding column of another **integer array *etl*** which is a three-dimensional array with the dimensions $nbm_6 \cdot notp$ (maximum of the total edge information), np_{max} (max. processor) and 2 (send to the right/left). We also introduce an **integer counter array e_{cnt}** with length equal to np_{max} and width equal to 2 where we store the number of edges for each processor and each direction (right/left). So, if the current neighbour element is owned by processor ip_{own}, the data of the current edge k in column ip_{ov} of *ptl* is copied into the *etl* array (see listing 23):

$$\begin{aligned} & etl(nbm_6 \cdot e_{cnt}(ip_{rel}, rl) + i, ip_{rel}, rl) = \\ & \qquad ptl(nbm_6 \cdot (k-1) + i, ip_{ov}) \quad \text{for} \quad i = 1, \ldots, nbm_6. \end{aligned} \tag{4.5.56}$$

Then we continue with the next edge.

```
! saddr1: starting address of current edge in ptl
saddr1 = nbm_6*(k-1)
! noel: number of neighbour elements
noel  = ptl(saddr1+6,i)
! initialize lprocs array
lprocs = .false.

! for each neighbour element
do j = 1,noel
  ! nbel: global number of current neighbour element
  nbel = ptl(saddr1+6+j,i)
  ! nbel: local number of current neighbour element
  nbel = BINSCH(nenrs(1,1),nenrs(1,2),ne_nn,nbel)
  ! ip_rel: relative processor number of owning proc.
  ip_rel = nenr(nbel,4)-myproc
  ! rl = 1/2: proc. in right/left overlap
  rl = 1
  ! check if proc. in left overlap
  if (ip_rel < 0) then
    rl = 2
    ip_rel = -ip_rel
  end if
  ! insert true for proc. into lprocs
  lprocs(ip_rel,rl) = .true.
end do
```

Listing 22: Code for the computation of the array *lprocs*.

After the end of this process we have to pass the number of edges for which we want to send the edge data to the right in $np - 1$ cycles, and this time we also have to do the same for the edge data that has to be sent to the left. We make use of the communication pattern described in subsection 4.1.4 again. After it is known on each processor which length the messages have that it will receive from the left and right, the edge data is exchanged. On processor ip the received messages are stored in *etl* directly behind the data processor ip itself sent in the same cycle, i.e. if it sent the edge data to the second right overlap processor, it will receive the data exactly in the column for the second right overlap processor.

Then we go through the edge data we received from the left and from the right and check with the help of the *lnpl* array, the array where we mark the end points of the edges we already regarded, if the processor itself also wants to halve the edge. There are two possibilities for the given situation on the receiving processor:

1. The edge is in the right overlap and processor ip itself owns a refinement element that contains the edge, i.e. the bisection of the edge is also caused by processor ip. Then the processor has already received the new global node number for the mid-point of

```
! for each direction (right/left)
do rl = 1,2
  ! for each overlap processor
  do ip_rel = 1,np_maxr
    ! check if edge must be sent to current overlap proc.
    if (lprocs(ip_rel,rl)) then
      ! saddr2: starting address of edge in etl
      saddr2 = nbm_6*e_cnt(ip_rel,rl)
      ! copy edge information from ptl to etl
      do i = 1,nbm_6
        etl(saddr2+i,ip_rel,rl) = ptl(saddr1+i,ip_ov)
      end do
      ! increase edge counter for current overlap proc.
      e_cnt(ip_rel,rl) = e_cnt(ip_rel,rl)+1
    end if
  end do
end do
```

Listing 23: Code for the inserting of the data into *etl*.

the edge from the overlap processor that owns and therefore halved the edge, and has inserted all information into the corresponding arrays.

2. The edge is in the overlap but none of the elements that share this edge on processor ip has to be refined. Then the information about the mid-point is not yet known on processor ip and it has to append the edge information to its *rtl* array for the local refinement edge information. Again, the first new local node gets the number

$$nod_{local} = n_n + saddr_2 + 1 \quad (4.5.57)$$

and for each new local node we increase this number by one. The node information of the new mid-point has to be inserted into the node information array *nnr* (see on page 93):

$$\begin{aligned} nnr(nod_{local}, 1) &= etl\,(nbm_6 \cdot (i-1) + 1, ip_{rel}, rl) \\ nnr(nod_{local}, 2) &= nnr(nod_1, 2) \\ nnr(nod_{local}, 3) &= 1 \\ nnr(nod_{local}, 4) &= \min(ip_{nod_1}, ip_{nod_2}) \end{aligned} \quad (4.5.58)$$

where nod_1 and nod_2 are the local numbers of the end points of the edge and ip_{nod_1} and ip_{nod_2} are the corresponding numbers of the owning processors. The owning processor of the new mid-point must be computed by its definition, i.e. if the two end points of the edge are owned by the same processor, this processor will also own the mid-point, and if the end points are owned by different processors the left

processor will own the new mid-point. This is necessary because we do not get the edge information from the owning processor itself but from the processor that owns the refinement element that caused the generation of the mid-point. Additionally, we have to copy the edge information from the *etl* array, the array for the refinement edge information of edges of own neighbour elements of overlap refinement elements that has been received from the corresponding overlap processor, to the array for the local refinement edge information array *rtl*. As the data exchange is only possible by global node and element numbers, we have to transform these global numbers the processor received into local ones. As we have stored r_{cnt} array elements in *rtl* so far, we set

$$rtl\,(r_{cnt} + k) = etl\,(nbm_6 \cdot (i - 1) + k, ip_{rel}, rl) \tag{4.5.59}$$

for $k = 1, \ldots, nbm_6$ if the ith edge in column ip_{rel} of *etl* has to be appended to *rtl*. At the end the counter $saddr_2$ we introduced in (4.5.12) is set to the last local node number on the processor.

The algorithm for passing the edge information to the overlap processors that own the neighbour elements is shown in algorithm G.

RECAPITULATION: As we sort the nodes by x-coordinate it may happen—if we use a huge amount of processors and if the grid has got comparatively few nodes in x-direction—the neighbour elements of a refinement element that share the same edge may be owned by different processors in 3-D. If the edge is owned by the same processor as the refinement element the neighbour processors get knowledge of the new node by the process described on page 143ff. If the edge is owned by a different processor, i.e. if the edge is owned by a right overlap processor, the completed edge information must be sent to all those processors that own neighbour elements except the processor that owns the edge.

Then the global node numbers are issued to those of the new nodes that did not already get one yet. This is done exactly like in 2-D again.

For one refinement step we first want to generate a number of overlap nodes on processor ip and afterwards a number of own nodes, then follow the overlap nodes generated in the next refinement step and so on. We want to have the overlap nodes in front because before the refinement process the last nodes stored on processor ip are overlap nodes, and we want to combine the old and the new overlap nodes of the first refinement step because then we can easier identify own and overlap nodes by the means of array *nbrs*. In 3-D we have the situation that we generated some overlap nodes followed by some own nodes. So far, so good. But when we generated local nodes that only result from a refinement element on an overlap processor as described on page 163ff. these are overlap nodes again, and they are the last ones we generate! As both kinds of overlap nodes—those resulting from

etl: integer array for refinement edge information for overlap processors that own neighbour elements of the local refinement elements

G1 Determine own processors of all neighbour elements of an edge.

G2 Insert edge information in the corresponding columns of the *etl* array.

G3 Pass number of edges in each column of *etl* to the right.

G4 Pass the columns of *etl* to the corresponding processors on the right side.

G5 Pass number of edges in each column of *etl* to the left.

G6 Pass the columns of *etl* to the corresponding processors on the left side.

G7 Receiving processors:

- **a)** Go through all edge arrays received from the left.
- **b)** Check if edge already occurs in own *rtl* array.
- **c)** For an occurring edge search for the occurrence and insert local node number into *rtl* array.
- **d)** For a new edge insert node information into *nnr* array.
- **e)** For a new edge append edge information at *rtl* array.
- **f)** Repeat a) to e) for edge arrays received from the right.

Algorithm G: Algorithm for passing the edge information to overlap processors that own neighbour elements of refinement elements on the sending processor.

refinement edges in the overlap that belong to own refinement elements, and those resulting from the refinement of an element on an overlap processor where processor ip owns neighbour elements that share an edge of the refinemenent element owned by a third (overlap) processor—in a refinement step belong to the right overlap, we rearrange the local nodes in that way that all overlap nodes are in front, followed by the own nodes. Mind you, only the new nodes of the current refinement step are concerned! So we copy the $n_{overl,2}$ overlap nodes from the end of the node information array *nnr* into an auxiliary **integer array *nnr-cop***, shift the first overlap nodes and the own nodes to the end of *nnr* and insert the overlap nodes from *nnrcop* at the beginning of the new nodes at position $n_n + 1$, see figure 4.5.13. By this re-sorting of the nodes we change the local node numbering, so we also have to change the local node numbers in the local refinement edge information array *rtl*.

After the new nodes have got their coordinates and order as shown in (4.4.23) and (4.4.7) we interpolate the solution with consistency order $q = 2$ with the help of the $m = 10$ evaluated influence polynomials of the end points. If only one of the end points is owned

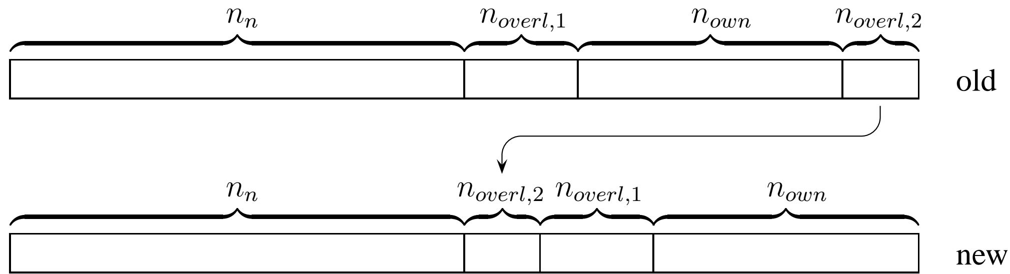


Figure 4.5.13: Illustration of the re-sorting of the new nodes during a refinement step in 3-D.

by the processor this end point solely determines the solution of the new node.

The updating of the *nek* array for the node numbers of the local elements is the same as in 2-D that has been shown in listing 10 for the refinement elements and in listing 11 for the neighbour elements of a refinement element. As an element has got six edges in 3-D, we have to distinguish between six possibilities here, but in principle it is the same as in listing 11. Next we update the *nek* array for those neighbour elements that are owned by an overlap processor as described for 2-D on page 143ff.

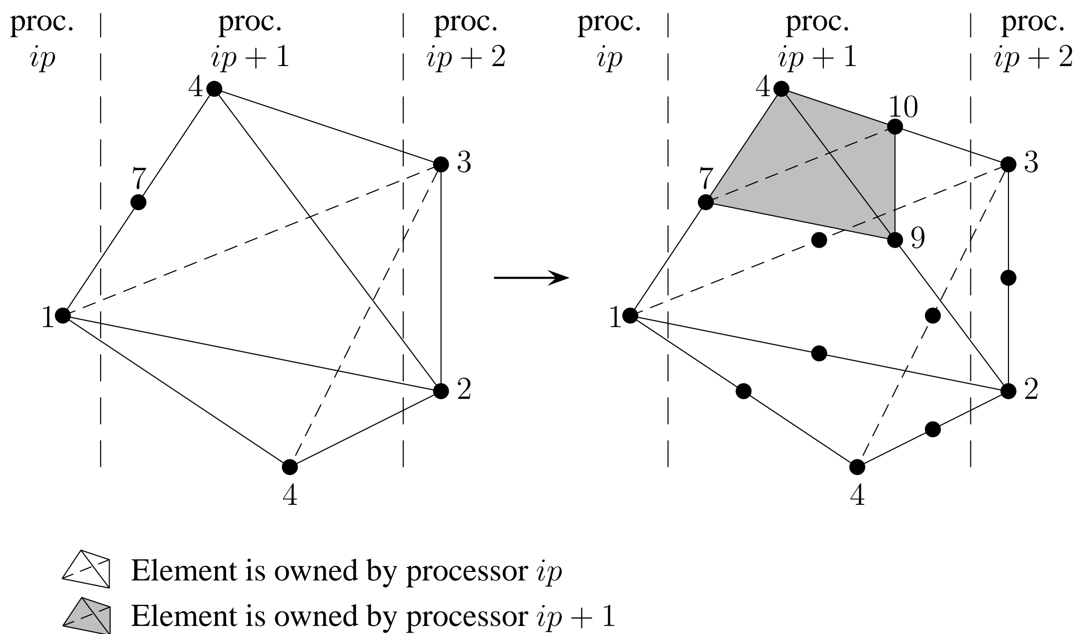


Figure 4.5.14: Illustration of the owning of the new elements in 3-D.

The only difference to 2-D for the generation of the new elements is that we generate eight elements from one old refinement element in 3-D. This has been described for the single

processor run in subsection 4.4.2 on page 107. Like in 2-D, the new elements do not have to be owned by the processor that owns the old refinement element. We illustrate this in figure 4.5.14 where we only show the new elements that “change” the owning processor for reasons of clearness. Again, we consider two elements, the upper element has already a mid-point on edge 7 between node 1 and node 4, the lower element has not any mid-point yet. The mid-points on edge 5 (1–2), edge 6 (1–3) and edge 7 (1–4) of the lower element are generated and owned by processor ip as node 1 is owned by processor ip. As there are not any new elements that have not any of these four nodes (see table 4.4.6), all new elements are owned by processor ip. When we refine the upper element, the nodes 4, 7 and the new mid-points 9 on edge 5 (2–4) and 10 on edge 6 (3–4) are owned by processor $ip+1$, and therefore the new element 4 that consists of these four nodes (see table 4.4.6) is also owned by processor $ip+1$.

Generally one can say how many mid-points must already exist and be owned by processor $ip+1$ before the refinement process to have at least one new element that is owned by processor $ip+1$ subject to the number of corner nodes of the old element owned by processor ip. If there is—like in the example above—only one corner node on processor ip, two of the three mid-points that will be generated and owned by $ip+1$ and a corner node on $ip+1$ will form a new element if one additional mid-point exists. If another corner node is owned by processor ip, we need a second mid-point on processor $ip+1$ existing before the refinement process, too. A general rule that is also valid in 2-D is that the number of mid-points that must exist and be owned by processor $ip+1$ before the refinement is equal to the number of corner nodes owned by processor ip to have at least one new element that is owned by processor $ip+1$.

The determination of the new external boundary nodes is executed purely local, so that the procedure on page 106 is valid here, too.

At the end of a refinement step we have to execute the update step where processor ip has to pass the information what happened to the refinement elements that have been refined because of the refinement cascade to the overlap processors that own the evoking elements. The procedure is exactly the same as in 2-D, but as we have different values for n_{edge}, el_{new} and $nolnod$ the length l_{infarr} of the message that has to be sent to the neighbour processor is computed by

$$l_{infarr} = 124 \cdot ne_s + 1, \tag{4.5.60}$$

which we get from (4.5.38) with the 3-D values

$$n_{edge} = 6, \quad el_{new} = 7, \quad nolnod = 10 \text{ in 3-D.}$$

This means that we have to send 124 integer numbers for each concerned element to the overlap processor. Furthermore, the number $n_{overl,2}$ (see figure 4.5.10) does not only consist of the nodes resulting from the updating of the *nek* array for neighbour elements of

refinement elements owned by the overlap processors, but also of the nodes resulting from algorithm G, i.e. from the refinement of elements on overlap processors where processor ip owns neighbour elements that share an edge of the refinemenent elements owned by a third (overlap) processor. The whole algorithm for one refinement step is shown in algorithm H.

After having refined the refinement elements of all stages, we have to eliminate the overlap data for the nodes, elements and external boundary nodes as described for 2-D before we re-sort the grid onto the np processors.

4.5.3 Mesh refinement with dividing lines in 2-D and 3-D

Important notations:

dlote:	integer array for twin node information of dividing line edges
dloteadr:	integer array for starting addresses of dividing line elements in *dlote*
iglob:	integer array for starting addresses of own ref. edges in *rtl*
lp:	integer array for numbers of edges to receive from overlap processors
ptl:	integer array for refinement edge information (overlap edges)
rtl:	integer array for refinement edge information (local edges)
snod:	integer array for sliding dividing line nodes
tnod:	integer array for dividing line nodes

There is hardly anything left to explain for the mesh refinement for a domain with several subdomains coupled by dividing lines. Most of the differences come from the refinement cascade but in the refinement process itself there are only few differences to the mesh refinement on a single processor. So we mentioned all but one points in subsection 4.4.3. The current subsection holds both for 2-D and 3-D unless otherwise noted.

On page 138 we mentioned the integer array *iglob* where we store the starting addresses of the edges in the local refinement edge array *rtl* of which the mid-points still have to get a new node number. The edges in the array *rtl* have to be examined if they are dividing line edges and thus if the mid-points become dividing line nodes. On a processor we therefore go through the *iglob* array and determine for the edges of which we stored the starting addresses there if the new mid-points become dividing line nodes as described for a single processor in subsection 4.4.3. But we must also examine the edges and mid-points the processor received from the left side in the array for the overlap refinement edges, i.e. the *ptl* array, as these nodes are also owned by the processor.

For this purpose we introduce another one-dimensional **integer array** $\boldsymbol{iglob_2}$. As the maximum number of different edges for which a processor ip received the refinement edge information from the left is the sum of all received numbers of edges over all overlap

processors on the left side which we denote by lp_{sum}, this is the length of the array *iglob*$_2$:

$$lp_{sum} = \sum_{ip=1}^{np_{max,l}} lp(ip). \tag{4.5.61}$$

When we go through the columns of the *ptl* array and insert the received edge information into the *rtl* array, we simultaneously insert the information into *iglob*$_2$ which nodes have to be examined if they are dividing line nodes. This is done without taking into account if an edge already occurs in *rtl* or not which does not mean that the starting address of each edge is inserted into *iglob*$_2$ as we do not want any multiple examination of nodes that have been sent from different processors but nevertheless represent the same physical node in 3-D. The new global node numbers generated on the refinement edges stored in *ptl* are issued by the processor that owns the refinement edge. As several processors may request a new node number for the same edge in 3-D, this processor must avoid multiple numbering of the same node as we explained on page 161f. Therefore we use another **logical array *lpt*** of length $notp$ and width np_{max} that is initialized by *true*. As $notp$ is the number of overlap refinement edges on a processor, there is one entry for each edge in *lpt*. When the issuing processor comes upon a refinement edge that already occurred, the entry for this edge in *lpt* is set to *false*. We insert the starting address of an edge only then into *iglob*$_2$ if the entry in *lpt* is still equal to *true*.

In *iglob*$_2$ we store, analogously to array *iglob*, for each node the starting address of the received edge in *rtl*. We want to determine the new dividing line nodes like we did it for the own nodes but we have not got the information if an element is a dividing line element and which edges of the elements are dividing line edges, as this information is only stored for the own dividing line elements but not for dividing line elements in the overlap. This information must be additionally stored in the *dlote* array, i.e. the array we store the information for each dividing line edge of the dividing line elements. So we have to insert the starting address of the information for the refinement element into the corresponding row of the *dloteadr* array, i.e. the array for the starting addresses of the dividing line element information in *dlote*. For each received edge k that is a dividing line edge, i.e. for the edge holds

$$rtl\,(iglob_2(k) + nbm_6 - 1) = -1, \tag{4.5.62}$$

with nbm_6 from (4.5.1) for 2-D and (4.5.51) for 3-D, respectively, we insert the number of subdomains and the respective twin nodes into the *dlote* array. This is done very similar to the insertion of the edge data during the refinement cascade described on page 72 and shown in listing 7.

nek:	integer array for node numbers of the elements
nekinv:	integer array for elements a node belongs to
ptl:	integer array for edges to be refined by overlap processors
rtl:	integer array for refinement edge information

H1 Create the *nekinv* array by inverting the element array *nek*.

H2 Search for the neighbour elements of the same refinement stage for each edge of the refinement elements.

H3 Insert the necessary edge information into the *rtl* array for the own and into the *ptl* array for the overlap edges.

H4 Issue new node numbers for mid-points of the edges in the *ptl* array (Algorithm D in 2-D, Algorithm F in 3-D).

H5 Append data from own *ptl* array at own *rtl* array.

H6 Append data from received *ptl* arrays at the own *rtl* array.

H7 Only for 3-D: Pass edge information to overlap processors that own neighbour elements of refinement elements on the sending processor (Algorithm G).

H8 Issue new node numbers for mid-points of the edges in the own *rtl* array.

H9 Only for coupled domains with dividing lines: Check if the new nodes are dividing line nodes.

H10 Assign coordinates and order to the new nodes.

H11 Interpolate the solution for the new nodes.

H12 Insert the new nodes into the refinement elements and their neighbour elements in the *nek* array.

H13 Check if the new nodes are external boundary nodes.

H14 Only for coupled domains with sliding dividing lines: Check if the new nodes are sliding dividing line nodes.

H15 Update *nek* array for neighbour elements owned by overlap processors.

H16 Generate four elements in 2-D and eight elements in 3-D out of each old refinement element.

H17 Update element information of elements that have to be refined because of the refinement cascade on overlap processors (Algorithm E).

H18 Prepare the next refinement step.

Algorithm H: Algorithm for one refinement step of the mesh refinement on a distributed memory parallel computer.

