Kommunikation auf verschiedenen Parallelrechnern, Auswirkung auf die Programmierung

Hartmut Häfner und Willi Schönauer
Rechenzentrum Universität Karlsruhe
Postfach 6980, D–76128 Karlsruhe, Deutschland

Zusammenfassung

Die Kommunikationsnetze von Parallelrechnern haben Auswirkungen auf die parallele Programmierung. Sowohl durch Hardware bestimmte Parameter des Netzwerkes wie Bandbreite, Startup-Zeit und Topologie als auch durch Systemsoftware bestimmte Parameter wie Synchronität bzw. Asynchronität von Nachrichten und die Bereitstellung von Kommunikationssoftware sind von Interesse für den Programmentwickler und von ihm bei der Programmerstellung zu berücksichtigen. In der vorliegenden Arbeit werden all diese Parameter unter besonderer Berücksichtigung der Programmierung erläutert.

1 Einleitung

Wie der Titel dieser Veröffentlichung schon andeutet, soll an dieser Stelle einzig und allein die Auswirkung der Kommunikation bei Parallelrechnern auf die Erstellung paralleler Programme besprochen werden. Unerwähnt bleiben hier all die Überlegungen, die notwendig sind, um ein paralleles Programm auf einem einzelnen Prozessor effizient zu implementieren. Bei allen nachfolgenden Überlegungen sollte jedoch nicht vergessen werden, daß nur dann eine hohe Leistung von einem parallelen Programm erwartet werden kann, wenn auf jedem einzelnen Prozessor ein hoher Datendurchsatz erreicht wird.

Mehrere Arten der parallelen Programmierung sind möglich. Es gibt parallele Programme für SIMD-Rechner (Single Instruction Multiple Data). Diese unterscheiden sich wesentlich von Programmen für MIMD-Rechner (Multiple Instruction Multiple Data), wobei für die letzteren Rechner bei einer einfachen Klassifikation zwei Programmierstile genannt werden können:

- „message passing"-Programmierstil,
- „(virtual) shared memory"-Programmierstil.

Alle hier vorgestellten Überlegungen und Erkenntnisse sind nur für den „message passing"-Programmierstil gültig. Nur hier hat der Benutzer volle Kontrolle über Prozessoren und Daten.

Um große numerische Probleme in einer akzeptablen Zeit lösen zu können, müssen serielle Programme im Hinblick auf ihre Leistung optimiert sein. Dabei sollten die Programme portabel , d.h. mit geringem Aufwand auf einem beliebigen anderen Rechner zum Laufen gebracht werden können, und wartbar sein. Die genannten Anforderungen sind natürlich auch an parallele Programme zu stellen und betreffen insbesondere die Kommunikation zwischen den Programmen, die unabhängig voneinander auf den einzelnen Prozessoren laufen.

Anmerkung: Beim „message passing“-Programmiermodell laufen auf jedem Prozessor des allokierten Prozessorfeldes vollständige Programme - d.h. eigenständige Prozesse -, die alle einzig und allein auf den lokalen, im zugehörigen Speicher abgelegten Daten arbeiten können. Die Kommunikation erfolgt über Systemaufrufe, die ein Senden oder Empfangen von Anwendungsfeldern zu oder von dem Speicher eines anderen Prozessors veranlassen.

Bei der Erstellung paralleler Software sind somit 3 Punkte besonders zu beachten:

- Effizienz der parallelen Software,
- Portabilität der parallelen Software und
- Wartbarkeit der parallelen Software.

Welche Probleme bei der Berücksichtigung dieser Punkte auftreten und wie diese Anforderungen an die Software erfüllt werden können, darauf wird in den folgenden 3 Kapiteln eingegangen.

2 Die Effizienz von paralleler Software

Das oberste Prinzip, das die Kommunikation zwischen voneinander unabhängigen laufenden Prozessen betrifft, lautet:

- Verbergen der Kommunikation hinter der Berechnung (latency hiding).