After we determined the new dividing line nodes that result from the refinement of the own refinement elements like described on page 115 and shown in listings 14 and 15, we determine the new dividing line nodes that result from the refinement of elements in the left overlap. This is done exactly the same way, we only have to take the edges of which we have stored the starting addresses in $iglob_2$ instead of the data in $iglob$.

For the determination of the new sliding dividing line nodes there is no difference to the single processor run as we do not have the peculiarity that the refinement continues on the other side of the dividing line and we therefore do not have to insert the newly-created nodes into the elements on the other side of the sliding dividing line. Of course, the determination of the new sliding dividing line nodes is always done by the processor that owns the respective new node and not by the one that causes the refinement of the element.

Before we insert the new dividing line nodes into the information array for the dividing line nodes, i.e. the *tnod* array, and the new sliding dividing line nodes into the information array for the sliding dividing line nodes, i.e. the *snod* array, we have to check if we exceeded the limit for the boundary nodes nb_{max}. nb_{max} is a local value and we use the same value for all three kinds of boundary nodes, so the check is the same as for the external boundary nodes we described on page 142: The number of local dividing line nodes including the overlap is nib_n and the number of sliding dividing line nodes is denoted by nsb_n. For dividing line nodes we therefore have to

$$\text{check if } \; nib_n + nib_{new} > nb_{max}$$

and for sliding dividing line nodes

$$\text{check if } \; nsb_n + nsb_{new} > nb_{max}$$

where nib_{new} is the number of new dividing line nodes and nsb_{new} is the number of new sliding dividing line nodes.

The algorithm for the mesh refinement with dividing lines or sliding dividing lines is also included in algorithm H.

4.6 Mesh coarsening

For the solution of parabolic partial differential equations we have a selfcontrolled time grid. Each fully implicit time step is an elliptic problem for that we can refine the mesh. On principle, we may also execute a mesh coarsening.

For some problems in physics or engineering it occurs that the region of the domain where we need finer grid changes in course of time. If we consider a flame that moves through a

pipe we always need a fine grid in the proximity of the flame. But if the flame moves on, we still have the fine grid where the flame is not any more and where we therefore do not need this fine grid any more. So it would be useful if we could undo the mesh refinement and restore the original coarser grid. This process is called mesh coarsening.

We did not (yet) implement mesh coarsening in the FDEM program package but all prerequisites are established by the choice of our data structures. The refinement stage is known for each element and with this information and with the help of the nearest neighbour ring of a node, we are able to reconstruct a coarser grid.

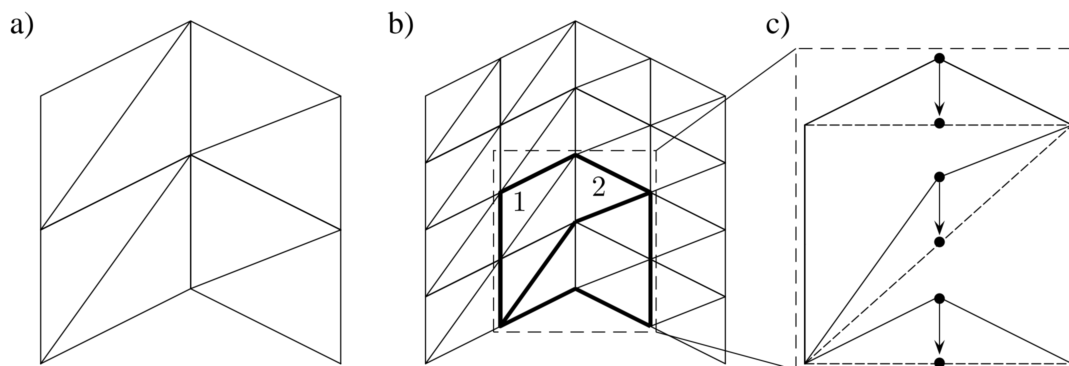


Figure 4.6.1: Illustration of the change of node positions by re-refinement of the mesh: a) original mesh, b) refined mesh with coarsened elements, c) re-refinement of mesh, if the refinement history has not been stored.

If we really wanted to restore the original grid we had to store the grid history, i.e. we had to store the parent element for each new element that we generate by the mesh refinement. Otherwise, the coarsened grid will probably be not identical with the original grid, especially we will not have a triangular grid in 2-D or a tetrahedral grid in 3-D any more, as we probably will combine edges that do not have the same direction, see edges 1 and 2 in figure 4.6.1b). But as we need the elements only for the structure of the space, this is not a serious problem. It is not even a problem to refine these elements again. However, there we eventually change the position of nodes by halving the edges again, see figure 4.6.1. Therefore we consider as useful only a coarsening that reconstructs the parent elements.

4.7 What we have learned from the mesh refinement algorithm

There are two main principles that are so important that we want to mention them once more at the end of this chapter. They are not only valid for the mesh refinement algorithm of the Finite Difference Element Method but can be applied to all kinds of algorithms implemented on distributed memory parallel computers.

The first point concerns the method by which we control the communication so that the processors are in a position to set their sends and receives in a way that we, on the one hand, avoid unnecessary communication and, on the other hand, manage with shorter message lengths and therefore save communication time.

SUMMARY: We want to compare a method to pass the messages to the overlap processors that is easy to implement to our method that is more difficult to implement but very efficient in return. Instead of allocating the send and receive buffers with the same length on each processor that also determines the message length like for the easy method, for our method the buffers and messages have exactly the required length. We do not send a message to each overlap processor (easy method) but only to the processors for which we have data to send to. Therefore we save transfer and also startup time. The preparation of the data transfer is approximately equally expensive for both methods.

When using the message passing model we have some alternatives to parallelize a given code. We want to compare an easy way to control the communication with the way we choose to parallelize the FDEM program package. We explain the differences between the "easy" method and our method examplarily for the passing of the *ptl* array, i.e. the array for the overlap refinement edges (see on page 125), to the right. An easy way to solve this task with the message passing model is to allocate a send buffer by the maximum number of array elements that occurs on the processors, i.e. if we want to pass the *ptl* array to the right, we had to allocate the *ptl* array with length equal to $notp_{max} \cdot nbm_6$ where

$$notp_{max} = \max_{ip=1}^{np} notp_{ip} \tag{4.7.1}$$

holds and $notp_{ip}$ denotes the number of edges that belong to the right overlap. nbm_6 (see (4.5.1)) is the number of entries per refinement edge. With mem_{int} denoting the memory requirement for an integer variable in bytes we had to send a message of length

$$\ell_1 = notp_{max} \cdot nbm_6 \cdot mem_{int} \text{ (bytes)} \tag{4.7.2}$$

to the right overlap processor $ip + i$ in communication cycle i and we have $np_{max,r}$ communication cycles altogether as $np_{max,r}$ is the number of right overlap processors, see subsection 4.1.4. The sum of all message lengths therefore is

$$\begin{aligned} \ell_{total,1} &= \sum_{i=1}^{np_{max,r}} \ell_1 \text{ (bytes)} \\ &= \sum_{i=1}^{np_{max,r}} notp_{max} \cdot nbm_6 \cdot mem_{int} \text{ (bytes)} \\ &= np_{max,r} \cdot notp_{max} \cdot nbm_6 \cdot mem_{int} \text{ (bytes)} \end{aligned} \tag{4.7.3}$$

(as each message has the length ℓ_1, the index i does not occur in the sum), and as we have to send $np_{max,r}$ messages, we need $np_{max,r}$ times the startup time. The preparation of the data exchange consists of the computation of the global maximum of the $notp_{ip}$ values, $notp_{max}$, over the processors. This global integer maximum is computed by subroutine `LL4INM` we described in subsection 4.1.3 and needs $\log_2 np$ reduction steps, i.e. this is the number of communication cycles to determine the maximum that is known on all processors at the end.

On the other hand, with our way of exchanging the data, the array *ptl* is allocated with the length $notp_{ip} \cdot nbm_6$ on processor ip and each message we pass to the right overlap processor $ip + i$ has got the length

$$\ell_2 = p_{cnt,ip}(i) \cdot nbm_6 \cdot mem_{int} \text{ (bytes)} \tag{4.7.4}$$

where

$$\sum_{i=1}^{np_{sr}} p_{cnt,ip}(i) = notp_{ip} \tag{4.7.5}$$

holds. This means that the sum of all message lengths on processor ip is

$$\begin{aligned} \ell_{total,2} &= \sum_{i=1}^{np_{sr}} p_{cnt,ip}(i) \cdot nbm_6 \cdot mem_{int} \text{ (bytes)} \\ &= notp_{ip} \cdot nbm_6 \cdot mem_{int} \text{ (bytes).} \end{aligned} \tag{4.7.6}$$

As we only send exactly that number of refinement edges to a right overlap processor that is owned by this processor, whereas for the easy method we always send the complete column to each right overlap processor, the value $\ell_{total,2}$ is smaller than $\ell_{total,1}$.

The number of communication cycles $np_{real,r}$ is the last cycle for which we have to send any data to the right overlap at all, so

$$np_{real,r} = \max_{i=1}^{np_{max,r}} \left(i \quad \text{with} \quad \max_{ip=1}^{np} p_{cnt,ip}(i) > 0 \right) \tag{4.7.7}$$

holds on processor ip. That means that we search for the relative number i of the rightmost overlap processor $ip + i$ for that the number of refinement edges to send to is still greater than zero on any of the np processors. Therefore we have to add to the time for the pure data exchange $np_{real,r}$ times the startup time. Here the preparation of the data exchange is the passing of the array p_{cnt} to the right which consists of $np_{max,r}$ communication cycles.

Let us compare the cost for the preparation of the data exchange first. In table 4.7.1 you can see for a given number of processors np the number of reduction steps for the easy

method of data exchange in the second column. In the third column the maximum number of processors is given of which we may store the nodes and elements on a processor in order to have the same number of messages to be sent to the right during the preparatory step. Therefore it must hold

$$np_{max,r} = \log_2 np, \tag{4.7.8}$$

and as we assume that the number of overlap processors on the left side is the same as the number of overlap processors on the right side ($np_{max,l} = np_{max,r}$) for a uniform grid, we can store the nodes and elements of $np_{st} = 2 \cdot np + 1$ processors on each processor. In the fourth column we set this number of processors in relation to the total number of processors np to get the percentage of the grid we may store on a processor. The formula for this percentage p is

$$p = \frac{np_{st}}{np} \cdot 100. \tag{4.7.9}$$

If we want to have the same number of messages that have to be sent during the preparatory step for both methods, the number of overlap processors $np_{max,r}$ must be equal to the number of reduction steps. For $np = 2$ processors we cannot have more than one right overlap processor, so there the number of messages is equal. As the number of processors increases it becomes more and more difficult to fulfil the criterion of equal number of messages but for 128 processors we may have 7 right overlap processors, i.e. it holds $np_{max,r} = 7$, which means, as the left overlap usually is as wide as the right one, that we have stored the nodes and elements of $np_{st} = 2 \cdot 7 + 1 = 15$ processors on a processor which is about 12% of the whole grid. From the fourth column of table 4.7.1 we can clearly see that we must store a lesser and lesser part of the grid on a processor the higher we choose the number of processors np we compute on. Only for grids with a very small number of nodes in x-direction in relation to the number of processors the number of overlap processors will not be sufficient to determine the difference stars for each node of the processor but usually, we only choose a higher number of processors for finer grids where we really need them. So we can conclude that the preparatory step for the data exchange is usually equally expensive for both methods.

If we want to compare the cost for the real data exchange we have to make some assumptions before as we must have a relationship between $notp_{ip}$ and $notp_{max}$. We want to regard two different cases. First we want to have a mesh refinement where the whole grid must be refined uniformly, i.e. we have the same number of overlap edges $notp_{ip}$ at each processor border. Then it holds

$$notp_{max} = notp_{ip}. \tag{4.7.10}$$

We denote by γ the ratio of the sum $\ell_{total,2}$ of the message lengths that processor ip must send to its right overlap processors for our own method to the sum $\ell_{total,1}$ of the message lengths for the easy method. So the smaller the ratio γ the more time for the communication

number of processors np	easy method	own method	
	nr. of red. steps $\log_2 np$	nodes/elements max. stored on a proc.	
		nr. of proc. np_{st}	percentage p of grid
2	1	2	100.00
4	2	4	100.00
8	3	7	87.50
16	4	9	56.25
32	5	11	34.38
64	6	13	20.31
128	7	15	11.72
256	8	17	6.64
512	9	19	3.71
1024	10	21	2.05

Table 4.7.1: Comparison of the cost of the preparatory step for the data exchange for the easy and our own method.

we save. It holds for the ratio γ for the considered first case

$$\gamma = \frac{\ell_{total,2}}{\ell_{total,1}} = \frac{notp_{max} \cdot nbm_6 \cdot mem_{int}}{np_{max,r} \cdot notp_{max} \cdot nbm_6 \cdot mem_{int}} = \frac{1}{np_{max,r}}. \qquad (4.7.11)$$

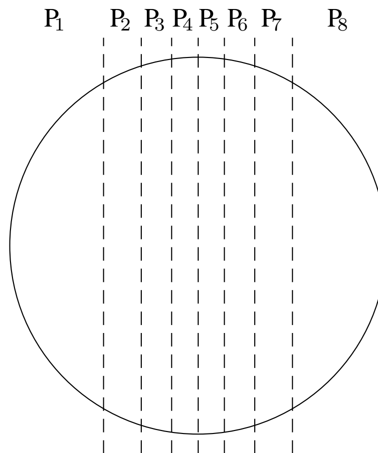


Figure 4.7.1: Illustration of the processor borders and the processor widths for a circular domain for $np = 8$ processors.

For two processors we have $\gamma = 1$ as it holds $np_{max,r} = 1$, and the ratio becomes smaller and smaller when we increase the number of processors np as with the number of proces-

sors the number of overlap processors on the right side $np_{max,r}$ increases, too. It holds

$$\lim_{np\to\infty} \gamma = 0. \tag{4.7.12}$$

And this was even the better one of the two cases we want to regard. If the solution domain is a circle, the number of nodes and therefore the number of edges at a processor border will decrease from the middle of the domain to the left and right side, see figure 4.7.1. Remember that the domain is always separated into slices with the same number of nodes for reasons of load-balancing, and the nodes are always sorted by x-coordinate.

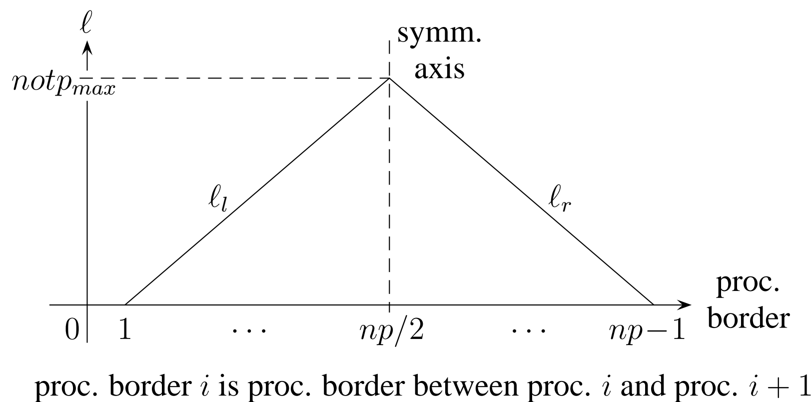


Figure 4.7.2: Comparison of the cost of the preparatory step for the data exchange for the easy and our own method.

If we suppose a solution that looks like a sugar-loaf, see figure 5.3.2, and an even number of processors np, we have the maximum $notp_{max}$ of refinement edges in the overlap in the middle of the grid, between processor $np/2 - 1$ and processor $np/2 + 1$ where the solution has its maximum. At the leftmost and rightmost processor border between processor 1 and 2 and between processor $np - 1$ and np, respectively, we want to have no refinement at all because there the run of the solution is very even, i.e. it holds $notp = 0$. At the processor borders lying in-between we want to have $notp$ to decrease linearly, see figure 4.7.2. The left function ℓ_l is computed by

$$\ell_l(pb) = \frac{notp_{max}}{\frac{np}{2} - 1} \cdot (pb - 1) \tag{4.7.13}$$

where pb is the number of the processor border. In order to get the average number $notp_{avg}$ of refinement edges in the overlap at a processor border, we sum up the just described values of all processor borders and divide the sum by the number of processor borders, which is $np - 1$. As the function ℓ_r is axially symmetric to ℓ_l with regard to the symmetrical axis at $np/2$, we can take twice the function values of ℓ_l (processor borders 1 to $np/2 - 1$) and

leave away the right function ℓ_r. So the average number of edges in the right overlap on a processor, $notp_{avg}$, is computed by

$$\begin{aligned} notp_{avg} &= \frac{notp_{max} + 2 \cdot \sum_{i=1}^{np/2-1} \left(\frac{notp_{max}}{\frac{np}{2}-1} \cdot (i-1) \right)}{1 + 2 \cdot \left(\frac{np}{2} - 1 \right)} \\ &= \frac{notp_{max} \cdot \left(1 + \frac{2}{\frac{np}{2}-1} \cdot \sum_{i=0}^{np/2-2} i \right)}{np-1} \\ &= \frac{notp_{max} \cdot \left(1 + \frac{2}{\frac{np}{2}-1} \cdot \frac{\left(\frac{np}{2}-2\right) \cdot \left(\frac{np}{2}-1\right)}{2} \right)}{np-1} \\ &= notp_{max} \cdot \frac{\frac{np}{2}-1}{np-1} \qquad (4.7.14) \\ &\stackrel{np \to \infty}{\longrightarrow} \frac{notp_{max}}{2} \end{aligned}$$

and therefore the average sum of message lengths on a processor to pass to the right is

$$\ell_{total,3} = notp_{avg} \cdot nbm_6 \cdot mem_{int} \text{ (bytes).} \qquad (4.7.15)$$

So the ratio γ of the sum $\ell_{total,3}$ of the message lengths of our own method to the sum $\ell_{total,1}$ of the message lengths of the easy method for the considered second case is

$$\gamma = \frac{\ell_{total,3}}{\ell_{total,1}} = \frac{notp_{avg} \cdot nbm_6 \cdot mem_{int}}{np_{max,r} \cdot notp_{max} \cdot nbm_6 \cdot mem_{int}} = \frac{1}{2 \cdot np_{max,r}}. \qquad (4.7.16)$$

So we see that the ratio γ for the second case is half the ratio of the first case, and therefore our own method is even better.

Furthermore, we have to add for the easy method of data exchange $np_{max,r}$ times the startup time as we have to pass a message to each of the $np_{max,r}$ overlap processors on the right side. For the method we implemented we only have to pass a message to at most $np_{real,r}$ processors in the right overlap but for practical problems and if we have a grid where the minimum and the maximum of the space step size do not differ much, we have $np_{real,r} = 1$ so that we only have to send a message to the direct right neighbour processor. So the ratio β of the startup times of the easy method to our own method is

$$\beta = \frac{np_{real,r}}{np_{max,r}} = \frac{1}{np_{max,r}}. \qquad (4.7.17)$$

The transfer rate of the interconnection network is not of importance here as it is the same for both data exchange methods. We also do not take into account that the message that is sent to an overlap processor consists of $\ell + \ell_0$ bytes if ℓ is the length of the message we want to send as there will be added a header of ℓ_0 bytes to the message but if we did, the ratio would become even worse. If we assume that the other factors that determine the time needed for the data transfer from one node to another (remote) node—a time to switch a line to a remote node over intermediate hops and a blocking time if the message must wait because one of the intermediate paths is blocked by another transfer—can be neglected, the gain of performance we yield by our method of data exchange in comparison with the simple method is a factor of $np_{max,r}$!

The second important principle is that we do not start the communication between two processors until we have collected in a buffer all data that has to be sent to the target processor. Therefore we must think about the length (and width) of this send buffer at the beginning to be able to allocate it with the right dimension(s) so that we can exchange all necessary data. An alternative was to send in the naive way each single information individually to the target processor as soon as it is known but then the efficiency of the algorithm would suffer from this very much because for each time we start the communication between two processors the startup time must be added to the time the real data transfer takes. And if we only send very short messages the startup time will be greater than the time needed for the data transfer:

$$t_{startup} \geq t_{send}. \tag{4.7.18}$$

For the computers measured in [8] the equality in (4.7.18) was achieved for a message length of at least 256 bytes, on some of the computers for 32 kbytes. So it is clear that the communication time increases excessively if we send many very short messages instead of few long messages where the ratio of startup time to transfer time is rather small.

RECAPITULATION: We learned that it is quite easy to implement a data transfer method for the mesh refinement algorithm but that this method is not efficient at all. We always allocate the communication buffers with the maximum length that is required on any processor, and the message lengths are also the same for the communication between any of the processors. This means that there is also a data transfer with the same message length between two processors although there may be no data to transfer at all only because the target processor is in the overlap of the sending processor. The data transfer is made much more efficient by splitting up the communication into two parts. In the first part we only send numbers of nodes, elements, edges etc. to the overlap processors so that on each processor the send and receive commands can be set in such way that it is known on each processor from which processor it will receive a message and that we may also compute the corresponding message length. We are also able to allocate the length of the receive buffer by the required length so that we save storage

on top. The greater the difference between the longest and the shortest message is, the worse the relation betwwen the data transfer times of the easy method and our own method gets. This is because for the easy method we always send a message with the length that corresponds to the longest message whereas for our method we always send a message that has exactly the required length. As we collect all send data in the send buffers before the communication starts and do not send each piece of information individually to the target processor, the consumed startup time is drastically reduced.