Das Verbergen der Kommunikation ist nur möglich, wenn asynchrone Kommunikation vom Parallelrechner-Hersteller unterstützt wird. Hierbei ist allerdings noch zu klären, was asynchrone Kommunikation heißt (siehe 2.1.2). Insgesamt bestimmen 3 Einflußfaktoren den Anteil der Kommunikation, der hinter der Berechnung verborgen werden kann. Diese sind:

- das Verhältnis der Kommunikationsleistung zu der Berechnungsleistung eines Parallelrechners,
- das bestmögliche Ausnutzen der asynchronen Kommunikation durch eine geeignete Zerlegung des Problems und
- das „volume-to-surface“-Verhältnis der gestellten Aufgabe bzw. des gewählten Algorithmus.

Die beiden letzteren Faktoren sind abhängig von dem Algorithmus und dem Programmierer, der einen Algorithmus und eine zugehörige Problemzerlegung wählt. An dieser Stelle soll nicht auf die algorithmische Seite beim Erreichen von Effizienz eingegangen werden, sondern nur auf das Wechselspiel zwischen Rechnerarchitektur und Effizienz.

2.1 Auswirkungen der Rechnerarchitektur auf die Effizienz

Natürlich gilt die Aussage, je schneller die Kommunikation im Vergleich zur Berechnung ist, desto besser läßt sich die Kommunikation hinter der Berechnung verstecken und damit die Effizienz erhöhen. Es ist auch richtig, daß ein Programmierer keinen Einfluß auf die Schnelligkeit des Kommunikationsnetzes hat und auch nicht auf die Unterstützung asynchroner Kommunikation durch den Rechnerhersteller. Dennoch ist es von Vorteil für ihn, die Parameter des Kommunikationsnetzes zu kennen und zu wissen, was ein bestimmter Rechnerhersteller unter asynchroner Kommunikation versteht. Leider ist es zur Zeit so, daß verschiedene Rechnerhersteller asynchrone Kommunikation unterschiedlich definieren.

2.1.1 Die Kommunikationsgeschwindigkeit verschiedener Parallelrechner

In der nachfolgenden Tabelle finden Sie die für die Programmierung wichtigen Parameter des Kommunikationsnetzes. Die Spalten „Architektur“ (beschreibt die Topologie des Kommunikationsnetzes eines Parallelrechners) und „Routing“ (beschreibt, wie die Daten durch das Kommunikationsnetz bewegt werden) sind nicht für die Effizienz eines Programmes von Bedeutung; sie spielen bei der Portabilität und Wartbarkeit eines Programmes eine Rolle. In den Spalten „Bandbreite“ und „Startup-Zeit“ finden Sie jeweils die vom Hersteller angegebenen Höchstwerte und, wo verfügbar, die mittels „Ping-Pong“-Test gemessenen Werte.

Man sieht, daß die theoretische Bandbreite von 20–400 MByte/sec und die gemessene von 6,4–106 MByte/sec bei den „im Handel erhältlichen“ Parallelrechnern variiert. Auch die Startup-Zeiten unterscheiden sich von Rechner zu Rechner erheblich. Am interessantesten für den Programmierer sind die Spalten „Bandbreite, gemessen, Zyklen pro Operand“ und „Startup-Zeit, gemessen, in Zyklen“. Die vordere der beiden Spalten besagt, wieviele Maschinenzyklen man benötigt, um einen 64-Bit Operanden von einem Speicher eines Prozessors in einen anderen zu übertragen. Nahezu alle angegebenen Rechner benötigen 1 Maschinenzyklus, um einen 64-Bit Operanden aus dem lokalen Speicher zu holen. Um einen 64-Bit Operanden aus einem entfernten Speicher in den lokalen Speicher zu übertragen, werden z.B. auf dem Cray T3D 11,3 [4] und auf der IBM SP1 78 Maschinenzyklen [5] zusätzlich benötigt unter der Voraussetzung, daß die Anzahl der durch ein Senden/Empfangen geholten Operanden groß genug ist, um die gemessene Bandbreite bei der Kommunikation zu erreichen. Wenn man einzelne 64-Bit Operanden überträgt, dann benötigt man dafür ungefähr die in der hinteren der beiden Spalten angegebene Anzahl an Maschinenzyklen. Man sieht, daß kurze Nachrichten zu einem sehr hohen Kommunikations-Overhead führen und daß selbst bei langen Nachrichten ein Verstecken der Kommunikation hinter der Berechnung nur dann möglich ist, wenn ein Mehrfaches an Berechnungen (d.h. Floating-Point Operationen) während der Übertragung der Nachrichten durchgeführt werden kann.

2.1.2 Synchrone/asynchrone Kommunikation

Wie bereits kurz erwähnt, haben verschiedene Rechnerhersteller unterschiedliche Definitionen für die Begriffe synchrone/asynchrone Kommunikation. Um die Unterschiedlichkeit der Begriffsbildung zu verdeutlichen, zeigen wir für ausgewählte „message passing“-Softwaresysteme die

Rechner	Architektur	Routing	Bandbreite theor.	Bandbreite gemessen		Startup-Zeit theor.	Startup-Zeit gemessen	
			$\frac{MB}{sec}$	$\frac{MB}{sec}$	Zyklen pro Operand	μs	μs	Zyklen
iPSC/860	Hypercube	'telefon line'	2,8	2,8	114		175	7000
nCUBE2S	Hypercube	'telefon line'	2,8	2,8	71	120	120	3000
KSR-1	Hierachie von Ringen	'Allcache-Engine'	30	8–10	16	7,5–28,5	48	960
Parsytec GC	3-D Gitter	'Wormhole'	20			10–15		
CM-5	'Fat tree'	Pakete 'virtual links'	20	8,6	15	3–7	86	1376
Paragon	2-D Gitter	'Wormhole'	200	18	22	30	140	7000
Meiko CS-2	'Fat tree'	'Wormhole'	50	42,5	9,4	8–10	78	3900
VPP-500	Crossbar	Punkt-zu-Punkt	400	355	2,3	3–6	31	3100
Cray T3D	3-D Torus	Kombination aus 'Virtual-Cut-Through' und 'Wormhole'	300	106	11,3	< 1	1,5	225
Convex SPP-1	Crossbar / Ring	'Cachelines on request'	64/21			3/0,5		
IBM SP1	'M-stage Switch'	'Wormhole'	20	6,4	78	0,5	35	2187

Tabelle 1: Parameter des Kommunikationsnetzes von verschiedenen Parallelrechnern

Wege der Nachrichten auf und erklären, welche Wegstrecke von Nachrichten synchron bzw. asynchron durchlaufen werden kann und welche Wegstrecke von dem jeweiligen Softwaresystem-Hersteller zur Kennzeichnung einer Nachrichtenübertragung als synchron oder asynchron benutzt wird (durch Unterstreichungen in Tabelle 2).

Nehmen wir z.B. das Softwaresystem NX/2 von Intel, das sowohl auf dem iPSC/860 als auch auf dem Paragon läuft [1]. Mit Hilfe von Bild 1 und Tabelle 2 sieht man, daß beim Senden einer Nachricht die Daten immer in einen Systempuffer kopiert werden, bevor sie asynchron verschickt werden. Das Kopieren der Daten von dem Anwendungspuffer in den Systempuffer kann synchron oder asynchron erfolgen durch die Wahl unterschiedlicher Kommunikationsroutinen. Für Intel findet ein asynchrones Senden statt, wenn das Kopieren asynchron erfolgt. Das eigentliche Senden der Daten findet immer asynchron statt, hat aber keinen Einfluß darauf, ob von Intel ein Sendevorgang als synchron oder asynchron bezeichnet wird. Das Gleiche gilt für das Empfangen von Daten. Wiederum wird ein Empfangen von Daten nur dann von Intel als

Software-sytem	Weg der Nachricht	(a) synchrones 'send' bedeutet	(a) synchrones 'receive' bedeutet
VERTEX	$\boxed{\text{1a}}$,$\boxed{\text{2}}$,$\boxed{\text{3a}}$	$\boxed{\text{1a}}$ synchron $\boxed{\text{2}}$ asynchron	$\boxed{\text{2}}$ asynchron $\boxed{\text{3a}}$ synchron
NX/2	$\boxed{\text{1a}}$,$\boxed{\text{2}}$,$\boxed{\text{3a}}$ oder $\boxed{\text{3b}}$	$\boxed{\text{1a}}$ (a)synchron $\boxed{\text{2}}$ asynchron	$\boxed{\text{2}}$ asynchron $\boxed{\text{3a}}$,$\boxed{\text{3b}}$ (a)synchron
PVM	$\boxed{\text{1a}}$ oder $\boxed{\text{1b}}$,$\boxed{\text{2}}$,$\boxed{\text{3a}}$	$\boxed{\text{1a}}$ synchron $\boxed{\text{2}}$ asynchron	$\boxed{\text{2}}$ asynchron $\boxed{\text{3a}}$ synchron
PARMACS	$\boxed{\text{1a}}$ oder $\boxed{\text{1b}}$,$\boxed{\text{2}}$,$\boxed{\text{3a}}$ oder $\boxed{\text{3b}}$	$\boxed{\text{1a}}$ synchron $\boxed{\text{2}}$ asynchron	$\boxed{\text{2}}$ asynchron $\boxed{\text{3a}}$ synchron