5 Numerical examples

In this chapter we want to give some examples for the mesh refinement. Here we first introduce the test PDE for which we know the exact solution of the system of partial differential equations. Then we explain how we test the Jacobian matrices that we need for the computation of the large sparse matrix of the linear system of equations and for the computation of the error estimate. These matrices are, especially for nonlinear systems of partial differential equations, very fault-prone, so that their check is indispensable and must be executed after having implemented a new system of PDEs. Afterwards we introduce the system of PDEs and the test functions we use for the 2-D examples and give an example for the mesh refinement on a circular domain. The performance of the mesh refinement is illustrated next on different rectangular grids with different numbers of processors before we go to the examples for domains with dividing lines and sliding dividing lines, respectively. Finally, we give an example for a 3-D refinement process on a cubical domain.

5.1 The test PDE

For the test of our program, i.e. both the test of the FDEM solver and the test of the partial differential equations and boundary conditions that have been programmed by the user, we use a partial differential equation of which we know the exact solution. This PDE should have the same properties as the original problem

$$Pu \equiv P(x, y, u, u_x, u_y, u_{xx}, u_{yy}, u_{xy}) = 0 \tag{5.1.1}$$

in 2-D, at least as far as possible. So we prescribe the test solution $\bar{u}(x, y)$ and generate from (5.1.1) a problem that has this solution $\bar{u}$. This problem is the test PDE

$$Pu - P\bar{u} = 0. \tag{5.1.2}$$

$P\bar{u}$ is our problem with the known function $\bar{u}(x, y)$ instead of the unknown function u. Then $P\bar{u}$ is a given function of x and y which is in (5.1.2) an absolute term that contains no variables. For illustration we take the Poisson equation

$$Pu \equiv u_{xx} + u_{yy} = 0 \tag{5.1.3}$$

and prescribe as test solution a polynomial of order 4:

$$\bar{u}(x, y) = \left(x^2 + y^2\right)^2 . \tag{5.1.4}$$

Then the derivatives of this polynomial are

$$\begin{aligned} \bar{u}_x &= 4x^3 + 4xy^2, \\ \bar{u}_{xx} &= 12x^2 + 4y^2, \qquad (5.1.5) \\ \bar{u}_y &= 4x^2y + 4y^3, \\ \bar{u}_{yy} &= 4x^2 + 12y^2. \qquad (5.1.6) \end{aligned}$$

From (5.1.5), (5.1.6) we get

$$\begin{aligned} P\bar{u} = \bar{u}_{xx} + \bar{u}_{yy} &= 12x^2 + 4y^2 + 4x^2 + 12y^2 \\ &= 16x^2 + 16y^2 \end{aligned} \tag{5.1.7}$$

and therefore our test PDE becomes

$$Pu - P\bar{u} \equiv u_{xx} + u_{yy} - \left(16x^2 + 16y^2\right) = 0. \tag{5.1.8}$$

Trivially, if you put for u the function $\bar{u}$ (5.1.4) the PDE is satisfied. We see that $P\bar{u}$ is a pure forcing term that does not have any influence on the part with the variables. Quite naturally we must proceed analogously with the boundary conditions (and the coupling conditions). Let us assume that we have Dirichlet boundary conditions

$$u - f(x, y) = 0 \tag{5.1.9}$$

on the boundary, then the test boundary conditions are

$$\underbrace{u - f(x, y)}_{BC(u)} - \underbrace{(\bar{u}(x, y) - f(x, y))}_{BC(\bar{u})} = 0, \tag{5.1.10}$$

which formally gives

$$u - \bar{u} = 0 \tag{5.1.11}$$

with $\bar{u}$ from (5.1.4).

The test problem (5.1.8), (5.1.11) has the desired solution $\bar{u}$ (5.1.4). As the test solution is a polynomial of order 4 we must get the exact solution $\bar{u}$ and an error estimate in the range of the rounding error if we use a solution method of consistency order $q = 4$. If we use a solution method of consistency order $q = 2$ we get an error that should be well estimated. As the error is estimated by a polynomial of order 4, we should theoretically get an exact error estimate, but the polynomial of order 4 (which would be $\bar{u}$) cannot be reproduced exactly by the solution method of order $q = 2$.

If we want to check if we have programmed our system of partial differential equations correctly, we must program $P\bar{u}$ totally independent of Pu. We start the solution process, i.e. the Newton iteration, for the test problem (5.1.2) with the initial solution $u = \bar{u}$. The first thing to pay attention to is the norm of the right hand side of the linear system of equations (3.1.11). Of course, this norm depends on the partial differential equations but also on the norm of the test solution $\bar{u}$. If the norm of the test solution is 10^9 you cannot expect a norm of the defect in the range of 10^{-12}. So it is quite important that the norm of the test solution is between 1 and 10 in the solution domain. Then the norm of the starting defect $(Pu)_d$ of the Newton iteration is usually in the range of 10^{-11}. The relative norm of

the corrections Δu_{Pu} should be in the same range. There will be only one Newton step and afterwards we have got the discretized solution u_d. Then we compare the estimated relative error $\|\Delta u_d\|_{rel}$ (3.1.14) to the exact error

$$\frac{\|\bar{u} - u_d\|}{\|u_d\|}. \tag{5.1.12}$$

Usually, we choose as test solution $\bar{u}$ a polynomial of order 2, 4, 6 and solve with consistency order $q = 2$, 4, 6. Then the relative error $\|\Delta u_d\|_{rel}$ and the exact error (5.1.12) are usually in the range of the rounding error. To check the error estimate we select a test polynomial of order $4, 6, 8$ and solve with consistency order $q = 2, 4, 6$. Then the exact and estimated error should be nearly equal—if the mesh is fine enough.

Another important question is if the Newton-Raphson method converges. Newton's method converges quadratically if we are close enough to the solution. If not, anything may happen. For this reason we introduced the damped Newton method with a relaxation factor that controls if the Newton defect $(Pu)_d$ decreases in the Newton step, see on page 17. According to our experience, the test PDE gives a good impression how the corresponding physical problem will behave. Therefore we start for the test of the Newton convergence by a disturbed initial solution. We solve the test PDE (5.1.2) with the starting solution $1.01 \cdot \bar{u}$ ($1\,\%$ disturbance) or with $1.1 \cdot \bar{u}$ ($10\,\%$ disturbance) and observe the convergence behaviour. If we cannot solve our PDE for the test solution, we also will not find a solution of the original physical problem.

5.2 The test of the Jacobian matrices

If we solve the test problem with the initial solution $u = \bar{u}$ and the starting defect is in the range of the rounding error but nevertheless, the Newton iteration does not converge, it is quite probable that there is an error in the Jacobian matrices. So we have developed a "Jacobi tester" as the programming of the Jacobian matrices is a dangerous source of errors, especially if we have nonlinear partial differential equations. The Jacobian matrices are used in the generation of the Newton correction Δu_{Pu} as shown in figure 3.1.1 for $\partial Pu/\partial u_x$ (3.1.6). They also enter into the computation of the error as can be seen in (3.1.10). In order to test a Jacobian matrix, here for example $\partial Pu/\partial u_x$, we compute a difference quotient

$$\frac{\Delta Pu}{\Delta u_x} = \frac{P(\ldots, u_x + \varepsilon, \ldots) - Pu}{\varepsilon} \tag{5.2.1}$$

where a typical value for ε is 10^{-4}. Afterwards we check if

$$1 - tol_{jac} \leq \frac{\Delta Pu}{\Delta u_x} / \frac{\partial Pu}{\partial u_x} \leq 1 + tol_{jac} \tag{5.2.2}$$

with a prescribed tolerance tol_{jac} where $\partial Pu/\partial u_x$ is computed in the corresponding subroutine. For nonlinear problems we have to choose a greater value for tol_{jac} than for linear problems because the Jacobian matrices are not constant necessarily but may depend on the derivatives of the solution or on the solution itself. So we choose

$$\begin{aligned} tol_{jac} &= 0.01 \quad \text{for linear problems and} \\ tol_{jac} &= 0.02 \quad \text{for nonlinear problems.} \end{aligned} \tag{5.2.3}$$

If (5.2.2) does not hold we additionally check if

$$\left| \frac{\Delta Pu}{\Delta u_x} - \frac{\partial Pu}{\partial u_x} \right| \leq 10^{-9} \tag{5.2.4}$$

holds because if both values are in the range of the rounding error the quotient in (5.2.2) may be out of the interval $[1 - tol_{jac}, 1 + tol_{jac}]$. If neither (5.2.2) nor (5.2.4) hold, there is an error in the Jacobian matrix and we print out the node. If at least one of the two values to compare is equal to zero we just check if the other one is greater than 10^{-4}. If yes, the Jacobian matrix is false. For a system of m partial differential equations we test the $m \times m$ Jacobian matrices in the same way by components. The term $P\bar{u}$ in the test PDE (5.1.2) does not change the Jacobian matrices because it is an explicit function of x and y (and z in 3-D).

5.3 System of PDEs for the 2-D test examples

For all the following 2-D examples we use a model of the nonlinear Navier-Stokes equations in velocity-vorticity form for the unknown functions u, v and ω with the forcing functions f_i and Reynolds number $Re = 100$:

$$\begin{aligned} u_{xx} + u_{yy} + \omega_y - f_1 &= 0 \\ v_{xx} + v_{yy} - \omega_x - f_2 &= 0 \\ u\omega_x + v\omega_y + (\omega_{xx} + \omega_{yy})/Re - f_3 &= 0 \end{aligned} \tag{5.3.1}$$

under the boundary conditions

$$\begin{aligned} u - g_1 &= 0 \\ v - g_2 &= 0 \\ \omega + u_y - v_x - g_3 &= 0 \end{aligned} \tag{5.3.2}$$

with forcing functions g_i. For the diverse problems that we will introduce in the following sections we use different test functions, and we determine the forcing functions f_i and g_i in such a way that the exact solution $\bar{u}$, $\bar{v}$ and $\bar{\omega}$ is equivalent to the respective test function. This procedure corresponds to the generation of the term $P\bar{u}$ in (5.1.2). In detail, the test functions are

- a polynomial of order $q = 6$

$$\bar{u}(x, y) = x^6 + y^6 = \bar{v}(x, y) = \bar{\omega}(x, y), \tag{5.3.3}$$

see figure 5.3.1.

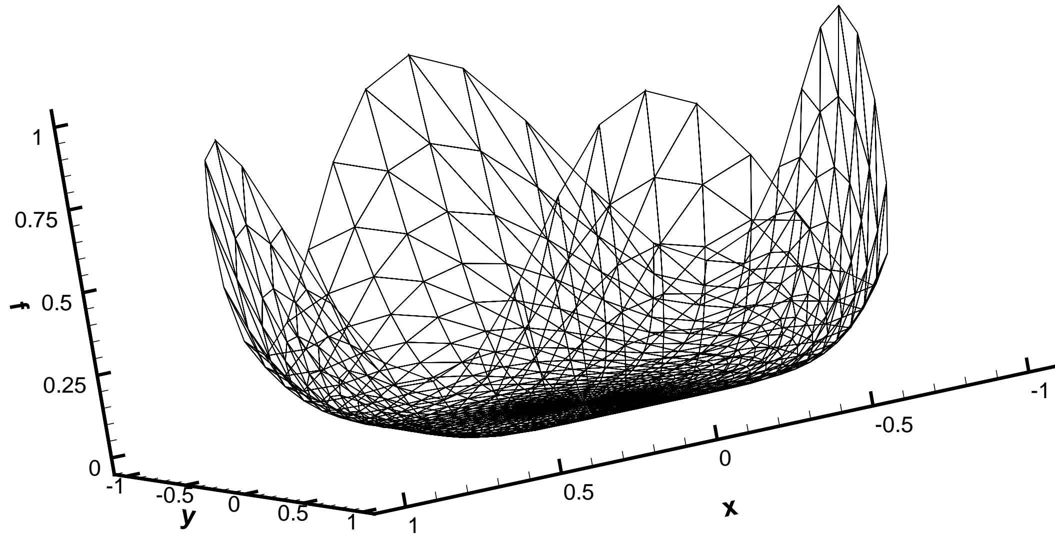


Figure 5.3.1: Polynomial $f = x^6 + y^6$ on unit circle.

- a sugar-loaf-type function

$$\bar{u}(x, y) = e^{-32(x^2+y^2)} = \bar{v}(x, y) = \bar{\omega}(x, y). \tag{5.3.4}$$

This sugar-loaf has a maximum function value $\bar{u} = 1$ in the center of the unit circle and rapid decay to the boundaries and is shown in figure 5.3.2.

- a sugar-loaf-type function

$$\bar{u}(x, y) = e^{-8(x^2+y^2)} = \bar{v}(x, y) = \bar{\omega}(x, y). \tag{5.3.5}$$

This sugar-loaf is similar to the one mentioned before, but it is falling down slower towards the boundary of the domain.

- a constant function (polynomial of order 0)

$$\bar{u}(x, y) = 2 = \bar{v}(x, y) = \bar{\omega}(x, y). \tag{5.3.6}$$

Thus, we can determine the exact (global relative) error (5.1.12). The computer is an IBM SP WinterHawk-II with 375 MHz Power3-2 processors with a peak performance of 1500 MFLOPS. The given CPU time is that for the master processor 1. The number of processors we compute on depends on the respective example and is given there.

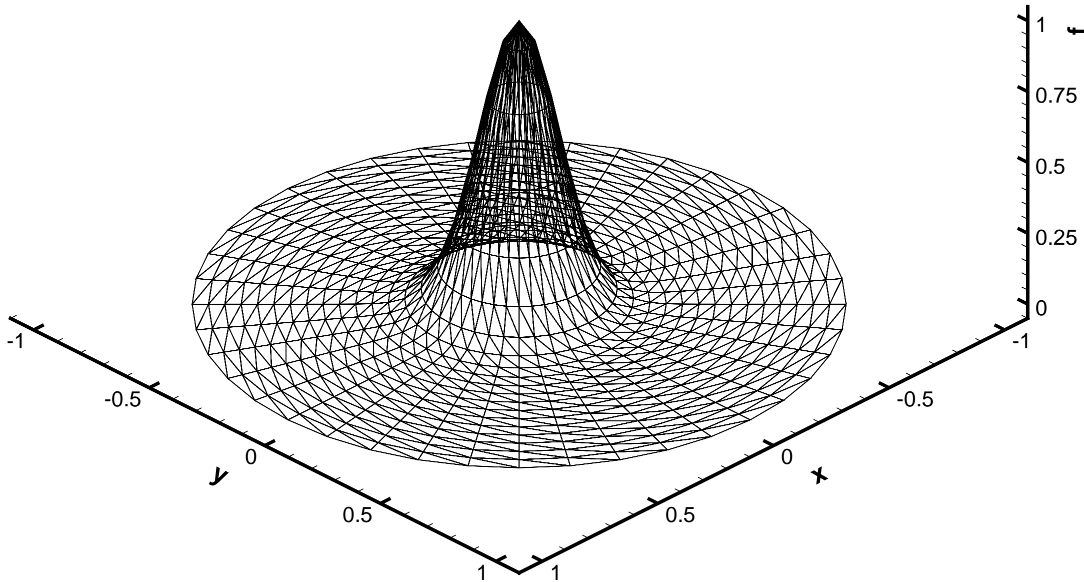


Figure 5.3.2: Sugar-loaf function $f = e^{-32(x^2+y^2)}$ on unit circle.

5.4 Approximating a polynomial by pure mesh refinement

We solve (5.3.1) and (5.3.2) on the unit circle with 751 nodes and $1{,}410$ elements, the grid is generated by the commercial mesh generator I-DEAS. We prescribe a (global relative) tolerance $tol = 10^{-2}$, i.e. $1\,\%$ in the check (3.4.1) and use pure mesh refinement with fixed consistency order $q = 2$ where $\Delta q = 4$ (i.e. we search for nodes up to order 6 for the generation of the difference and error formulas) and $\varepsilon_{pivot} = 10^{-1}$ (see on page 27) holds. As initial solution we choose the exact solution of the problem which is the test solution. We could choose any arbitrary solution but the computation needed more Newton steps then but the result would be the same. We use a polynomial of order 6 (5.3.3) as test function to show that not only the estimated error decreases with finer grid but also the exact error so that we see that the solution really gets better. By solving the basic problem without mesh refinement we have got an estimated error of the component ω of 0.866, so the chosen tolerance tol corresponds to $1.15\,‰$ of the error of the basic solution. We solve the problem on $np = 8$ processors and use our CG solver PRES20 with full LU preconditioning as linear solver. The bandwidth is optimized in each Newton step by the fast single pass bandwidth optimizer introduced in [12]. We need 6 cycles to reach the prescribed tolerance, the results are presented in table 5.4.1.

The maxima of the exact and estimated errors come from the third component ω in all 6 cycles. In cycle 1 we start with the 751 nodes and $1{,}410$ elements of the original grid. By the condition (3.4.15) we get 696 refinement nodes (as there is no refinement cascade

Cycle number	number of nodes	number of elements	number of ref. nodes	number of ref. elem.	comp.	global relative error estimated	global relative error exact	time for cycle [sec]
1	751	1,410	696	1,363	u	$0.616 \cdot 10^{-1}$	$0.584 \cdot 10^{-1}$	3.43
					v	$0.715 \cdot 10^{-1}$	$0.649 \cdot 10^{-1}$	
					ω	0.866	0.738	
2	2,853	5,499	2,278	4,827	u	$0.389 \cdot 10^{-1}$	$0.158 \cdot 10^{-1}$	13.77
					v	$0.446 \cdot 10^{-1}$	$0.166 \cdot 10^{-1}$	
					ω	0.666	0.324	
3	10,290	19,980	5,269	12,770	u	$0.814 \cdot 10^{-2}$	$0.359 \cdot 10^{-2}$	87.22
					v	$0.733 \cdot 10^{-2}$	$0.414 \cdot 10^{-2}$	
					ω	0.236	$0.808 \cdot 10^{-1}$	
4	30,190	58,290	1,451	4,873	u	$0.151 \cdot 10^{-2}$	$0.806 \cdot 10^{-3}$	287.85
					v	$0.155 \cdot 10^{-2}$	$0.792 \cdot 10^{-3}$	
					ω	0.152	$0.242 \cdot 10^{-1}$	
5	38,455	72,909	1,396	5,315	u	$0.209 \cdot 10^{-3}$	$0.199 \cdot 10^{-3}$	418.35
					v	$0.366 \cdot 10^{-3}$	$0.226 \cdot 10^{-3}$	
					ω	$0.378 \cdot 10^{-1}$	$0.854 \cdot 10^{-2}$	
6	47,900	88,854	—	—	u	$0.115 \cdot 10^{-3}$	$0.144 \cdot 10^{-3}$	574.55
					v	$0.257 \cdot 10^{-3}$	$0.233 \cdot 10^{-3}$	
					ω	$0.925 \cdot 10^{-2}$	$0.642 \cdot 10^{-2}$	

Table 5.4.1: Results for pure mesh refinement on unit circle with order $q = 2$.

in the first cycle yet). Then between cycle 1 and cycle 2 the triangles that contain the 696 refinement nodes are refined by which there result totally $2{,}853$ nodes and $5{,}499$ elements that we then redistribute on the 8 processors according to the x-coordinate with the overlap as described in section 3.6. In the cycles 2 and 3 the number of refinement nodes increases, here we have refinement nodes not only because of condition (3.4.15) but also because of the refinement cascade. At the same time the exact and the estimated error decrease. In the cycles 4 and 5 the number of refinement nodes falls off to about $1{,}451$ or $1{,}396$, respectively, and the exact and estimated error continue decreasing. Here we only refine elements of the two highest refinement stages, i.e. the “old” elements of the first two or three stages, respectively, are not changed any more, only in those regions where we do not yet meet the prescribed tolerance we must refine the mesh. This leads for cycle 6 to $47{,}900$ nodes and $88{,}854$ elements. Then there are not found any refinement nodes any more as the estimated relative errors are below the requested tolerance tol.

In figure 5.4.1 we show solution component ω on the unit circle for the first cycle (figure 5.4.1a)) and for the last cycle (figure 5.4.1b)). In the first cycle we see that the solution is not symmetrical as expected (by the unsymmetry of the grid) which explains the high level of the exact and estimated error. In the sixth cycle we can see that the solution has got the expected appearance and therefore the errors are small and the prescribed tolerance is met.

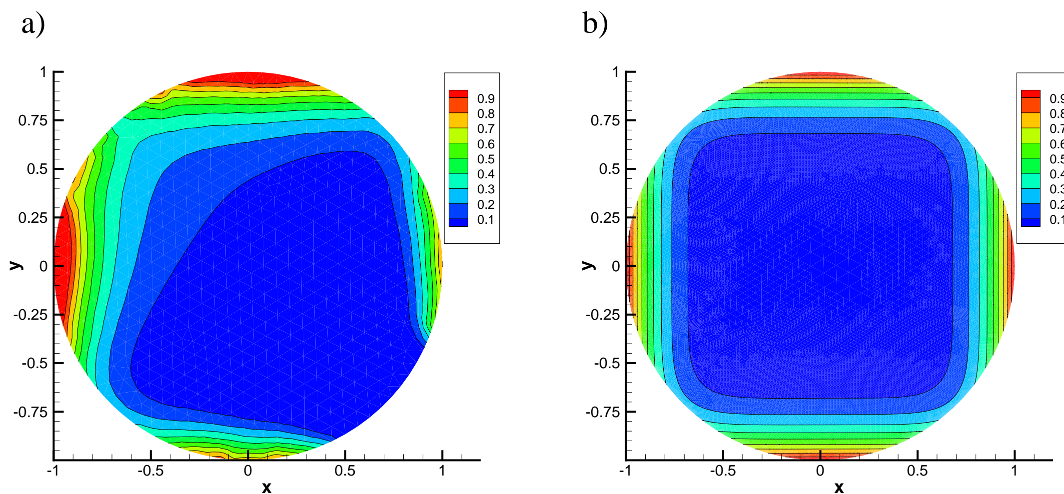


Figure 5.4.1: Solution of component ω in the a) first and b) sixth computation cycle.