Tabelle 2: Wege von Nachrichten und deren Synchronität/Asychronität für verschiedene „message passing“-Softwaresysteme

asynchron bezeichnet, wenn das Kopieren der Daten aus dem Systempuffer asynchron erfolgt.

Unter VERTEX auf dem nCUBE 2 läuft nur das eigentliche Senden/Empfangen der Daten asynchron ab. Das Kopieren der Daten in und aus dem Systempuffer erfolgt immer synchron. Die Firma nCUBE bezeichnet dabei ihre Sende- und Empfangs-Routinen als asynchron.

Wenn man eine Sende/Empfangs-Routine dann als asynchron bezeichnet, wenn das eigentliche Verschicken der Daten (Weg $\boxed{\text{2}}$ in Bild 1) asynchron zu auf die Kommunikationsroutinen folgenden Berechnungen im Programm laufen kann, dann bietet eigentlich jede „message passing“-Systemsoftware asynchrone Kommunikation. Selbst PVM und PARMACS [2,3] auf Workstations verfügen bei dieser Definition von Asynchronität über einen von der Unabhängigkeit der Netzkarte profitierenden Anteil an asynchroner Kommunikation.

Welchen Anteil die Zeit für die Kopiervorgänge (Wege $\boxed{\text{1a}}$,$\boxed{\text{3a}}$ in Bild 1) an der Gesamtzeit für die Kommunikation hat, wäre eine Untersuchung wert. Es steht aber zu vermuten, daß dieser Zeitanteil relativ gering ist, da das Umkopieren von Daten innerhalb eines Speichers viel schneller erfolgt als das Verschicken von Daten (siehe auch 2.1.1). Wenn der Zeitanteil klein genug ist, kann auf das asynchrone Kopieren von Datenfeldern verzichtet werden. Aus der Sicht des Anwendungsprogrammierers hat dieser Verzicht zwei große Vorteile:

- geringere Fehlerwahrscheinlichkeit des „message passing“-Programmes,
- Erstellung portabler Software leichter erreichbar (siehe auch Kapitel 3).

2.2 Programmiermethodik zum Erreichen von Effizienz

Zwei Punkte sind beim Erstellen von „message passing“-Programmen für das Erreichen von Effizienz von Bedeutung (die algorithmische Seite des Problems wird hier nicht betrachtet):

- Wahl der effizientesten Stellen zum Aufruf der Kommunikationsroutinen und
- eine optimale Abbildung der Kommunikationsstruktur eines Programmes auf die Topologie des Parallelrechners.

Für den ersten Punkt gibt es einfache Kochrezepte. Zu sendende Daten sollen, sobald sie zur Verfügung stehen, verschickt werden, d.h. die Sende-Routine sollte direkt nach der Berechnung von zu sendenden Daten im Code plaziert werden. Das Empfangen der Daten sollte dagegen so spät wie möglich erfolgen, d.h. die Empfangs-Routine sollte direkt vor dem Codestück plaziert werden, in dem die empfangenen Daten benötigt werden. Bei einem asynchronen Empfangen (Intel-Terminologie!) auf Intel-Rechnern muß die letzte Regel leicht abgewandelt werden. Die asynchrone Empfangs-Routine, die ein Empfangen der Daten nur anstößt und sich anschließend sofort beendet, sollte so früh wie möglich im Code plaziert werden. An die Stelle, an der bei anderen Parallelrechnern die Empfangs-Routine plaziert wird, muß nun die „wait“-Routine, die das Programm veranlaßt, zu warten, bis die zu empfangenden Daten im Anwendungspuffer zur Verfügung stehen, geschrieben werden. Schon hier erhält man einen ersten Eindruck davon, daß das Ausnutzen des asynchronen Sendens und Empfangens (Intel-Terminologie!) auf Intel-Rechnern die Erstellung von portabler Software erschwert.