The average edge length of the original grid is

$$h_{avg,1} = 0.717 \cdot 10^{-1},$$

in the last cycle we have an average edge length of

$$h_{avg,6} = 0.733 \cdot 10^{-2}$$

which is about 10% of the original edge length so that we can expect in the last cycle an error of about 1% of the error of the basic solution because the error should go down quadratically with the space step size. As the errors for ω are 0.866 for cycle 1 and $0.925 \cdot 10^{-2}$ for cycle 6, we see that the error really fulfills this guess. The CPU time for a cycle clearly increases with the increasing number of nodes and elements. The total time for the 6 cycles is $1{,}385.35$ seconds.

In figures 5.4.2a) to f) on pages 194 and 195 we see the 6 grids on the left side and the visualization of the estimated relative error for component ω on the right side. In figure 5.4.2f), the grid that finally meets the requested tolerance, you can still see the coarsest grid of cycle 1 around the origin. The central region of the grid does not change any more between the third and the sixth cycle as the prescribed tolerance is met. This comes from the fact that the function values of the test function do not change very much in this region. The maxima of the estimated errors are at the boundary of the domain in each cycle, but with decreasing level.

Cycle number	1	2	3	4	5	6
Grid sorting[1]	0.04	0.10	0.25	0.59	0.77	0.85
Point collection[1]	0.12	0.18	0.59	2.69	3.79	5.42
Formula generation[1]	0.03	0.10	0.41	1.39	1.74	2.07
Newton iteration[1]	2.87	12.97	85.49	282.56	411.27	565.85
number of Newton steps	2	2	2	1	1	1
LINSOL[1]	2.85	12.90	85.29	282.24	410.85	565.29
Bandwidth optimizer[1]	0.14	0.82	6.11	18.50	28.76	42.00
Bandwidth	222	508	1,301	3,088	4,072	5,048
Factorization[1]	2.45	11.30	73.02	235.54	340.46	465.60
Newton iteration without LINSOL[1]	0.02	0.07	0.20	0.32	0.42	0.56
Mesh refinement[1]	0.20	0.35	0.39	0.40	0.46	—
FDEM without LINSOL[1]	0.58	0.87	1.93	5.61	7.50	9.26

[1] CPU time in seconds

Table 5.4.2: Time analysis for pure mesh refinement with order $q = 2$ (part one).

When we have a look at the CPU times needed by the single parts of FDEM in the 6 cycles shown in table 5.4.2 we see—if we divide the CPU time by the number of nodes in the cycle, see table 5.4.3—that the increase in time does not come from the FDEM parts such

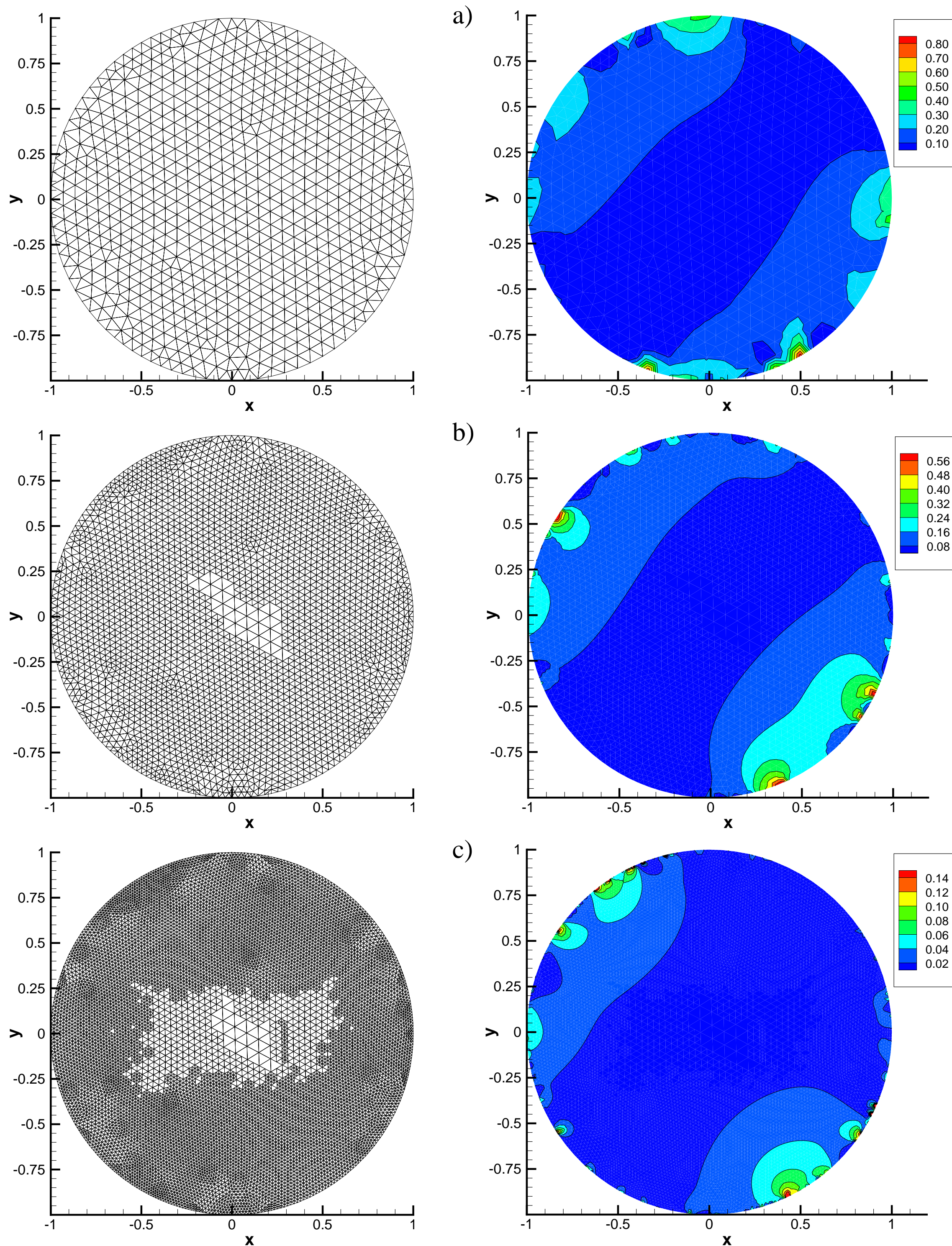


Figure 5.4.2: Resulting grids and estimated errors computed on these grids from pure mesh refinement: a) original grid, b) grid in cycle 2, c) grid in cycle 3.

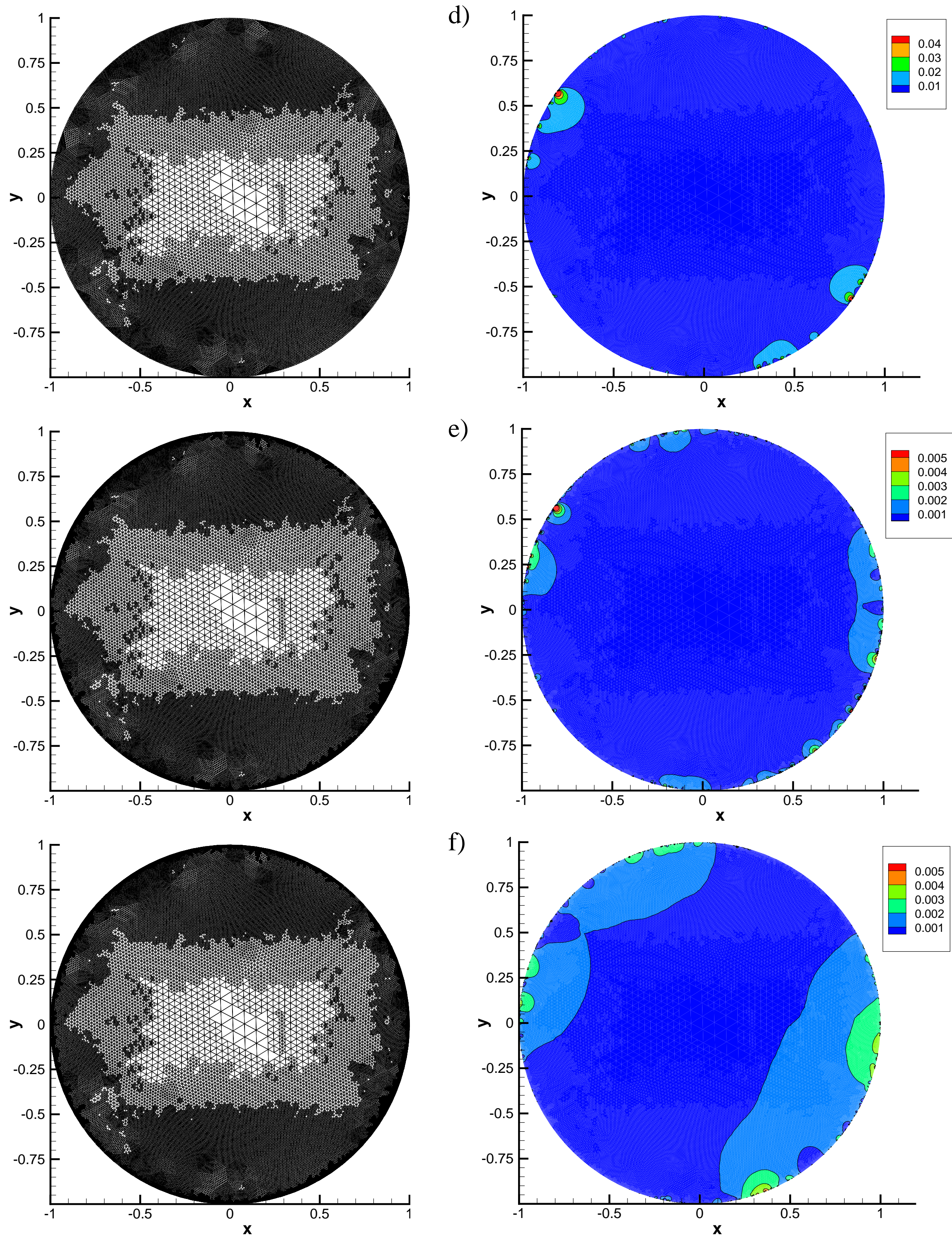


Figure 5.4.2: (Continued) Resulting grids and estimated errors computed on these grids from pure mesh refinement: d) grid in cycle 4, e) grid in cycle 5, f) grid in cycle 6.

as grid sorting, point collection, generation of the difference and error formulas and mesh refinement, but from the linear solver LINSOL that solves the large sparse linear system of equations during each Newton step. This comes from the fact that the bandwidth of the matrix Q_d increases from one cycle to the other, and therefore the time for the factorization increases much more than we expected from the number of unknowns.

Cycle number	1	2	3	4	5	6
Grid sorting[1]	5.3E-5	3.5E-5	2.4E-5	2.0E-5	2.0E-5	1.8E-5
Point collection[1]	1.6E-4	6.3E-5	5.7E-5	8.9E-5	9.9E-5	1.1E-4
Formula generation[1]	4.0E-5	3.5E-5	4.0E-5	4.6E-5	4.5E-5	4.3E-5
Newton iteration[1]	3.8E-3	4.5E-3	8.3E-3	9.4E-3	1.1E-2	1.2E-2
LINSOL[1]	3.8E-3	4.5E-3	8.3E-3	9.3E-3	1.1E-2	1.2E-2
Newton it. without LINSOL[1]	2.7E-5	2.5E-5	1.9E-5	1.1E-5	1.1E-5	1.2E-5
Mesh refinement[1]	2.7E-4	1.2E-4	3.8E-5	1.3E-5	1.2E-5	—
FDEM without LINSOL[1]	7.7E-4	3.0E-4	1.9E-4	1.9E-4	2.0E-4	1.9E-4

[1] CPU time in seconds divided by number of nodes

Table 5.4.3: Time analysis for pure mesh refinement with order $q = 2$ (part two).

5.5 Experiments with rectangular grids

We first compare a complete FDEM computation with all substages on several rectangular grids where we want to have the same number of nodes per processor for each computation. Then we examine the mesh refinement itself for different grids.

5.5.1 Investigating all steps of FDEM

In order to see the influence of the grid we make a series computation with 7 different grids for the solution of (5.3.1) and (5.3.2) for the test function (5.3.4) (sugar loaf with top in the middle of the domain) on a 4×1 domain with the grid type of figure 5.5.1. The characteristics of the 7 grids are shown in table 5.5.1.

The number of grid points is approximately doubled from one grid to the other which is not possible to do exactly as we had to multiply the number of points in x- and y-direction by $\sqrt{2}$, but these numbers are doubled in every second grid. This also results in the doubled number of the elements and unknowns. After the first cycle we refine the mesh completely, i.e. we set the tolerance $tol = 0$ so that all elements become refinement elements. Again, we compute on the IBM SP WinterHawk-II as above, but now we also double the number

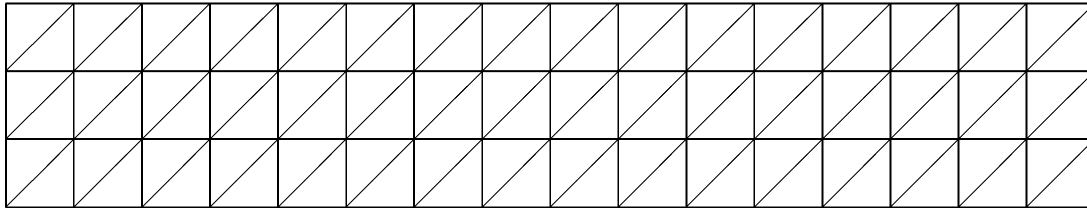


Figure 5.5.1: Type of grid for 4×1 domain.

of processors when we double the number of nodes. Thus, the number of nodes, elements and unknowns per processor is the same for each computation. We use difference formulas of consistency order $q = 2$. This time, we use for the computations the BiCGStab2 method as iterative solver, a smoothed biconjugate gradient method, see [13]. We are only interested in the exact and estimated relative errors and in the CPU time (of processor 1).

From table 5.5.2 we can see that the relative exact error of the second cycle goes down to 25% of the relative exact error of the first cycle for each of the seven grids. As each

Grid number	Cycle number	Dimension	Number of nodes	Number of elements	Number of unknowns	No. of proc.
1	1	128×32	4,096	7,874	12,288	1
	2	255×63	16,065	31,496	48,195	
2	1	181×45	8,145	15,840	24,435	2
	2	361×89	32,129	63,360	96,387	
3	1	256×64	16,384	32,130	49,152	4
	2	511×127	64,897	128,520	194,691	
4	1	362×90	32,580	64,258	97,740	8
	2	723×179	129,417	257,032	388,251	
5	1	512×128	65,536	129,794	196,608	16
	2	$1{,}023 \times 255$	260,865	519,176	782,595	
6	1	724×181	131,044	260,280	393,132	32
	2	$1{,}447 \times 361$	523,367	1,041,120	1,567,101	
7	1	$1{,}024 \times 256$	262,144	521,730	786,432	64
	2	$2{,}047 \times 511$	1,046,017	2,086,920	3,138,051	

Table 5.5.1: Characteristics of the 7 grids for the complete computation. After cycle 1 all elements are refined.

edge of the grid is halved during the mesh refinement process this is what we expected for consistency order $q = 2$. Furthermore, the errors get halved for two subsequent grids, both for the first and for the second cycle, as the number of nodes is increased by a factor $\sqrt{2}$, i.e. the space step size is reduced by a factor $\sqrt{2}$. The comparison of the estimated and the exact error shows that the error estimation gets better for finer grid. Both the exact and the estimated error go down from one grid to the other (finer) grid. In the second cycle the refined grid i is almost the same grid as the original grid $i + 2$, and we see that the relative errors are very similar. Usually in investigations of discretization methods the exact and the computed solution are compared, i.e. the quality of the computed solution. However, here we compare the exact and the estimated error, i.e. the quality of the error estimate. This is a quality control one level higher as usual and it gives you the indispensable information if you can "trust" your solution. And this is for a "black-box" where neither the PDEs nor BCs nor the domain that the user puts in are known to the designer of the program. This is in complete contrast to commercial codes that are always specialized to a certain type of problem.

Grid number	Cycle number	comp.	global relative error estimated	global relative error exact	CPU [sec]	CPU ref. [sec]
1	1	u	$0.792 \cdot 10^{-2}$	$0.648 \cdot 10^{-2}$	6.23	0.36
		v	$0.794 \cdot 10^{-2}$	$0.810 \cdot 10^{-2}$		
		ω	$0.786 \cdot 10^{-2}$	$0.136 \cdot 10^{-1}$		
	2	u	$0.195 \cdot 10^{-2}$	$0.136 \cdot 10^{-2}$	54.87	
		v	$0.202 \cdot 10^{-2}$	$0.204 \cdot 10^{-2}$		
		ω	$0.200 \cdot 10^{-2}$	$0.254 \cdot 10^{-2}$		
2	1	u	$0.391 \cdot 10^{-2}$	$0.289 \cdot 10^{-2}$	8.36	0.36
		v	$0.400 \cdot 10^{-2}$	$0.406 \cdot 10^{-2}$		
		ω	$0.397 \cdot 10^{-2}$	$0.556 \cdot 10^{-2}$		
	2	u	$0.974 \cdot 10^{-3}$	$0.644 \cdot 10^{-3}$	72.99	
		v	$0.100 \cdot 10^{-2}$	$0.101 \cdot 10^{-2}$		
		ω	$0.995 \cdot 10^{-3}$	$0.117 \cdot 10^{-2}$		
3	1	u	$0.192 \cdot 10^{-2}$	$0.132 \cdot 10^{-2}$	13.02	0.36
		v	$0.198 \cdot 10^{-2}$	$0.200 \cdot 10^{-2}$		
		ω	$0.197 \cdot 10^{-2}$	$0.245 \cdot 10^{-2}$		
	2	u	$0.481 \cdot 10^{-3}$	$0.305 \cdot 10^{-3}$	107.00	
		v	$0.494 \cdot 10^{-3}$	$0.499 \cdot 10^{-3}$		
		ω	$0.491 \cdot 10^{-3}$	$0.505 \cdot 10^{-3}$		

Continued on next page

Grid number	Cycle number	comp.	global relative error estimated	exact	CPU [sec]	CPU ref. [sec]
4	1	u	$0.962 \cdot 10^{-3}$	$0.632 \cdot 10^{-3}$	16.16	0.40
		v	$0.988 \cdot 10^{-3}$	$0.999 \cdot 10^{-3}$		
		ω	$0.983 \cdot 10^{-3}$	$0.115 \cdot 10^{-2}$		
	2	u	$0.240 \cdot 10^{-3}$	$0.150 \cdot 10^{-3}$	140.52	
		v	$0.247 \cdot 10^{-3}$	$0.250 \cdot 10^{-3}$		
		ω	$0.246 \cdot 10^{-3}$	$0.265 \cdot 10^{-3}$		
5	1	u	$0.476 \cdot 10^{-3}$	$0.300 \cdot 10^{-3}$	33.03	0.39
		v	$0.490 \cdot 10^{-3}$	$0.494 \cdot 10^{-3}$		
		ω	$0.487 \cdot 10^{-3}$	$0.504 \cdot 10^{-3}$		
	2	u	$0.119 \cdot 10^{-3}$	$0.725 \cdot 10^{-4}$	196.63	
		v	$0.122 \cdot 10^{-3}$	$0.124 \cdot 10^{-3}$		
		ω	$0.122 \cdot 10^{-3}$	$0.128 \cdot 10^{-3}$		
6	1	u	$0.222 \cdot 10^{-3}$	$0.146 \cdot 10^{-3}$	40.32	0.40
		v	$0.246 \cdot 10^{-3}$	$0.247 \cdot 10^{-3}$		
		ω	$0.252 \cdot 10^{-3}$	$0.258 \cdot 10^{-3}$		
	2	u	$0.540 \cdot 10^{-4}$	$0.362 \cdot 10^{-4}$	332.61	
		v	$0.617 \cdot 10^{-4}$	$0.616 \cdot 10^{-4}$		
		ω	$0.631 \cdot 10^{-4}$	$0.634 \cdot 10^{-4}$		
7	1	u	$0.119 \cdot 10^{-3}$	$0.721 \cdot 10^{-4}$	56.38	0.43
		v	$0.122 \cdot 10^{-3}$	$0.123 \cdot 10^{-3}$		
		ω	$0.121 \cdot 10^{-3}$	$0.127 \cdot 10^{-3}$		
	2	u	$0.259 \cdot 10^{-4}$	$0.182 \cdot 10^{-4}$	518.91	
		v	$0.309 \cdot 10^{-4}$	$0.308 \cdot 10^{-4}$		
		ω	$0.318 \cdot 10^{-4}$	$0.310 \cdot 10^{-4}$		

Table 5.5.2: Results for seven different grids with sugar-loaf forcing function. CPU ref. is the CPU time for the refinement only.