Eine sehr häufig vorzufindende Kommunikationsstruktur eines Programmes ist der „nearest neighbor ring“. Bei dieser Struktur werden die Daten im Ring immer zum nächsten Nachbarn geschickt. Hat ein Prozessor die logische Prozessornummer i ($1 \leq i \leq$ Anzahl der allokierten Prozessoren), dann sendet er seine Daten immer entweder zu dem Prozessor mit der logischen Prozessornummer $i-1$ oder zu dem mit der logischen Prozessornummer $i+1$. Ein einfaches Beispiel für den „nearest neighbor ring“ ist eine Gruppe von Personen, die an einem runden Tisch sitzt und ein Würfelspiel spielt. Dabei wird üblicherweise der Würfelbecher ($\equiv$ den Daten beim Parallelrechner) von jedem Spieler immer entweder nach links oder nach rechts an einen der beiden Tischnachbarn weitergegeben. Das Problem besteht nun darin, eine optimale Abbildung von logischen Prozessornummern auf physikalische Prozessornummern oder „Process-IDs“ zu finden. Die Abbildung ist notwendig, da die Kommunikationsroutinen entweder physikalische Prozessornummern oder „Process-IDs“ zur Adressierung benutzen. Diese Abbildung sollte dabei in dem Sinne optimal sein, daß logisch benachbarte Prozessoren auch physikalisch benachbart sind, wodurch Kommunikationspfade über mehrere Prozessoren hinweg und unnötige Leitungsblockierungen vermieden werden.

Beispielhaft wird hier der Weg zur optimalen Abbildung beim „nearest neighbor ring“ für den Intel Paragon, d.h. für ein 2-dimensionales Gitter, aufgezeigt. Bei der Allokierung eines Prozessorfeldes auf dem Paragon gehen wir davon aus, daß nur rechteckige Gebiete erlaubt sind und in mindestens einer Richtung eine gerade Anzahl Prozessoren vorliegt. Die physikalischen Prozessornummern werden mit 0 beginnend von links oben fortlaufend von dem Betriebssystem vergeben (siehe Bild 2).

In Bild 2 ist ein möglicher Weg durch das physikalische Netzwerk eingezeichnet, der zu einer optimalen Abbildung, d.h. benachbarte Prozessoren auf logischer Ebene sind auch physikalisch benachbart, im „nearest neighbor ring“ führt. Anhand von Tabelle 3 läßt sich dies leicht nachvollziehen.

logische Prozessornr.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
physikalische Prozessornr.	0	1	2	3	7	6	5	9	10	11	15	14	13	12	8	4

Tabelle 3: Eine optimale Abbildung von logischen auf physikalische Prozessornummern für den „nearest neighbor ring“ und ein 2-dimensionales 4x4-Gitter

Für die Effizienz eines Programmes ist die angesprochene Abbildung wichtig. Fast noch wichtiger ist aber, daß der Programmentwickler ein „message passing“-Programm basierend auf logischen Prozessornummern entwerfen kann, wobei er die bei den Kommunikationsroutinen benötigten physikalischen Prozessornummern über die Abbildungstabelle erhält.

3 Die Portabilität von paralleler Software

Software soll in diesem Zusammenhang portabel genannt werden, wenn, um das jeweilige Programm auf einem anderen Parallelrechner zum Laufen zu bringen, nur ein paar wenige Unterprogramme ausgetauscht werden müssen. Zwei Forderungen finden sich auf dem steinigen Weg zu portabler Software; die erste, die Abstraktion von physikalischen Prozessoren, muß erfüllt sein, während die zweite, die eigene Definition einer portablen Kommunikationsschnittstelle, bei in Kauf nehmen einiger Nachteile umgangen werden kann.