As the number of processors is quadruplicated the time for the computation of the solution and of the error of cycle 1 of grid $i + 2$ should be about one quarter of the time for cycle 2 of grid i (same number of unknowns). The more processors we have the worse the ratio gets, this comes from the additional communication overhead we get if we compute the same problem on more and more processors. If we compare the CPU times for the second and the first cycle for the same grid, we get a time ratio between 8.2 and 9.2 instead of the expected ratio of 4. From table 5.5.3 we learn that this overhead comes from two parts of FDEM: the point collection and the Newton iteration. The reason for the time

Grid number	1		2		3		4	
Cycle number	1	2	1	2	1	2	1	2
Grid sorting[1]	0.00	0.00	0.23	0.94	0.33	1.36	0.46	2.14
Point collection[1]	1.98	27.72	1.73	22.00	2.08	25.25	2.27	27.17
Formula generation[1]	0.71	2.76	0.71	2.80	0.78	2.91	0.76	2.76
Newton iteration[1]	2.97	24.21	5.17	47.10	9.27	77.24	12.08	107.98
LINSOL[1]	2.27	20.26	4.52	43.45	8.55	73.55	11.36	104.24
Newton it. without LINSOL[1]	0.70	3.95	0.65	3.65	0.72	3.69	0.72	3.74
Mesh refinement[1]	0.36	—	0.36	—	0.36	—	0.40	—
Refinement cascade[1]	0.09	—	0.08	—	0.09	—	0.09	—
Refinement of elements[1]	0.27	—	0.28	—	0.27	—	0.31	—
FDEM without LINSOL[1]	3.96	34.61	3.84	29.54	4.47	33.45	4.80	36.28

Grid number	5		6		7	
Cycle number	1	2	1	2	1	2
Grid sorting[1]	0.89	3.58	1.60	6.32	3.58	14.05
Point collection[1]	2.70	30.73	3.34	35.42	4.23	42.50
Formula generation[1]	0.75	2.80	0.77	2.81	0.77	2.86
Newton iteration[1]	28.04	158.82	33.88	286.95	46.99	458.63
LINSOL[1]	27.23	154.85	32.95	282.78	45.97	453.81
Newton it. without LINSOL[1]	0.81	3.97	0.93	4.17	1.02	4.82
Mesh refinement[1]	0.39	—	0.40	—	0.43	—
Refinement cascade[1]	0.08	—	0.10	—	0.09	—
Refinement of elements[1]	0.31	—	0.30	—	0.34	—
FDEM without LINSOL[1]	5.80	41.78	7.37	49.83	10.41	65.10

[1] CPU time in seconds

Table 5.5.3: Time analysis for pure mesh refinement with order $q = 2$.

increase is that the data comes from the cache in the first cycle and from the memory in the second cycle as the L2-cache has a capacity of 8 MB. For the point collection this data is the logical array we use for the determination of the next neighbour ring and for the Newton iteration it is mainly the array(s) for the large sparse matrix Q_d. Furthermore, the condition of the matrix will become worse if the number of unknowns increases, so that the CG method needs more matrix vector multiplications (MVMs). The generation of the difference and error formulas and the mesh refinement scale very well as the time is more or less constant for the seven computations. The time for the mesh refinement is only about 6% of the time for the first cycle on a single processor, and decreases to 0.8% for the computation on 64 processors. When we re-sort the nodes onto the processors, only up to $np/2$ processors are active in parallel. Furthermore, the elements are sent around in a ring shift. The more processors we use, the more communication tacts we need. This explains why the time for the grid sorting approximately doubles from one grid to the other. For grid 5 we have a time ratio of 6.0 between the two cycles which comes from the fact that the time for the Newton iteration is mainly determined by the number of MVMs, and the number of MVMs usually increases with the number of unknowns, but for grid 5 we have got a considerably smaller number of MVMs, i.e. it falls out of line. We have chosen the CG method BiCGStab2 without LU preconditioning because we will get much fill-in by the preconditioning involving storage problems. On the other hand, the CG method will fail if we strongly increase the number of unknowns. For grid 4 we made the same computation that we made with the BiCGStab2 method with LU preconditioning on $np = 8$ processors. The CPU time for this computation is $1{,}616$ seconds in comparison with 157 seconds for the computation with BiCGStab2 without preconditioning.

5.5.2 Scaling of the mesh refinement only

As the CPU time for the mesh refinement is quite small for the examples in the last subsection, we now want to show how the CPU time behaves for grids with more nodes and elements. As the memory of one node of the IBM SP is only 2 GB, we cannot execute the complete computation. On the other hand we need the influence polynomials for each node to be able to interpolate the solution of the new nodes. So we execute the point collection and the generation of the difference and error formulas, omit the Newton iteration and jump directly to the mesh refinement. Afterwards we stop the computation.

Again, we make a series computation with 7 different grids on a 4×1 domain with the grid type of figure 5.5.1. The characteristics of the 7 grids are shown in table 5.5.4. We approximately double the number of grid points from one grid to the other which is not possible to do exactly for the same reason as above. To be able to compare the measured CPU times, we refine the mesh completely, i.e. we set the tolerance $tol = 0$ so that all elements become refinement elements. Again, we compute on the IBM SP WinterHawk-II as above and we also double the number of processors when we double the number of nodes so that the

Grid nr.	Dimension	Original grid Number of nodes	Number of elements	Refined grid Number of nodes	Number of elements	No. of proc.
1	724 × 181	131,044	260,280	522,367	1,041,120	1
2	1,024 × 256	262,144	521,730	1,046,017	2,086,920	2
3	1,448 × 362	524,176	1,044,734	2,093,085	4,178,936	4
4	2,048 × 512	1,048,576	2,092,034	4,189,185	8,368,136	8
5	2,896 × 724	2,096,704	4,186,170	8,379,577	16,744,680	16
6	4,096 × 1,024	4,194,304	8,378,370	16,766,977	33,513,480	32
7	5,792 × 1,448	8,386,816	16,759,154	33,532,785	67,036,616	64

Table 5.5.4: Characteristics of the 7 grids for the mesh refinement.

number of nodes, elements and unknowns on a processor is the same for each computation. We use difference formulas of consistency order $q = 2$. We are only interested in the CPU time (of processor 1) for the mesh refinement which is shown in table 5.5.5.

Grid number	CPU time[1] for determ. of ref. elem.	mesh refinement
1	1.10	6.18
2	1.07	6.16
3	1.08	6.14
4	1.14	6.56
5	1.10	6.53
6	1.16	6.75
7	1.20	6.77

[1] in seconds

Table 5.5.5: CPU time for the mesh refinement for the seven grids.

One can see in this table that the CPU time for the determination of the refinement elements is between 1.07 and 1.20 seconds for each computation so that we can say that this part of the mesh refinement algorithm is scaling very well, but this is no wonder as there is not any communication in the first refinement cycle. All elements have the same refinement stage so that there the difference of the refinement stages of two neighbour elements is at most 1

after the refinement and therefore we do not have to execute the refinement cascade here. As the scaling property of the mesh refinement can only be tested if we refine the whole mesh in a refinement cycle, we cannot test the scaling of the refinement cascade because all elements will have the same refinement stage in each cycle.

For the second part of the mesh refinement process, the real mesh refinement, the CPU times are between 6.14 seconds and 6.77 seconds. This is a difference of about 10% which is an acceptable result but obviously the communication overhead gradually affects the performance as the time increases more and more for the higher numbers of processors. On the other hand, as we increase the number of nodes in y-direction from one grid to the other, the number of overlap edges also increases, and so the time for the exchange of the *ptl* array where we store the data of the overlap refinement edges must also increase.

5.6 Example for 2-D dividing line

The following is an example for a 2-D dividing line. In figure 5.6.1 the grid is shown. It consists of two subdomains divided by a dividing line at $y = 0$. First we generated subdomain 1 with I-DEAS and then we got subdomain 2 by mirroring the nodes of subdomain 1. We use on both subdomains a 40×19 grid, i.e. we have 760 nodes and $1{,}404$ elements in each subdomain, and we have matching grids. Again we solve the problem (5.3.1), (5.3.2) from above. The test function in the upper subdomain is the sugar loaf (5.3.5) with top at $x = 0$, $y = 0$, but we add $+1$, so the function is

$$\bar{u}(x, y) = e^{-8(x^2+y^2)} + 1 = \bar{v}(x, y) = \bar{\omega}(x, y). \tag{5.6.1}$$

In the lower subdomain we use the constant function (5.3.6). The dividing line consists of an upper dividing line which is the lower boundary of the upper domain and of a lower dividing line which is the upper boundary of the lower domain. The coupling conditions at the dividing line are the jump in the function values and equal y-derivatives, so we have

$$u_{up} - \bar{u}_{up} = u_{low} - \bar{u}_{low} \tag{5.6.2}$$

on the upper dividing line and

$$u_{y,low} - \bar{u}_{y,low} = u_{y,up} - \bar{u}_{y,up} \tag{5.6.3}$$

on the lower one. We choose the consistency order $q = 2$, the parameters Δq and ε_{pivot} for the generation of the difference and error formulas are set to $\Delta q = 4$ and $\varepsilon_{pivot} = 10^{-2}$. We prescribe a relative tolerance $tol = 10^{-2}$, the safety factor s_{grid} for the determination of the refinement nodes in (3.4.15) is $s_{grid} = 1$. Again, we start the computation with the exact test solution. We compute on $np = 2$ processors of the IBM SP as above, and we use the BiCGStab2 method without preconditioning for the solution of the large sparse linear system of equations in LINSOL.

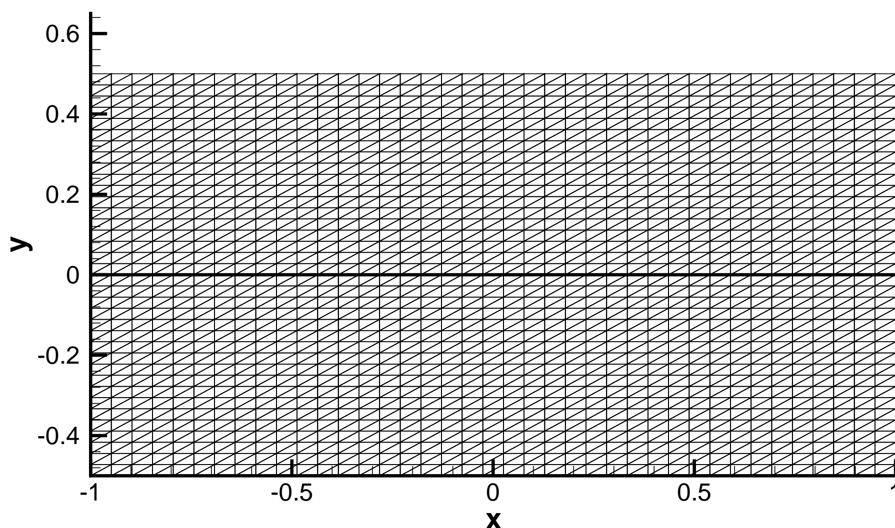


Figure 5.6.1: Domains for the example with dividing line and initial grid.

In table 5.6.1 the results of the refinement process are shown. For each cycle we have in the third column the maximum of the three solution components u, v and ω. These maxima are the maxima over all nodes, i.e. we do not distinguish between nodes of subdomain 1 and 2. In columns 4 and 5 we have the maximum of the relative estimated and exact error, respectively, which are maxima over all nodes as well. The CPU time for the whole cycle is given in column 6, in column 7 the CPU time for the mesh refinement process.

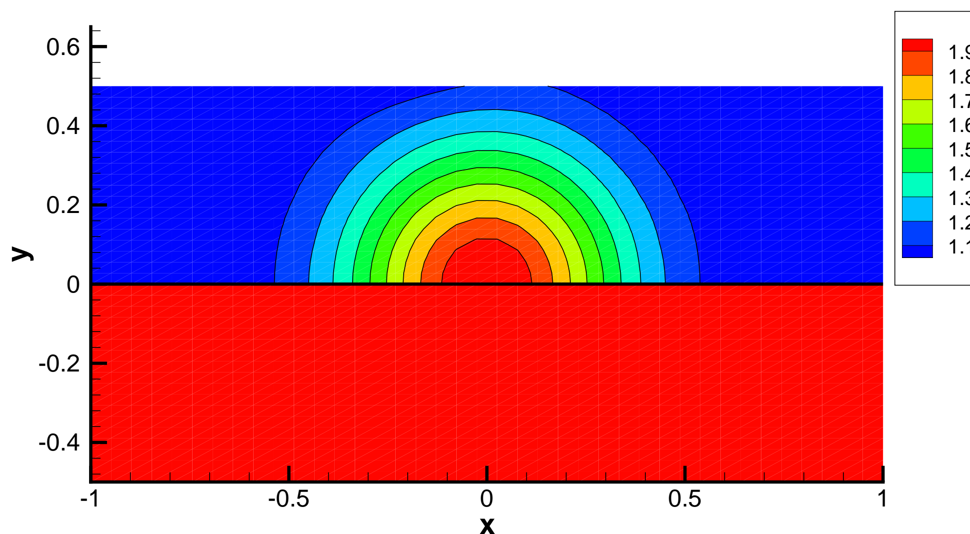


Figure 5.6.2: Solution for the example with dividing line for the four cycles.

Cycle number	comp.	max. of comp.	global relative error estimated	exact	CPU [sec]	CPU ref. [sec]
1	u	2.0000	$0.125 \cdot 10^{-2}$	$0.128 \cdot 10^{-2}$	1.10	0.05
	v	2.0001	$0.882 \cdot 10^{-3}$	$0.916 \cdot 10^{-3}$		
	ω	2.0003	$0.235 \cdot 10^{-1}$	$0.250 \cdot 10^{-1}$		
2	u	2.0000	$0.675 \cdot 10^{-3}$	$0.483 \cdot 10^{-3}$	4.00	0.08
	v	2.0000	$0.287 \cdot 10^{-3}$	$0.261 \cdot 10^{-3}$		
	ω	2.0001	$0.324 \cdot 10^{-1}$	$0.165 \cdot 10^{-1}$		
3	u	2.0000	$0.129 \cdot 10^{-2}$	$0.268 \cdot 10^{-3}$	8.09	0.13
	v	2.0001	$0.456 \cdot 10^{-3}$	$0.329 \cdot 10^{-3}$		
	ω	2.0000	$0.145 \cdot 10^{-1}$	$0.112 \cdot 10^{-1}$		
4	u	2.0000	$0.449 \cdot 10^{-3}$	$0.335 \cdot 10^{-3}$	23.92	
	v	2.0000	$0.305 \cdot 10^{-3}$	$0.257 \cdot 10^{-3}$		
	ω	2.0001	$0.732 \cdot 10^{-2}$	$0.848 \cdot 10^{-2}$		

Table 5.6.1: Results for pure mesh refinement with dividing line (part one).

The maximum of the exact solution is 2 in both subdomains, and this should be the maximum of the computed solution for each solution component. This goal is already attained in the first cycle where we obtain an exact error for the maximum of the solution of $0.15 \cdot 10^{-3}$ for component ω. But there are nodes with a relative estimated error of 2.4% which is above our requested tolerance of 1%—in other words: the requested tolerance is about 40% of the maximum of the relative estimated error in the first cycle—, so that the mesh has to be refined. The computed solution of component ω in the first cycle is illustrated in figure 5.6.2. You can see very well the contours of constant height in the upper subdomain where the exact solution is the sugar loaf function. In the lower subdomain the solution is constant over the whole subdomain. For the solution components u and v the appearance is very similar just as the solution in the following cycles, so we forgo their illustration.

Table 5.6.2 shows the development of the refinement process. With the request (3.4.15) we get 578 refinement nodes altogether in the first cycle of which 562 are refinement nodes (97%) in the upper subdomain and 16 refinement nodes (3%) in the lower one. As the maximum of the relative estimated error is below the requested tolerance in the lower subdomain, this is not astonishing at all. The refinement in the first cycle results in $2{,}676$ nodes in the upper subdomain and 905 nodes in the lower one. This grid is illustrated in figure 5.6.3. Here you can see that the grid of the lower subdomain has not only been refined in the middle of the dividing line—because of errors in the upper subdomain—but also at the right boundary. Here the selection of the nodes for the error formulas is unfavourable.

Cycle number	Domain number	number of nodes	number of elements	number of ref. nodes	number of ref. elem.
1	1	760	1,404	562	1,238
	2	760	1,404	16	61
2	1	2,676	5,119	347	902
	2	905	1,586	0	8
3	1	4,178	7,831	234	790
	2	943	1,613	2	16
4	1	5,674	10,213	—	—
	2	982	1,661	—	—

Table 5.6.2: Results for pure mesh refinement with dividing line (part two). Domain 1: upper domain, domain 2: lower domain.

In the following cycles the estimated errors go down as the grid in the upper subdomain gets finer, see figure 5.6.6a)–d) for the relative estimated error of solution component ω. At the same time the errors in the lower subdomain go down as a consequence of the fact that by the coupling conditions the discretization error of a dividing line node on the upper dividing line enters into the right hand side of the dividing line node on the lower side of the dividing line in the linear system of equations for the computation of the error estimate.

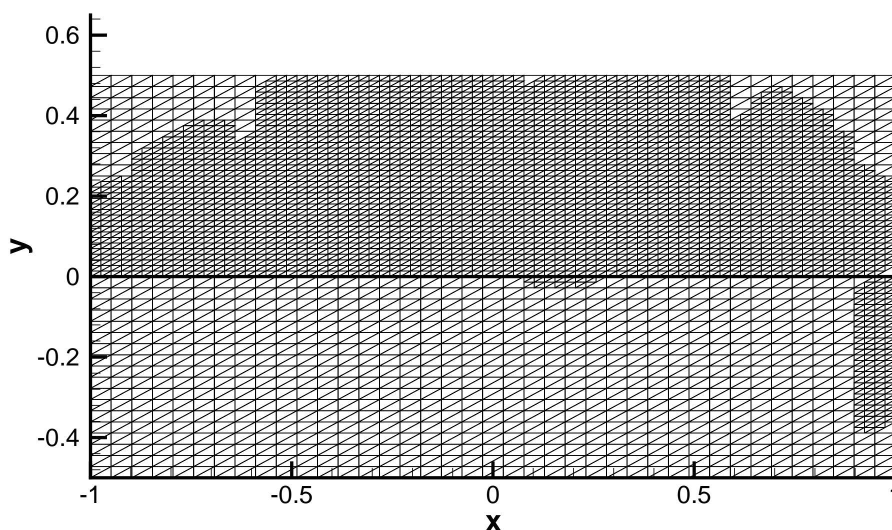


Figure 5.6.3: Domains for the example with dividing line in the second cycle.

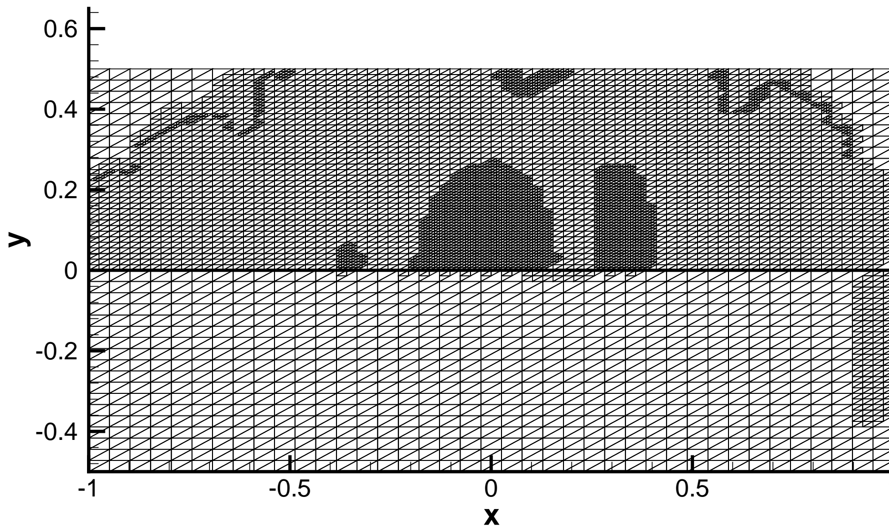


Figure 5.6.4: Domains for the example with dividing line in the third cycle.

The grids for cycle 3 and 4 are illustrated in figure 5.6.4 and 5.6.5, respectively. The lower subdomain is almost unchanged as one can also see from table 5.6.2: we have not any refinement nodes in the second refinement and only 2 refinement nodes after the third cycle. The greatest estimated errors occur in the region of the upper boundary in all 4 cycles, see figure 5.6.6a)–d). This may be quite surprising as the test function is rather harmless here.

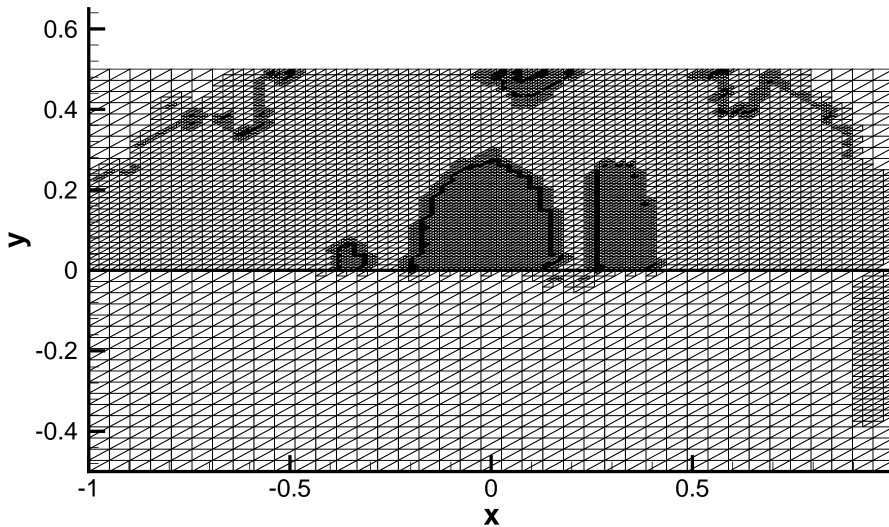


Figure 5.6.5: Domains for the example with dividing line in the fourth cycle.

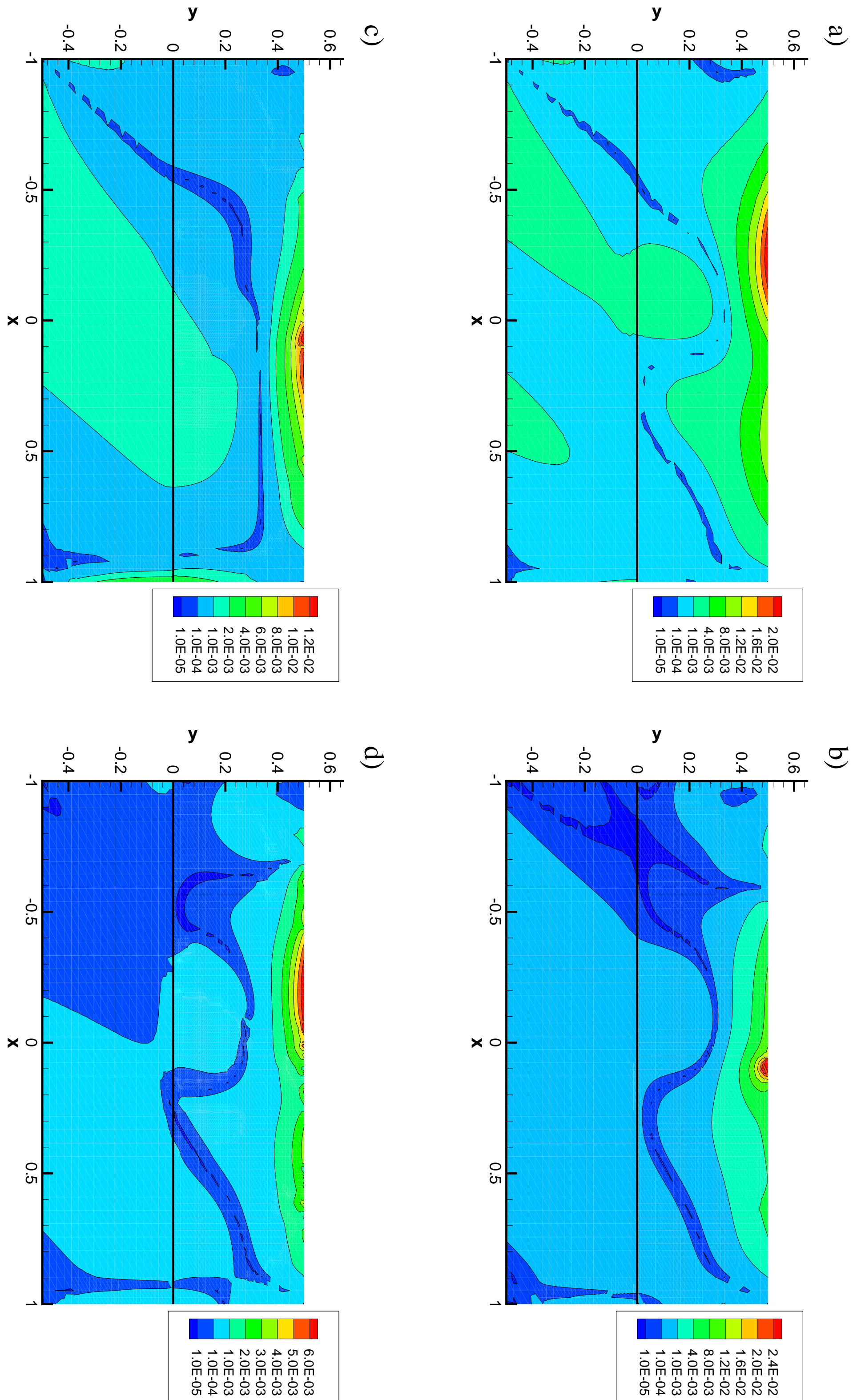


Figure 5.6.6: Relative estimated error of solution component w for the example with dividing line for the four cycles: a) cycle 1, b) cycle 2, c) cycle 3, d) cycle 4.

So the reason is that we have one or more nodes on the upper boundary for which we get bad error formulas. You can also see very well that the grid is locally refined in the middle of the upper boundary (see figures 5.6.4 and 5.6.5).

The refined elements in the third grid directly below the dividing line result from the refinement cascade because of the refinement of elements in the upper subdomain. In the upper subdomain the mesh is mainly refined around the peak of the sugar-loaf in the third cycle but also in the region where the function goes flat. In the fourth cycle two thirds of the refinement nodes are of refinement stage 2, i.e. they have been created just during the last mesh refinement. So this is only a minor correction of the grid that finally leads to a relative estimated error below the prescribed tolerance.

At the end we have 7 times the original number of nodes in the upper subdomain, whereas in the lower subdomain the number of nodes has been increased only by 30%! The total CPU time for the four cycles is 37.11 seconds. Most of the time—overall 82%—is spent in the linear solver LINSOL, this part needs 66% of the time for the first cycle and the percentage increases to 88% in the last cycle. So the time increase one may observe in table 5.6.2 results mainly from LINSOL. This comes from the fact that not only the number of unknowns increases from one cycle to the next but also because we need more matrix vector multiplications for the solution of the large linear system of equations. Remember that the more unknowns we have, the worse the condition of the large sparse matrix Q_d becomes. The time for the mesh refinement is about 3% of the total time for one cycle. At the end the average edge length of the mesh is about 40% of the original average edge length and we meet the requested tolerance that has been 42% of the maximum of the relative estimated error of the basic solution.

5.7 Example for 2-D sliding dividing line

The next example is for a 2-D sliding dividing line. The original grid that consists of two subdomains similar to the last example is illustrated in figure 5.7.1a). First we generated the 2×1 subdomain 1 which is the upper subdomain in the figure with I-DEAS and then we got subdomain 2 by mirroring, scaling and displacing the nodes of subdomain 1. Here we used a scaling factor of 0.5. We use 41 nodes in x-direction and 21 nodes in y-direction on both grids, so that each subdomain has got 861 nodes and $1{,}600$ elements. Therefore the lower subdomain has half the edge lengths of the larger subdomain, and thus we have non-matching grids. Additionally, we have moving grids, i.e. we refine both grids until the maximum of the estimated error is lesser than the prescribed tolerance. Then the lower domain is displaced by -0.175 which corresponds to 3.5 mesh sizes of the upper grid and we start a new refinement process if the tolerance is not met for the new grids. We want to move the lower grid twice. Again we solve the problem (5.3.1), (5.3.2) from above with Reynolds number $Re = 100$.

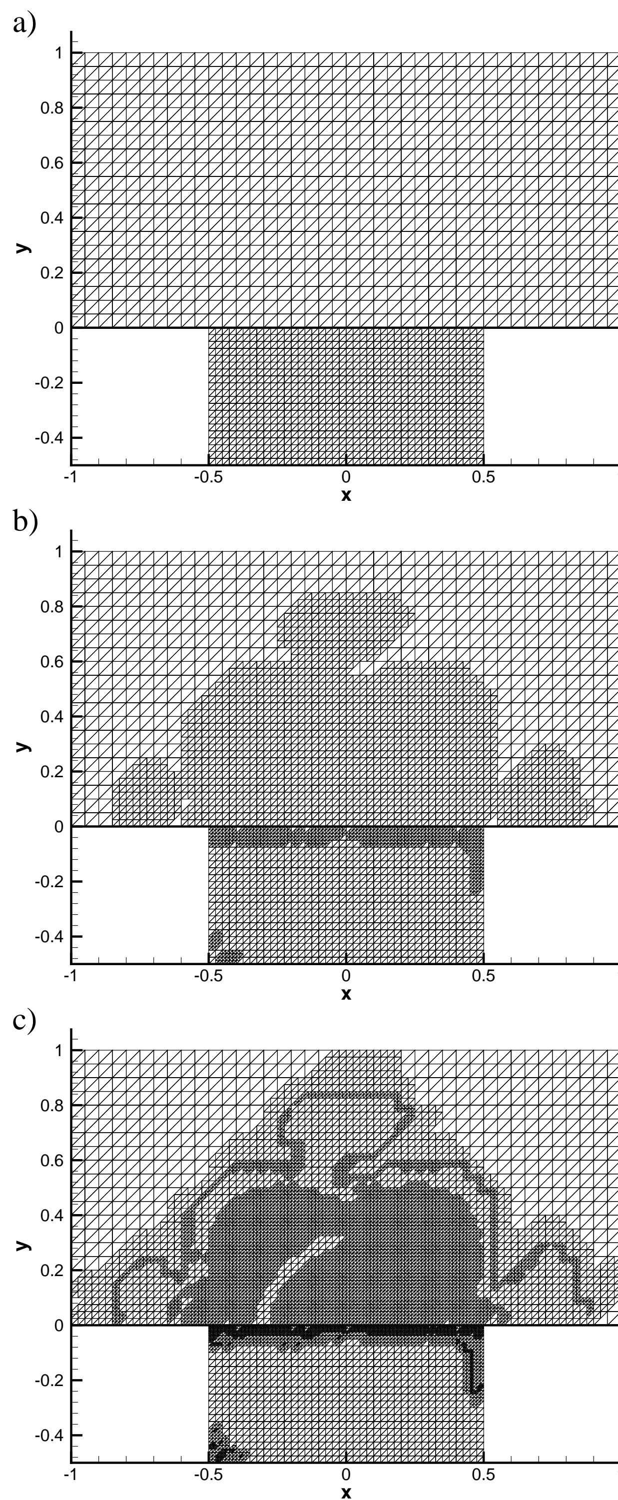


Figure 5.7.1: Domains for the example with sliding dividing line for a) cycle 1, b) cycle 2, c) cycle 3.

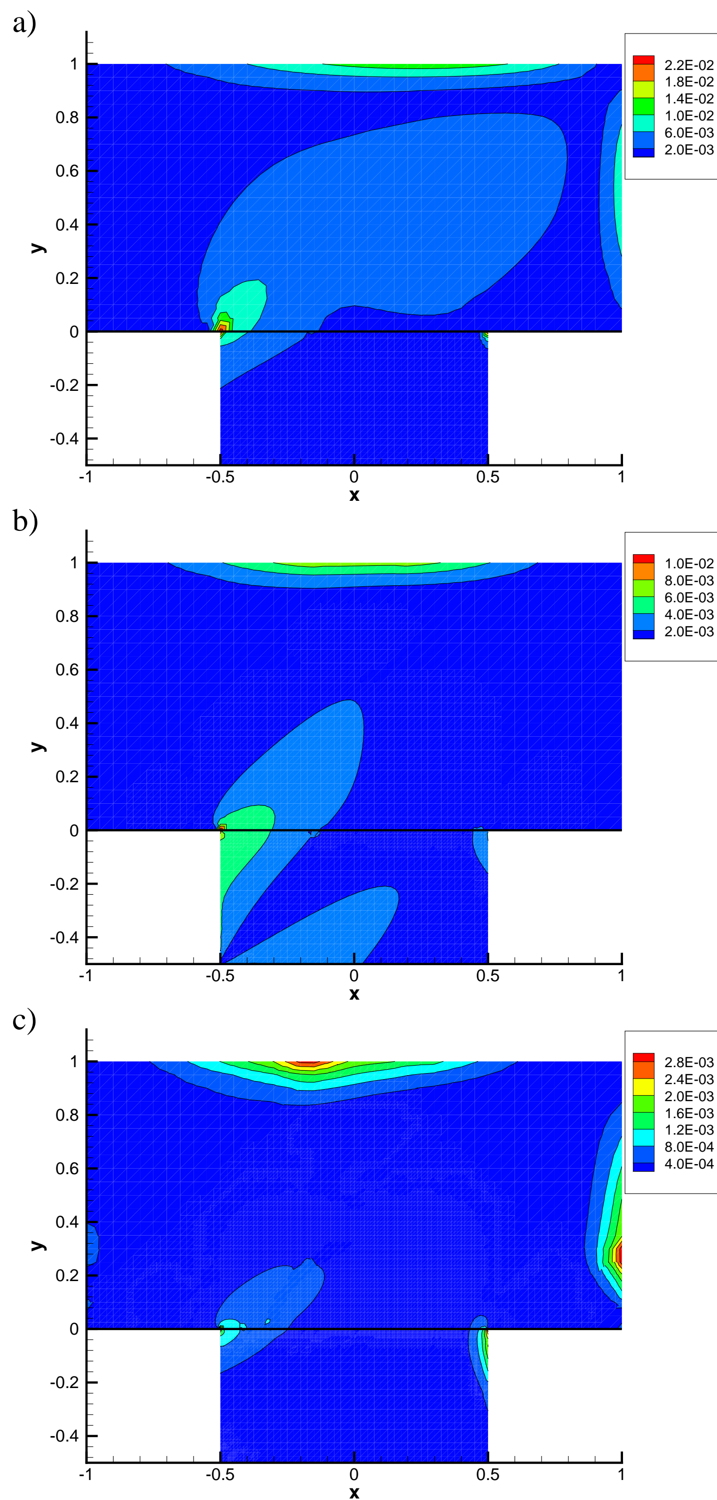


Figure 5.7.2: Estimated error of component ω for the example with sliding dividing line for a) cycle 1, b) cycle 2, c) cycle 3.

We have one test function for each subdomain, in the upper one it is the sugar loaf

$$\bar{u}(x,y) = e^{-8(x^2+y^2)} = \bar{v}(x,y) = \bar{\omega}(x,y). \tag{5.7.1}$$

with top at $x = 0$, $y = 0$, but we add $+1$, and in the lower subdomain we use the constant test function

$$\bar{u}(x,y) = 2 = \bar{v}(x,y) = \bar{\omega}(x,y). \tag{5.7.2}$$

The coupling conditions at the sliding dividing line are the jump in the function values and equal y-derivatives, so we have

$$u_{up} - \bar{u}_{up} = u_{low} - \bar{u}_{low} \tag{5.7.3}$$

for the sliding dividing line nodes on the upper sliding dividing line and

$$u_{y,low} - \bar{u}_{y,low} = u_{y,up} - \bar{u}_{y,up} \tag{5.7.4}$$

for those on the lower one.

Cycle		max. of	global relative error		CPU	CPU ref.
number	comp.	comp.	estimated	exact	[sec]	[sec]
1	u	2.0000	$0.144 \cdot 10^{-2}$	$0.146 \cdot 10^{-2}$	1.27	0.05
	v	2.0000	$0.117 \cdot 10^{-2}$	$0.179 \cdot 10^{-2}$		
	ω	2.0480	$0.246 \cdot 10^{-1}$	$0.234 \cdot 10^{-1}$		
2	u	2.0000	$0.247 \cdot 10^{-2}$	$0.695 \cdot 10^{-3}$	3.08	0.11
	v	2.0000	$0.256 \cdot 10^{-2}$	$0.853 \cdot 10^{-3}$		
	ω	2.0291	$0.120 \cdot 10^{-1}$	$0.151 \cdot 10^{-1}$		
3	u	2.0000	$0.366 \cdot 10^{-3}$	$0.268 \cdot 10^{-3}$	9.97	0.00
	v	2.0001	$0.387 \cdot 10^{-3}$	$0.255 \cdot 10^{-3}$		
	ω	2.0017	$0.310 \cdot 10^{-2}$	$0.197 \cdot 10^{-2}$		
4	u	2.0000	$0.550 \cdot 10^{-2}$	$0.385 \cdot 10^{-2}$	9.27	0.00
	v	2.0001	$0.555 \cdot 10^{-2}$	$0.391 \cdot 10^{-2}$		
	ω	2.0166	$0.848 \cdot 10^{-2}$	$0.823 \cdot 10^{-2}$		
5	u	2.0001	$0.116 \cdot 10^{-2}$	$0.297 \cdot 10^{-3}$	9.12	0.00
	v	2.0002	$0.120 \cdot 10^{-2}$	$0.266 \cdot 10^{-3}$		
	ω	2.0005	$0.625 \cdot 10^{-2}$	$0.562 \cdot 10^{-2}$		

Table 5.7.1: Results for pure mesh refinement with sliding dividing line (part one).

We choose the consistency order $q = 2$ and set $\Delta q = 4$, $\varepsilon_{pivot} = 10^{-2}$ and start with the exact solution. The requested tolerance on the level of solution is $tol = 10^{-2}$ which is

about 40% of the maximum of the estimated error of the basic solution. The safety factor s_{grid} for the mesh refinement is set to 0.5.

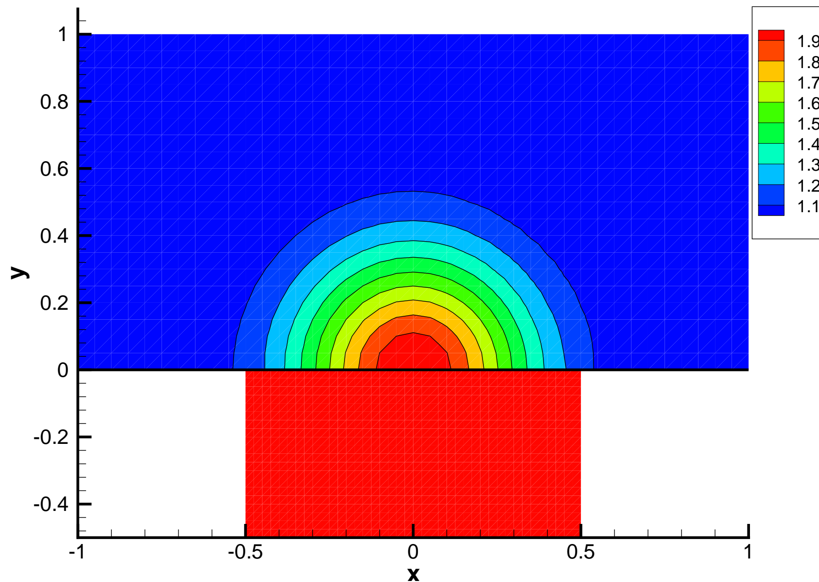


Figure 5.7.3: Solution component ω for the example with sliding dividing line.

We compute on $np = 2$ processors of the IBM SP as above. Again we use the BiCGStab2 method without preconditioning for the solution of the large sparse linear system of equations. The results of the refinement process are presented in tables 5.7.1 and 5.7.2. The layout of the tables is the same as in the preceding section, i.e. in table 5.7.1 we have the maxima of the solution and the relative estimated and exact error for each cycle and each solution component in columns 3, 4 and 5. The CPU times for the whole cycle and for the mesh refinement process are given in columns 6 and 7. In table 5.7.2 the numbers of nodes

Cycle number	Domain number	number of nodes	number of elements	number of ref. nodes	number of ref. elem.
1	1	861	1,600	292	713
	2	861	1,600	80	270
2	1	1,989	3,756	913	2,178
	2	1,335	2,405	201	609
3, 4, 5	1	5,506	10,279	0	0
	2	2,409	4,261	0	0

Table 5.7.2: Results for pure mesh refinement with sliding dividing line (part two). Domain 1: upper domain, domain 2: lower domain.

and elements for each cycle and each subdomain are presented as well as the number of refinement nodes and elements.

The maximum of the exact solution is 2 in both subdomains, and this should be the maximum of the computed solution for each solution component. This goal is attained in the first cycle for the solution components u and v, but for ω the maximum is 2.048. The maximum of the relative estimated error for ω is 2.5% and is found in the upper subdomain. The computed solution of component ω in the cycle 1 is illustrated in figure 5.7.3. You can see very well the contours of constant height in the upper subdomain where the exact solution is the sugar loaf function. In the lower subdomain the solution is constant over the whole subdomain. In the first three cycles the grid is not moved yet, so we forgo the illustration of ω for cycle 2 and 3. For the solution components u and v the appearance is very similar,

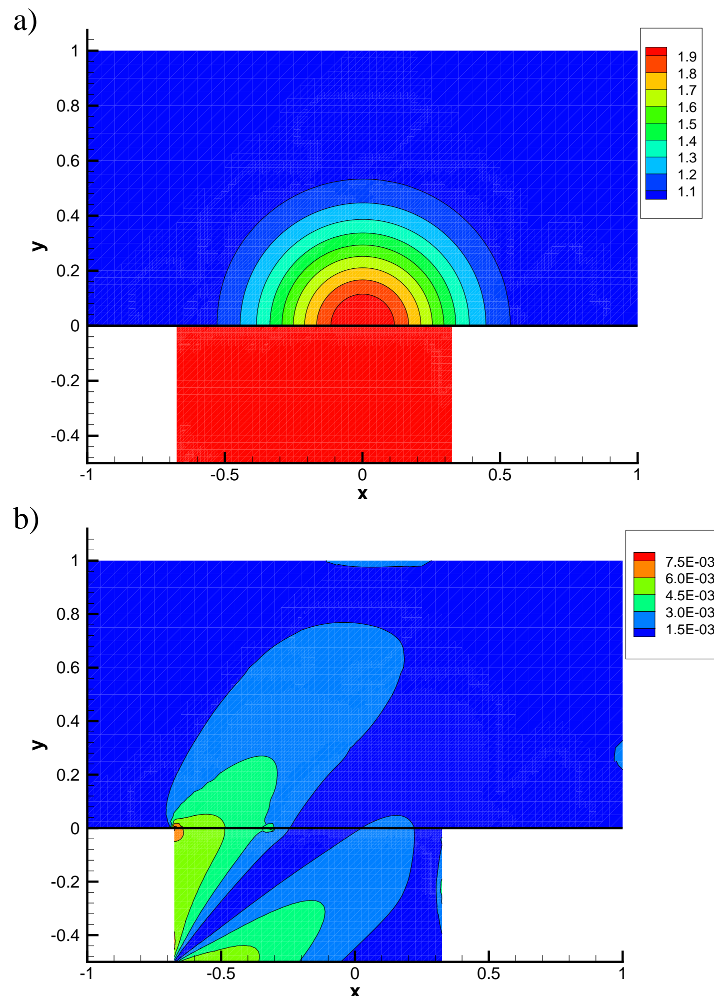


Figure 5.7.4: Cycle 4 for the example with sliding dividing line: a) solution component ω, b) error estimate of ω.

so we also forgo their illustration.