3.1 Abstraktion von physikalischen Prozessoren

Wie schon am Ende von Kapitel 2.2 angesprochen, sollte ein „message passing“- Programm immer allein basierend auf logischen Prozessoren entworfen werden. Der Bezug zur Realität, d.h. zu physikalischen Prozessornummern, schafft die Abbildungstabelle (siehe Tabelle 3), die in einer eigenständigen Prozedur erstellt werden kann. Durch Auswechseln dieser Prozedur läßt sich leicht eine optimale Abbildungstabelle für verschiedene Netzwerk-Topologien installieren; als weiterer Vorteil bleibt die Verknüpfung der abstrakten Welt (logische Prozessornummern) mit der realen Welt (physikalische Prozessornummern oder „Process-IDs“) auf diese eine Prozedur beschränkt.

3.2 Portabilität der Kommunikationsschnittstelle

Wie aus Tabelle 4 zu ersehen, verfügen nicht alle MIMD-Parallelrechner über eigene Kommunikations-Bibliotheken. Dagegen unterstützen alle hier aufgeführten Parallelrechner die PARMACS-Kommunikationssoftware und nahezu alle PVM3.x (Parallel Virtual Machine). Da PVM3.x bei geringem Portierungsaufwand von jedem Benutzer auf jedem Rechner mit dem Betriebssystem UNIX (oder Derivat) installiert werden kann, heißt **no** oder **?** nicht, daß PVM3.x auf diesen Rechnern nicht läuft, sondern nur, daß PVM3.x von dem jeweiligen Rechnerhersteller nicht unterstützt wird oder keine eindeutige Aussage des Rechnerherstellers vorliegt.

Rechner	Unterstützte 'message-passing'-Kommunikationssoftware		
	PARMACS (Version 6.0)	PVM 3.x	Name der proprietären Bibliothek
iPSC/860	yes	yes (MPP)	NX/2
nCUBE2	yes (Version 5.1)	yes	VERTEX
KSR-1	yes	yes	–
Parsytec GC	yes (Version 5.1)	?	Parix
CM-5	yes	yes (MPP)	CMMD
Paragon	yes	yes (MPP)	Interface to NX/2
Meiko CS-2	yes (Version 5.1)	yes	Elan (only low level communication)
VPP-500	yes	no	–
Cray T3D	yes	yes (extended)	–
Convex SPP-1	yes (Version 5.1)	yes	–
IBM SP1	yes	yes	PVMe (extended PVM 2.3)

MPP: Message Passing Port

Tabelle 4: Auf verschiedenen Parallelrechnern verfügbare „message-passing"-Kommunikationssoftware

Wenn Sie also die PARMACS- oder PVM-Routinen in Ihrem Programm benutzen, erreichen Sie bezüglich der Kommunikationsschnittstelle Portabilität. Dagegen läßt sich ein Verlust an Kommunikationsgeschwindigkeit in Folge des zusätzlichen Software-Overheads nicht vermeiden. Dies trifft in besonderem Maße auf PVM3.x zu. PVM3.x unterstützt zwar theoretisch über ein „Message Passing Port" die unmittelbare Abbildung seiner Kommunikationsroutinen auf die proprietären Kommunikationsroutinen einiger Parallelrechner; wenn man aber z.B. die Wege der Nachrichten mit deren zugehörigem Attribut synchron/asynchron unter PVM mit denen unter NX/2 von Intel vergleicht, sieht man, wie schwierig es in der Paxis sein dürfte, die Kommunikationsroutinen von PVM3.x auf die von NX/2 ohne größeren Software-Overhead abzubilden. So gibt es zum gegenwärtigen Zeitpunkt keine auf einen Parallelrechner portierte PVM3.x-Version, die nur annähernd die Bandbreite und Startup-Zeit der proprietären Kommunikations-Bibliothek erreicht.