By request (3.4.15) we get 372 refinement nodes (about 80% of these nodes are in subdomain 1 and 20% in subdomain 2), see table 5.7.2. The refinement nodes are positioned in a circle around the peak of the sugar-loaf in the upper subdomain. In the lower subdomain the refinement nodes are in the proximity of the sliding dividing line. Only in the lower left corner and at the right boundary of the subdomain we have some bad error formulas that produce some refinement nodes, see figure 5.7.1b) for the grid of the second cycle. The refinement in the first cycle leads to $1{,}989$ nodes in the upper subdomain and $1{,}335$ nodes in the lower one. The maximum of the relative estimated error decreases to 1.2% for solution component ω and therefore is still above the prescribed tolerance so that another refinement and another computation cycle is necessary.

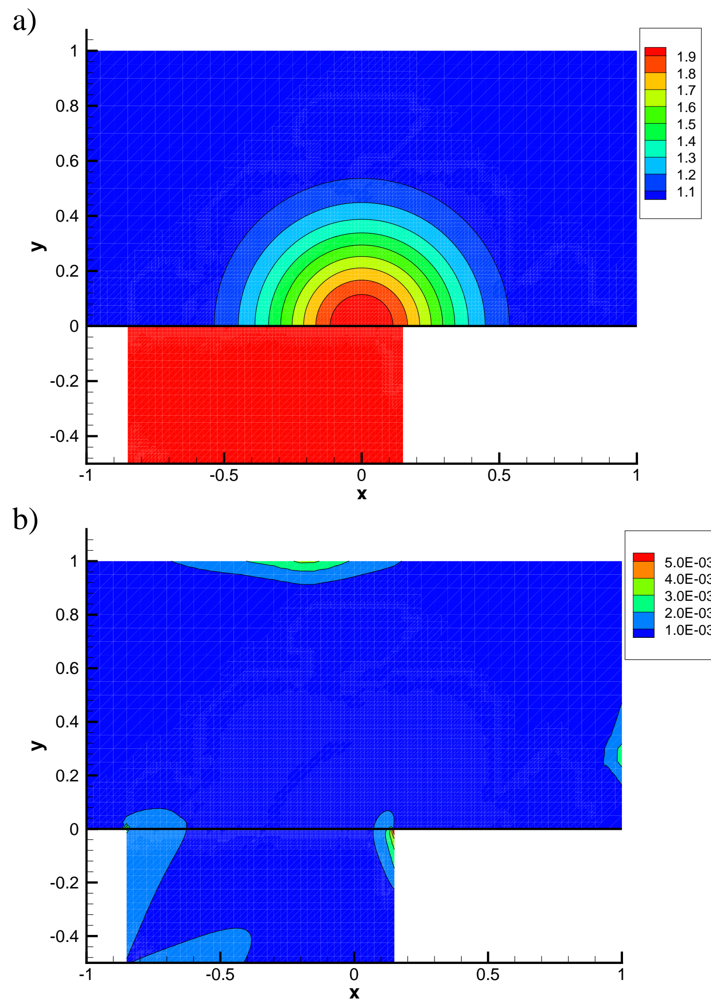


Figure 5.7.5: Cycle 5 for the example with sliding dividing line: a) solution component ω, b) error estimate of ω.

The grid of the third cycle is illustrated in figure 5.7.1c). We can see that the refinement concentrates on the peak of the sugar-loaf in the upper subdomain, in the lower subdomain only some nodes in the direct proximity of the sliding dividing line become refinement nodes. In the third cycle the maximum of the relative estimated error is 0.3% and so we meet the requested tolerance now. The estimated error for solution component ω, the significant component for the refinement, is illustrated in figure 5.7.2 for the three cycles.

Afterwards the lower subdomain is moved by 0.175 to the left and again, we start the refinement process until the prescribed tolerance is met for the new constellation. As the maximum of the estimated error is 0.8% for solution component ω, no further mesh refinement is necessary. We illustrate solution component ω and the error of this solution component in figure 5.7.4. Again, we forgo the illustration of the other solution components and their error estimates. The lower subdomain is moved to the left once more and we immediately get an estimated error of 0.6% so that we do not have to refine the mesh any more and the computation is finished. Figure 5.7.5 illustrates solution component ω and its error estimate for cycle 5, u and v are forgone. As the upper subdomain is not changed at all and the lower one is only moved to the left, we also forgo the illustration of the grid for cycle 4 and 5.

5.8 A 3-D example

In 3-D we want to solve a system of 3 partial differential equations for the variables u, v and w which is again of Navier-Stokes type:

$$\begin{aligned}
u_{xx} + u_{yy} + u_{zz} + u + Re\cdot(uu_x + vu_y + wu_z) - f_1 &= 0\\
v_{xx} + v_{yy} + v_{zz} + v + Re\cdot(uv_x + vv_y + wv_z) - f_2 &= 0\\
w_{xx} + w_{yy} + w_{zz} + w + Re\cdot(uw_x + vw_y + ww_z) - f_3 &= 0
\end{aligned} \tag{5.8.1}$$

with the boundary conditions

$$\begin{aligned}
u - g_1 &= 0\\
v - g_2 &= 0\\
w - g_3 &= 0.
\end{aligned} \tag{5.8.2}$$

The f_i and g_i are forcing functions that are determined so that the exact solution is

$$\bar{u}(x,y) = e^{-32(x^2+y^2+z^2)} = \bar{v}(x,y) = \bar{\omega}(x,y) \tag{5.8.3}$$

which is the 3-D equivalent to (5.3.4) with maximum 1 in the origin and decaying in balls around the origin. We solve (5.8.1) and (5.8.2) with Reynolds number $Re = 100$ on a unit cube with center in the origin of the coordinate system. We compute with $np = 16$ processors of the HP XC1 6000 with 1500 MHz Itanium-2 processors with a peak performance of

6000 MFLOPS. We use the slow, but very robust iterative CG solver ATPRES, see [10], because other CG solvers, e.g. BiCGStab2, do no longer converge for the third cycle because of the large number of unknowns, and LU preconditioning is slower because of the large fill-in for 3-D problems. We start with a $17 \times 17 \times 17$ grid, compute with fixed consistency order $q = 2$ with $\Delta q = 4$ and $\varepsilon_{pivot} = 10^{-3}$. The initial solution is the sugar-loaf function (5.8.3), i.e. the exact solution. We compute with pure mesh refinement where we choose $s_{grid} = 10^{-3}$ and request a tolerance on the level of solution $tol = 10^{-2}$. The given CPU time is that for the master processor 1.

The results of the refinement process are presented in tables 5.8.1 and 5.8.2. The layout of the tables is the same as in the preceding sections, i.e. in table 5.8.1 we have the maxima of the solution and the relative estimated and exact error for each cycle and each solution component in columns 3, 4 and 5. The CPU times for the whole cycle and for the mesh refinement process are given in columns 6 and 7. In table 5.8.2 the numbers of nodes and elements for each cycle are presented as well as the number of refinement nodes and elements.

Cycle number	comp.	max. of comp.	global relative error estimated	exact	CPU [sec]	CPU ref. [sec]
1	u	1.0417	$0.375 \cdot 10^{-1}$	$0.400 \cdot 10^{-1}$	2.21	0.25
	v	1.0417	$0.375 \cdot 10^{-1}$	$0.400 \cdot 10^{-1}$		
	ω	1.0417	$0.375 \cdot 10^{-1}$	$0.400 \cdot 10^{-1}$		
2	u	1.0121	$0.328 \cdot 10^{-1}$	$0.300 \cdot 10^{-1}$	70.80	44.99
	v	1.0121	$0.318 \cdot 10^{-1}$	$0.300 \cdot 10^{-1}$		
	ω	1.0121	$0.334 \cdot 10^{-1}$	$0.300 \cdot 10^{-1}$		
3	u	1.0003	$0.957 \cdot 10^{-2}$	$0.787 \cdot 10^{-2}$	488.29	
	v	1.0003	$0.857 \cdot 10^{-2}$	$0.785 \cdot 10^{-2}$		
	ω	1.0003	$0.887 \cdot 10^{-2}$	$0.784 \cdot 10^{-2}$		

Table 5.8.1: Results for pure mesh refinement for 3-D example (part one).

In cycle 1 we start with $4{,}913$ nodes and $24{,}576$ elements of the original grid. The maximum of the solution of the first cycle is 1.0417 for each component because of the symmetric structure of the PDEs. In figure 5.8.1 the solution for component u is illustrated, for the other components the appearance is almost the same, so we forgo their illustration. As it is quite difficult to illustrate the solution of a 3-D problem we have chosen the illustration of a 2-D cut through the cube, the section plane is the yz-plane, i.e. $x = 0$. The maximum of the relative error is 3.75% for all solution components. By condition (3.4.15) we get $2{,}649$ refinement nodes which results in $30{,}113$ nodes and $163{,}561$ elements for cycle 2.

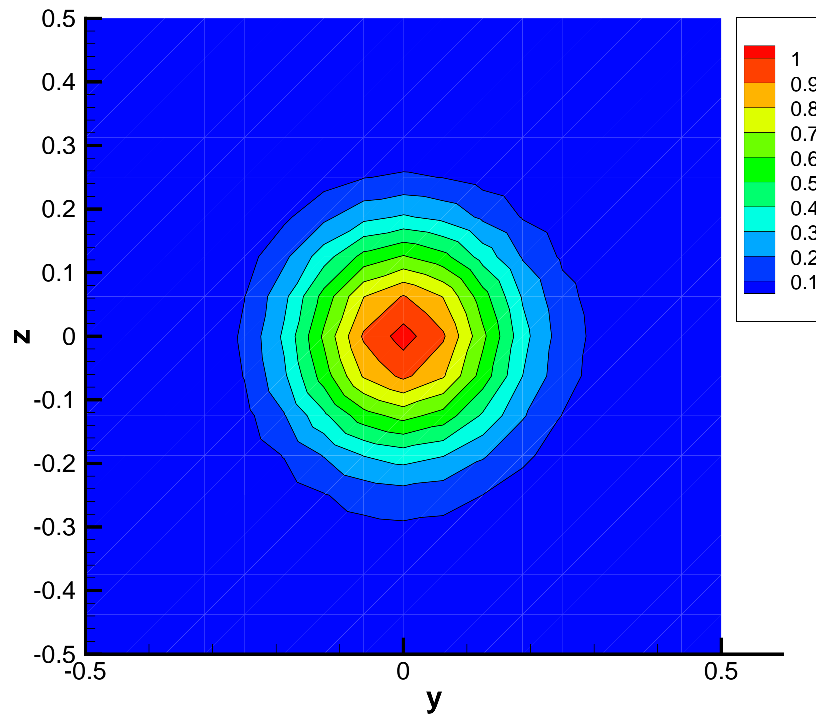


Figure 5.8.1: Solution u of the 3-D example for the section plane $x = 0$.

Of course, the illustration of the grids is difficult, too. So we do not illustrate the grid but the average edge length for all edges from each node to its neighbour nodes where we also show a 2-D cut through the cube with the section plane $x = 0$, see figure 5.8.2. You can see that the average edge length becomes smaller in the middle of the grid where we have the peak value of the 3-D sugar-loaf. As the maximum of the solution is 1.0121 for all components in the second cycle and as this is better than in the first one as the exact maximum is 1, we get a smaller error estimate of 3.34%. Nevertheless, we get $25{,}253$ refinement nodes by (3.4.15), and by the refinement process the grid for the third cycle has got $228{,}258$ nodes and 1.3 Million elements. Then the maximum of the solution is 1.0003 and the relative estimated error decreases to 0.96% which is below the prescribed tolerance.

Cycle number	number of nodes	number of elements	number of ref. nodes	number of ref. elem.
1	4,913	24,576	2,649	19,855
2	30,113	163,561	25,253	162,512
3	228,258	1,301,145	—	—

Table 5.8.2: Results for pure mesh refinement for 3-D example (part two).

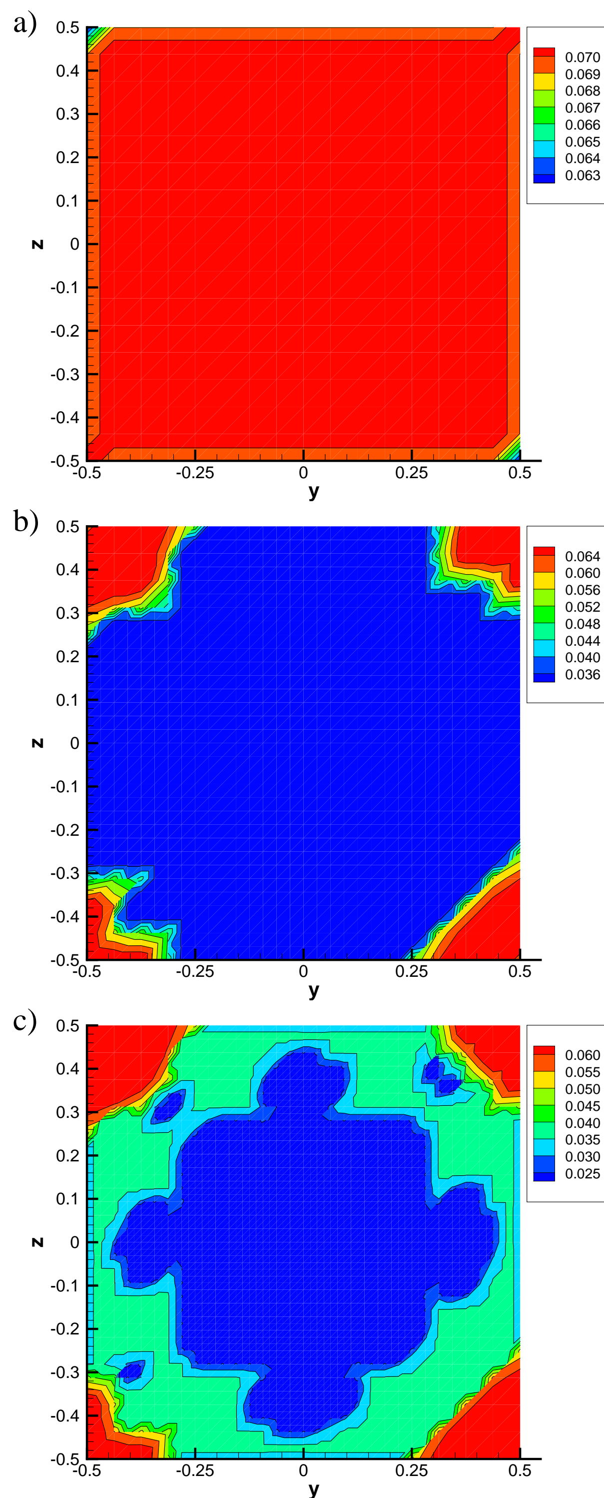


Figure 5.8.2: Average edge length between a node and its neighbour nodes for the 3-D example for a) cycle 1, b) cycle 2, c) cycle 3, section plane $x = 0$.

So the computation is finished. It is clear that in 3-D the number of nodes and elements increases rapidly with the number of refinement cycles. The attentive reader may observe that the subfigures with the average edge lengths in figure 5.8.2 are not symmetric. This comes from the fact that the grid is not symmetric, too. So this results in different difference stars and therefore also in (slightly) different values for the solution and the error estimate.

The total time for the computation is 561.30 seconds, but we can see that the time for the individual cycles increases more than we expect when we look at the number of nodes and elements. This comes from the fact that the data fits into the L2-cache in the first cycle and partly in the second cycle, but in the third cycle the arrays need too much memory so the data must be loaded from the memory. This also holds for the mesh refinement. The number of refinement nodes and elements is increased ninefold to tenfold in the second cycle compared with the first cycle, but the time for the mesh refinement is 45 seconds instead of 0.25 seconds! But with an increasing number of unknowns the condition number of the large sparse matrix Q_d increases, too.

For the 3-D example we can see that the FDEM program package is still working if we compute on a number of processors (16) that is almost equal to the number of nodes in x-direction (17). It is even working if you choose a number of processors that is greater than the number of nodes in x-direction. This has been quite difficult to realize because then the owning processor numbers of neighboured nodes and elements may differ by more than one and the righmost processors may own no elements at all. This shows the advantage of our automatic 1-D domain decomposition that we need for the black-box property of the domain of FDEM.

In 3-D the mesh refinement is very critical. It is very difficult to find parameters so that we get a computation for which the relative estimated error decreases when the space step size is reduced. If we refine the grid completely, the error will decrease to 25% of the error of the basic solution (for consistency order $q = 2$) but for a "mixed" grid where we have elements of different refinement stages, there will be almost surely at least one node with bad formulas. Therefore it is recommended to refine the grid completely in 3-D which gives better decrease in the error than local refinement, but the number of nodes is the eight-fold. If we look at the mean error, i.e. we sum up the relative errors of all n nodes and divide the result by n, we will see that it decreases from $0.79 \cdot 10^{-3}$ in the first cycle to $0.90 \cdot 10^{-4}$ in the third cycle. Simultaneously, the average edge length goes down to $0.21 \cdot 10^{-2}$ in the last cycle from $0.71 \cdot 10^{-2}$ for the original grid. So the ratio of the average edge lengths is 30% and we therefore expect an error ratio of about 9% as the error should decrease quadratically with the space step size. Actually, the ratio of the mean errors in our computation is 11%.

6 Summary and Outlook

In this chapter, a summary of the theoretical and practical results of this thesis is given. Further, open questions and problems are identified.

In chapter 3, we have explained the Finite Difference Element Method to solve nonlinear systems of partial differential equations on an arbitrary FEM mesh. We designed a robust black-box solver that gives us, additionally to the solution of the problem, a reliable error estimate for each component in each grid point, which is a unique feature among the FDM and FEM program packages where one usually only gets an error indicator. By the means of the error estimate we can assign an individual consistency order to each node and we are able to refine the mesh locally if necessary. By the introduction of dividing lines and coupling conditions we may solve problems on domains that are composed from different subdomains with different PDEs which is also unique for the FDM. Starting from this base we developed FDEM with sliding dividing lines where we do not need matching grids for the subdomains that even may slide relatively to each other.

For the mesh refinement the nodes that have a relative error estimate that is above a prescribed tolerance become refinement nodes, the elements that contain at least one refinement node become refinement elements. The refinement is executed by the bisection of the edges of the refinement elements. If an element is a neighbour element of a refinement element and has a smaller refinement stage at the same time, i.e. it is “larger”, this element must also be refined (refinement cascade) because otherwise, we would have more than 3 nodes on the edge the two elements share. This must be avoided because of the data storage scheme.

In chapter 4 we explained all aspects of the mesh refinement algorithm in 2-D and 3-D, with and without dividing lines and sliding dividing lines, respectively. The parallelization of the mesh refinement algorithm on distributed memory parallel computers is very challenging, especially in 3-D where we may have many elements that share an edge, and these elements may be owned by different processors. In order to manage this task we had to put up clear rules that define the responsibility of the processors for the refinement of the elements. The refinement then is reduced to the bisection of the edges and the combination of the new mid-points and corner nodes of the old elements to get the new refined elements for the following refinement cycle. Another hard problem was to take care that all necessary information is available on each processor at any time.

FDEM is a black-box, i.e. you never know what the solution domain looks like on which the user wants to solve the PDEs. Only by the means of our 1-D domain decomposition we are able to realize an automatic mesh refinement on an arbitraray domain with fixed data structures (maximal 3 nodes per edge). This results in the left and right overlap where

the processors hold the necessary data of their neighbours, and it results during the mesh refinement process in the refinement cascade. Therefore the 1-D domain decomposition is the kernel of the whole mesh refinement process.

For the parallelization we elaborated some principles that are of great importance, not only for the FDEM program package and its mesh refinement algorithm, but for many algorithms implemented on a distributed memory parallel computer. The first principle concerns the data itself to be passed to neighbour processors. We always collect as much data as possible before the data exchange is started. This is because the performance will go down drastically if we send each piece of information individually as the startup time of the communication will rise above the transfer time for the messages. The second principle concerns the method of data exchange: After we compiled in an array a number of data elements that we want to send to a remote process, we first send the number of data elements into the direction of the communication, so that it is known afterwards on each overlap processor how many data elements it will receive from which processor in the following data exchange process, and if there is no information at all to receive in a communication cycle, this is also known both on the sending and on the receiving processor, and the communication is omitted.