Ein zweiter Weg, Portabilität zu erreichen, ist, eine für alle Parallelrechner einheitliche Kommunikationsschnittstelle zu schaffen. Durch das Auswechseln von ein paar (4-5) selbst implementierten Kommunikationsroutinen, deren wesentliches Merkmal die erwähnte einheitliche Kommunikationsschnittstelle ist, lassen sich die proprietären Kommunikationsroutinen der Parallelrechner, soweit vorhanden, einsetzen. Da die Parameter der wesentlichen proprietären Kommunikationsroutinen aller Parallelrechner die gleiche Bedeutung haben und ihre Anzahl auf 5–6 beschränkt ist, läßt sich eine einheitliche Kommunikationsschnittstelle mit 5–6 Parametern definieren. Der Vorteil dieser Kommunikationsschnittstelle liegt einerseits darin, daß auch (einfache) PVM- und PARMACS-Routinen eingebunden werden können, andererseits darin, daß bei Benutzung von proprietären Kommunikationsroutinen die größtmögliche Effizienz erreicht wird. Der Nachteil besteht darin, daß viele Merkmale von Kommunikations-Bibliotheken nicht genutzt werden können. So bietet z.B. PVM die Möglichkeit, Elemente und Felder unterschiedlicher Datentypen gemeinsam in den Puffer der „system area“ hineinzupacken, von wo aus der Puffer dann versendet werden kann. Auf dieses Merkmal muß verzichtet werden, da die anderen Kommunikations-Bibliotheken diese Möglichkeit nicht unterstützen. Allgemein gilt, daß, sobald ein Merkmal von einer einzigen Kommunikations-Bibliothek nicht unterstützt wird, dieses über die einheitliche Kommunikationsschnittstelle nicht genutzt werden kann.

Im folgenden sollen die 5–6 Parameter der Kommunikationsschnittstelle kurz erläutert werden. <snd_tid> bzw. <rcv_tid> ist die physikalische Prozessornummer oder „Process-ID“ des sendenden bzw. empfangenden Prozessors oder Prozesses. <msgtyp> ist eine Integer-Variable, mit der eingehende Nachrichten identifiziert werden können (siehe Kapitel 4). <msg> ist ein Feld der Länge <msglen> vom Typ *char*, das das zu übertragende Feld beinhaltet. <msglen> ist die Länge des Feldes. Alle bisherigen Parameter sind Eingabeparameter. Der 5. Parameter - <error> - ist ein Ausgabeparameter und zeigt an, ob die aufgerufene Kommunikationsroutine erfolgreich beendet wurde. Der 6. Parameter - <sync> - kann derzeit nur von der NX/2-Software von Intel genutzt werden und ist demzufolge (momentan) nicht unbedingt erforderlich. <sync> wird von NX/2 ein Synchronisationsvariablenwert zugewiesen, der bei asynchroner Kommunikation (Intel-Terminologie!) zur Synchronisation von asynchronen Sende- und Empfangs-Routinen verwendet wird.

4 Die Wartbarkeit von paralleler Software

Eine ebenso häufige wie (bezogen auf die Auffindbarkeit) unangenehme Fehlerquelle stellt die nicht eindeutige Vergabe von „message types“ dar. So bestimmt z.B. bei dem Empfang zweier Nachrichten, die von verschiedenen Prozessoren bei gleichem Wert der <msgtyp>-Variablen kommen, die zufällige Reihenfolge des Eintreffens der Nachrichten darüber, in welches Anwendungsfeld die Daten kopiert werden. Dies gilt es natürlich zu verhindern. Aber wie erreicht man eine eindeutige Vergabe von <msgtyp>-Variablenwerten?

Dazu ist es wiederum notwendig, einen Blick auf die Rechnerarchitekturen zu werfen. Alle Routing-Verfahren („Wormhole“, „virtual links“, usw.) gewährleisten eine zeitliche Ordnung bei dem Versenden von Daten zwischen zwei festen Prozessoren, d.h. eine Nachricht, die als erste bzw. zweite usw. versand wurde, kommt auch als erste bzw. zweite usw. bei dem Zielprozessor an.

Dies bedeutet, daß die Werte von <msgtyp>-Variablen in Kommunikationsaufrufen zwischen zwei festen Prozessoren nicht verändert werden müssen, solange die Empfangs-Routinen die gleiche Aufrufreihenfolge haben wie die zugehörigen Sende-Routinen, d.h. keine über Kreuz gehenden Beziehungen im Code zwischen Sende- und Empfangs-Routinen auftreten. Aber wie erreicht man nun, daß der Wert von <msgtyp>-Variablen bei dem Empfang von Nachrichten von verschiedenen Prozessoren nie gleich ist?