In chapter 5 we first gave an example for the pure mesh refinement on the unit circle where we solved a nonlinear system of equations of Navier-Stokes type. We saw by the relative estimated error that the test polynomial of order 6 is approximated better and better from cycle to cycle, by the local mesh refinement, although we computed with consistency order $q = 2$.
Then we illustrated the scalability of the FDEM code in general, and afterwards of the mesh refinement algorithm in particular. We learned that the mesh refinement algorithm scales very well, whereas the scalability of the whole FDEM code largely depends on the linear solver LINSOL that does not scale very well. Ignoring the LINSOL time, we see that FDEM scales very well. The performance is excellent as long as the data comes from the L2-cache. However, when the data comes from the memory for large problems, the performance goes down as expected.
Afterwards we illustrated two examples for a computation with pure mesh refinement for a domain composed from several subdomains, first separated by dividing lines, then by sliding dividing lines where we additionally moved one of the subdomains. From the example with dividing lines we saw how the mesh refinement continues on the other side of a dividing line. For the dividing line example we met the prescribed tolerance after 4 refinement cycles, for the example with sliding dividing lines we must refine the mesh until the prescribed tolerance is met before we move the sliding subdomain. We needed 3 refinement cycles for this, afterwards we moved the sliding subdomain twice, and each time we immediately met the tolerance in the first refinement cycle.
The last example is a 3-D example where we solved a system of 3 Navier-Stokes equations

on the unit cube. We saw that the number of nodes and elements goes up very fast as almost the whole grid is refined in a refinement cycle, after three refinement cycles the prescribed tolerance is finally met. In the last computation cycle we see very clearly that the performance of FDEM goes down drastically if the data comes from the memory.
We saw that the relative estimated error can be brought down below the prescribed tolerance by the means of pure mesh refinement. However, this is also the result of an expensive process of the variation of the computational parameters such as the safety factor s_{grid} for the mesh refinement, consistency order q and the pivot bound ε_{pivot} for the selection of the appropriate nodes for the difference formaulas. Especially in 3-D, it is very difficult to find a set of parameters to get a computation where the estimated error decreases uniformly without refining the whole grid. So we can conclude that the best set of parameters that can be applied for each problem does not exist, and it is a challenge to find a set of parameters that gives the desired result. This search is executed with "open eyes" because the error estimate shows the quality of the solution.

As always there are left some improvements to be done in the future. We already mentioned in chapter 4 that the performance can be further improved for the search for neighbour elements of the same refinement stage by sorting the elements a node belongs to in ascending order before the search. If we also numbered the edges of the elements globally, we did not need to identify an edge by its end points but by its number. By this means the performance of the mesh refinement will improve but the realization is quite time-consuming, and poses new problems for the implementation. Furthermore, it may be useful to implement mesh coarsening in the FDEM program package for time-dependent problems where we need the finer grid only for a certain time period.

The refinement of an element is based on the bisection of its edges and the generation of the mid-points of these refinement edges. The new mid-points and the old corner nodes of a refinement element are connected such that we get new elements of the next refinement stage. This is done the same way in 2-D and 3-D, irrespective of the space dimension. So we can generalize the mesh refinement of the Finite Difference Element Method for n-dimensional problems in $\mathbb{R}^n$ with $n \geq 2$. Furthermore, it is possible by some not too extensive changes to use some different elements when generating the FEM mesh, not only triangles in 2-D and tetrahedrons in 3-D, respectively. This is one of the improvements for the Finite Difference Element Method we intend to implement in the future to achieve even more flexibility.

References

[1] H. P. Langtangen, Computational Partial Differential Equations, Texts in Computational Science and Engineering 1, 2nd edition, Springer-Verlag Berlin Heidelberg New York, 2003.

[2] S. Larsson, V. Thomée, Partial Differential Equations with Numerical Methods, Texts in Applied Mathematics 45, Springer-Verlag Berlin Heidelberg New York, 2003.

[3] W. Schönauer, T. Adolph, How WE solve PDEs, Journal of Computational and Applied Mathematics 131, 2001, pp. 473-492.

[4] W. Schönauer, T. Adolph, FDEM: How we make the FDM more flexible than the FEM, in Computational and Mathematical Methods in Science and Engineering, Proceedings of the CMMSE-2002, Alicante, Spain, September 20-25, 2003, edited by J. Vigo-Aguiar and B. A. Wade, Deposito Legal (Spain) S.1026-2002, Vol. II, pp. 313-322, and in Journal of Computational and Applied Mathematics, Vol. 158, Issue 1, 2003, pp. 157-167.

[5] W. Schönauer, Scientific Computing on Vector Computers, North-Holland, Amsterdam, 1987.

[6] T. Adolph, W. Schönauer, The generation of high quality difference and error formulae of arbitrary order on 3-D unstructured grids, ZAMM 81, Supplement 3, 2001, pp. 753-754.

[7] W. Schönauer, K. Raith, G. Glotz, The SLDGL program package for the selfadaptive solution of nonlinear systems of elliptic and parabolic PDEs, in Advances in Computer Methods for Partial Differential Equations-IV, edited by R. Vichnevetsky and R. S. Stepleman, IMACS, New Brunswick, 1981, pp. 117-125.

[8] W. Schönauer, Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers, selfedition Willi Schönauer, Karlsruhe, Germany, 2000, ISBN 3-00-005484-7, see
http://www.rz.uni-karlsruhe.de/∼rx03/book/ .

[9] M. Schmauder, W. Schönauer, CADSOL - A fully vectorized black box solver for 2-D and 3-D partial differential equations, in R. Vichnevetsky, D. Knight, G. Richter (Eds.), Advances in Computer Methods for Partial Differential Equations-VII, IMACS, New Brunswick, 1992, pp. 639-645.

[10] LINSOL, see
http://www.rz.uni-karlsruhe.de/rd/linsol.php .

[11] H. Häfner, W. Schönauer, The Integration of different variants of the (I)LU algorithm in the LINSOL program package, Applied Numerical Mathematics 41, 2002, pp. 39-59.

[12] D. Zundel, W. Schönauer, A fast "parallelized" single pass bandwidth optimizer for sparse matrices, accessible via [10], documentation.

[13] M. H. Gutknecht, Variants of BiCGStab for matrices with complex spectrum, SIAM Journal of Scientific Computing 14, 1993, pp. 1020-1033.

Remark: A detailed report on FDEM is available: W. Schönauer, T. Adolph, FDEM: The evolution and application of the Finite Difference Element Method (FDEM) program package for the solution of partial differential equations, Rechenzentrum der Universität Karlsruhe, 2005. The report is accessible online at
http://www.rz.uni-karlsruhe.de/rz/docs/FDEM/Literatur/fdem.pdf .

List of Algorithms

A Algorithm for the preparation step of the mesh refinement on a single processor. 75
B Algorithm for the preparation step of the mesh refinement on a distributed memory parallel computer. 83
C Algorithm for one refinement step of the mesh refinement on a single processor. 122
D Algorithm for the issuing of new node numbers to the mid-points of the edges in the *ptl* array (2-D). 135
E Algorithm for updating the element information of elements to be refined because of the refinement cascade on overlap processors. 157
F Algorithm for the issuing of new node numbers to the mid-points of the edges in the *ptl* array (3-D). 162
G Algorithm for passing the edge information to overlap processors that own neighbour elements of refinement elements on the sending processor. 169
H Algorithm for one refinement step of the mesh refinement on a distributed memory parallel computer. 174

List of Figures

3.1.1 Illustration for the generation of row i of the matrix Q_d for a scalar PDE. . 15
3.3.1 Example of $m = 6$ nodes for a polynomial of order $q = 2$. 21
3.3.2 Triangle and corresponding *nek* array. 23
3.3.3 Array *nekinv* which gives for each node the element numbers in which it occurs. 24
3.3.4 Nearest neighbour ring and 3 further rings for inner node. 25
3.3.5 Array *fstring* with the node numbers of the nearest neighbour ring. 25
3.3.6 Illustration of node collection limits for the order $q = 2$ on a rectangular mesh a) for an inner node, b) for a boundary node. 26
3.3.7 Illustration for the selection of m equations out of $m + r$ equations. 27
3.3.8 Illustration for the coordinate transformation $(x, y) \to (x', y')$. 29
3.3.9 Illustration for the difference formulas of order p for u_t. 31
3.4.1 Refinement of a linear triangular element. 36
3.4.2 Illustration of the refinement cascade on a single processor: a) original grid, b) grid after one refinement, c) grid after two refinement steps without refinement cascade, d) grid after two refinement steps with refinement cascade. 37
3.5.1 Illustration for dividing lines (DLs) and coupling conditions (CCs) with a) jump in derivative, b) jump in function. 38
3.5.2 Illustration of heat flux problem. 39

3.5.3 Illustration for sliding dividing line (SDL). 40
3.5.4 Illustration of the coupling across a sliding dividing line. 41
3.5.5 Illustration of algorithm for property free/coupling node. 42
3.5.6 Illustration of property free/coupling node. 43
3.6.1 Illustration of the distribution of the data to the processors. 44
3.6.2 Illustration for the owning of triangles. 45
3.6.3 Illustration of the basket principle for $np = 4$ processors. 46
3.6.4 Illustration of rings for a rectangular mesh. 47
3.6.5 Distribution of the matrix Q_d and the r.h.s. $(Pu)_d$ in row blocks to $np = 4$ processors. 48
4.1.1 Illustration of the passing of the message lengths to the right for $np = 4$ processors and $np_{max,r} = 3$. 55
4.1.2 Illustration of the passing of the message lengths to the left for $np = 4$ processors and $np_{max,l} = 2$. 58
4.1.3 Illustration of the communication pattern for $np = 8$ processors and $np_{max,r} = np_{max,l} = 3$. 59
4.3.1 Illustration of neighboured elements: neighbour element of left element is of a) higher, b) same, c) lower refinement stage. 66
4.3.2 Illustration of the search for neighbour elements of lower refinement stage. 69
4.3.3 Illustration of the neighbour element search with dividing lines. 70
4.3.4 Illustration of the arrays *dlote* and *dloteadr*. 73
4.3.5 Illustration for the transformation of the array *refel* to the arrays *indrel* and *narpl*. 74
4.3.6 Illustration of the refinement cascade on $np = 4$ processors. 77
4.4.1 Illustration of the arrays *nb* and *nbadd* for element 36 (2-D). 88
4.4.2 Illustration of the search for neighbour elements of the same refinement stage. 90
4.4.3 Local numbering of the nodes in an element (2-D). 90
4.4.4 Illustration of the neighbourhood of two nodes. 91
4.4.5 Illustration of the array *infpol* and *ipadd*. 95
4.4.6 Illustration of array el_{hlp}. 101
4.4.7 Illustration of the refinement of a triangle: a) original element, b) original element with new nodes and elements, c) divide element into four new ones. 102
4.4.8 Local numbering of the nodes in an element (3-D). 104
4.4.9 Illustration of the boundary node property (3-D): new node on refinement edge (bold) is a) a boundary node, b) not a boundary node. 107
4.4.10 Illustration of the refinement of a tetrahedron: a) original element, b) original element with new nodes and edges, c) take away the four corner elements, d) divide kernel into four elements. 110
4.4.11 Illustration of the arrays *nb* and *nbadd* for (dividing line) element 42 (3-D). 112
4.4.12 Illustration of array *rtl* for a dividing line edge a) in 2-D, b) in 3-D. 114

4.4.13 Illustration of *tnod* for new dividing line nodes, 3-D crossing of 8 dividing surfaces (DS). 117
4.4.14 Illustration of the dividing line property in 2-D, edge 1 (bold) is dividing line edge. 119
4.5.1 Illustration of the refinement process on $np = 4$ processors. 124
4.5.2 Illustration of array *ptl* on processor ip after sending to the right for $np_{sr} = 3$. 132
4.5.3 Illustration for global numbering of the new nodes originating from the *ptl* array for $np = 4$ processors. 133
4.5.4 Illustration of a situation where the same refinement edge occurs in the received *ptl* array (received from processor 1) and the (local) *rtl* array of processor 2. 137
4.5.5 Illustration for global numbering of the new nodes originating from the *ptl* and the *rtl* array for $np = 4$ processors. 141
4.5.6 Illustration for local numbering of the new nodes originating from the *ptl* and the *rtl* array on processor ip. 142
4.5.7 Illustration of the owning of the new elements in 2-D. 148
4.5.8 Illustration of the known information at the end of a refinement step a) on refinement processor ip, b) on overlap processor $ip - 1$. 149
4.5.9 Storage scheme of array *infarr*. 152
4.5.10 Illustration of a) the numbers of elements stored in the element arrays *nek* and *nenr*, b) the numbers of nodes stored in the node arrays *nnr*, *x*, *y*, *u* and *q* at the end of a refinement step. 155
4.5.11 Illustration of a refinement edge (bold) of which the refinement is caused by two different overlap processors. 161
4.5.12 Illustration of a refinement edge (bold) where the neighbour elements are owned by different overlap processors. 163
4.5.13 Illustration of the re-sorting of the new nodes during a refinement step in 3-D. 170
4.5.14 Illustration of the owning of the new elements in 3-D. 170
4.6.1 Illustration of the change of node positions by re-refinement of the mesh: a) original mesh, b) refined mesh with coarsened elements, c) re-refinement of mesh, if the refinement history has not been stored. 176
4.7.1 Illustration of the processor borders and the processor widths for a circular domain for $np = 8$ processors. 180
4.7.2 Comparison of the cost of the preparatory step for the data exchange for the easy and our own method. 181
5.3.1 Polynomial $f = x^6 + y^6$ on unit circle. 189
5.3.2 Sugar-loaf function $f = e^{-32(x^2+y^2)}$ on unit circle. 190
5.4.1 Solution of component ω in the a) first and b) sixth computation cycle. . . 192
5.4.2 Resulting grids and estimated errors computed on these grids from pure mesh refinement: a) original grid, b) grid in cycle 2, c) grid in cycle 3. . . 194

5.4.2 (Continued) Resulting grids and estimated errors computed on these grids from pure mesh refinement: d) grid in cycle 4, e) grid in cycle 5, f) grid in cycle 6. 195
5.5.1 Type of grid for 4×1 domain. 197
5.6.1 Domains for the example with dividing line and initial grid. 204
5.6.2 Solution for the example with dividing line for the four cycles. 204
5.6.3 Domains for the example with dividing line in the second cycle. 206
5.6.4 Domains for the example with dividing line in the third cycle. 207
5.6.5 Domains for the example with dividing line in the fourth cycle. 207
5.6.6 Relative estimated error of solution component w for the example with dividing line for the four cycles: a) cycle 1, b) cycle 2, c) cycle 3, d) cycle 4. 208
5.7.1 Domains for the example with sliding dividing line for a) cycle 1, b) cycle 2, c) cycle 3. 210
5.7.2 Estimated error of component ω for the example with sliding dividing line for a) cycle 1, b) cycle 2, c) cycle 3. 211
5.7.3 Solution component ω for the example with sliding dividing line. 213
5.7.4 Cycle 4 for the example with sliding dividing line: a) solution component ω, b) error estimate of ω. 214
5.7.5 Cycle 5 for the example with sliding dividing line: a) solution component ω, b) error estimate of ω. 215
5.8.1 Solution u of the 3-D example for the section plane $x = 0$. 218
5.8.2 Average edge length between a node and its neighbour nodes for the 3-D example for a) cycle 1, b) cycle 2, c) cycle 3, section plane $x = 0$. 219

List of Listings

1 Code for communication pattern used to send message lengths to the right. . 57
2 Code for communication pattern used to send messages to the right. 60
3 Code for determination of refinement nodes. 62
4 Code for entering refinement elements because of the error into *refel*. 64
5 Code for determination of local refinement elements due to the cascade rules. 67
6 Code for neighbour element search. 68
7 Code for twin node search for dividing lines. 72
8 Code for inserting data into arrays *sndlto*, *sndlct* and *lsent*. 81
9 Code for arrays *nb* and *nbadd*. 89
10 Code for the update of the *nek* array for refinement elements. 97
11 Code for the update of the *nek* array for neighbour elements. 98
12 Code for the determination of new external boundary nodes and sliding dividing line nodes in 2-D. 99

13 Code for the determination of new external boundary nodes and sliding dividing line nodes in 3-D. 109
14 Code for entering of the new dividing line node data into array *tnod* (first node). 115
15 Code for entering of the new dividing line node data into *tnod* (twin nodes). 116
16 Code for inserting the edge information into *ptl*. 128
17 Code for the processing of the *ptl* array that has been sent to the right and received back with the global node numbers of the mid-points. 136
18 Code for the processing of the *ptl* array received from the left. 139
18 Code for the processing of the *ptl* array received from the left (continued). . 140
19 Code for the processing of the received data for neighbour elements of refinement elements owned by an overlap processor. 146
20 Code for the elimination of the overlap data for the *nnr* array. 159
21 Code for the elimination of the overlap data for the *bnod* array. 160
22 Code for the computation of the array *lprocs*. 166
23 Code for the inserting of the data into *etl*. 167

List of Rules

1 Refinement elements 63
2 Difference of refinement stages 65
3 Refinement element processor 76
4 Owning of elements 76
5 Owning of edges 124
6 Generation of new nodes 125

List of Tables

3.4.1 Length of arrays for difference and error formulas for the orders $q = 2, 4, 6$. 35
4.1.1 Array *tids* with physical processor numbers. 53
4.1.2 Parameter list of the communication routines. 53
4.1.3 List of external procedures. 54
4.1.4 Example of storage scheme of send buffer *sndbuf*, i is the cycle number. . . 56
4.2.1 Shape of the logical array *refel*. *true* means that the corresponding element in the corresponding refinement stage is a refinement element. A *true* can occur only in the refinement stage of the corresponding element. The element 4 is not refined. 63
4.3.1 Length and type of arrays for refinement cascade. 78
4.3.2 Illustration of the contents of array *nenr*. 79

4.3.3 Illustration of the arrays a) *sndlto* and *sndlct*, b) *sndrto* and *sndrct* for processor ip, i is number of entry, j identifies number of processor to send data to. 80
4.4.1 Illustration of the contents of array *rtl* in 2-D (edge information). 92
4.4.2 Illustration of the contents of array *nnr*. 93
4.4.3 Update of the *nek* array information in 2-D for figure 4.4.6. 102
4.4.4 Illustration of the contents of array *rtl* in 3-D (edge information). 105
4.4.5 Use of array *lbnd* for assignment of boundary node property (3-D): new node is a) a boundary node, b) not a boundary node, see figure 4.4.9. 108
4.4.6 Update of the *nek* array information in 3-D. 108
4.4.7 dividing line property for the new elements in 2-D. 119
4.4.8 Illustration of the dividing line property in 3-D. 121
4.5.1 Illustration of the contents of column k of array *ptl* in 2-D, k is the column to be sent to processor $ip + k$. 126
4.5.2 Illustration of the array *nbrs*. 129
4.5.3 Illustration of the arrays p_{cnt} and *lp* for $np = 8$ processors. 131
4.5.4 Storage scheme of array *sndbuf* for inserting data of an element of processor ip_{own} in column col_{rel}. 144
4.7.1 Comparison of the cost of the preparatory step for the data exchange for the easy and our own method. 180
5.4.1 Results for pure mesh refinement on unit circle with order $q = 2$. 191
5.4.2 Time analysis for pure mesh refinement with order $q = 2$ (part one). 193
5.4.3 Time analysis for pure mesh refinement with order $q = 2$ (part two). 196
5.5.1 Characteristics of the 7 grids for the complete computation. After cycle 1 all elements are refined. 197
5.5.2 Results for seven different grids with sugar-loaf forcing function. CPU ref. is the CPU time for the refinement only. 199
5.5.3 Time analysis for pure mesh refinement with order $q = 2$. 200
5.5.4 Characteristics of the 7 grids for the mesh refinement. 202
5.5.5 CPU time for the mesh refinement for the seven grids. 202
5.6.1 Results for pure mesh refinement with dividing line (part one). 205
5.6.2 Results for pure mesh refinement with dividing line (part two). Domain 1: upper domain, domain 2: lower domain. 206
5.7.1 Results for pure mesh refinement with sliding dividing line (part one). 212
5.7.2 Results for pure mesh refinement with sliding dividing line (part two). Domain 1: upper domain, domain 2: lower domain. 213
5.8.1 Results for pure mesh refinement for 3-D example (part one). 217
5.8.2 Results for pure mesh refinement for 3-D example (part two). 218

Lebenslauf

Name:	Torsten Dirk Adolph
Geboren:	am 07. Mai 1971 in Marbach/Neckar
Familienstand:	verheiratet
1977 – 1981	Grundschule in Oberstenfeld
1981 – 1990	Herzog-Christoph-Gymnasium in Beilstein
1990	Abitur
1990 – 1991	Grundwehrdienst
1991 – 1997	Studium der Technomathematik an der Universität Karlsruhe (TH)
November 1997	Diplomprüfung
Nov. 1997 – Februar 2005	wissenschaftlicher Angestellter in der Forschungsgruppe „Numerikforschung für Supercomputer“ am Rechenzentrum der Universität Karlsruhe (TH)
Oktober 2002 – Juni 2005	Anfertigung der Dissertation *The Parallelization of the Mesh Refinement Algorithm in the Finite Difference Element Method Program Package* am Institut für Algorithmen und Kognitive Systeme der Fakultät für Informatik der Universität Karlsruhe (TH)