Durch die Nachbildung des „virtual links"-Konzept von Parsytec läßt sich dies sicher und einfach erreichen. Unter Parix von Parsytec lassen sich durch Systemaufrufe „virtual links" zwischen jeweils zwei beliebigen Prozessoren installieren, die über eine Variable angesprochen werden. Benützt man für eine Kommunikation eine dieser Variablen, dann steht über die vom Betriebssystem softwaremäßig installierten „virtual links" fest, zu bzw. von welchem Prozessor die Nachricht geschickt wird bzw. kommt. Da Nachrichten nur zu Prozessoren, zu denen ein „virtual link" installiert ist, geschickt werden können, lassen sich softwareseitig Netzwerk-Topologien nachbilden. So kann z.B. ein „nearest neighbor ring" installiert werden. Der Vorteil ist, daß man nur innerhalb der Ring-Struktur kommunizieren kann, d.h. keine Fehler dadurch auftreten können, daß man Daten (aus Versehen) an einen nicht benachbarten Prozessor schickt.

Die Nachbildung des „virtual link"-Konzeptes erfolgt über die Deklarierung eines zweidimensionalen <msgtyp>-Variablenfeldes. Der erste Feldindex repräsentiert den sendenden Prozessor, der zweite den empfangenden Prozessor. So könnte die Nachbildung eines „nearest neighbor ring" auf 4 Prozessoren, wobei Daten im Ring nur zu Nachbarn mit einer höheren logischen Prozessornummer verschickt werden dürfen, über ein Variablenfeld folgendermaßen aussehen: msgtype(1,2)=12, msgtype(2,3)=23, msgtype(3,4)=34, msgtype(4,1)=41. Nach der Deklarierung der <msgtyp>-Variablenfelder mit paarweise verschiedenen Variablenwerten wird im Programm beim Aufruf einer Sende-Routine einfach die <msgtyp>-Variable msgtype(me,mod(me,p)+1) und beim Aufruf einer Empfangs-Routine msgtype(me-1,me) für me=2,3,4 bzw. msgtype(p,me) für me=1, wobei me die eigene logische Prozessornummer und p die Anzahl der Prozessoren ist, in die Kommunikationsroutine eingetragen.

Diese Nachbildung hat mehrere Vorteile:

- sie ist kompatibel zu dem „virtual link"-Konzept von Parsytec;
- sie kann komplett in <u>einem</u> Unterprogramm, das am Anfang des Programmes aufgerufen wird, durchgeführt werden; dies erhöht ganz wesentlich die Wartbarkeit paralleler Software;
- Nachbildungen unterschiedlicher Topologien können in einer Bibliothek abgelegt werden; dies führt zu verbesserter Portabilität.

5 Schlußfolgerungen

Vielfältige Wechselwirkungen bestehen zwischen den Kommunikationsnetzen und seinen Parametern wie Bandbreite, Startup-Zeit, usw. und der Erstellung von paralleler Software. So hängt die Effizienz paralleler Software von dem Grad der durch die Kommunikations-Software unterstützten Asynchronität der Kommunikationsroutinen und der Geschwindigkeit des Kommunikationsnetzes ab. Die Wege der Nachrichten bei synchroner/asynchroner Kommunikation und die darauf aufsetzenden Kommunikationsroutinen beeinflussen dagegen wiederum die Portabilität paralleler Software.

Zusammenfassend läßt sich sagen, daß durch eine ansatzweise objektorientierte Programmerstellung, bei der die von der Topologie des Netzwerkes abhängigen Parameter und wichtige Parameter der Kommunikationsroutinen (<msgtyp>-Variable) in Unterprogrammen am Anfang des Gesamtprogrammes gekapselt werden und damit bei der Erstellung des eigentlichen parallelen Algorithmus für den Programmentwickler transparent gehalten werden können, sowohl

- Effizienz als auch
- Portabilität und Wartbarkeit paralleler Software

erreicht werden kann.

Literatur

1. Paragon OSF/1, USER'S GUIDE, Intel Corporation.
2. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, PVM 3.0 USER'S GUIDE AND REFERENCE MANUAL, Oak Ridge, TN 37831-6357.
3. R. Hempel, H.-C. Hoppe, U. Keller, W. Krotz, PARMACS V6.0 Specification, 1993, Pallas GmbH, Brühl, Germany.
4. Wilfried Oed, The Cray Research Massively Parallel Processor System Cray T3D, 1993, Cray Research GmbH, München, Germany.
5. K.E. Gates, W.P. Petersen, A Technical Description of Some Parallel Computers, ETH Zürich, 1993.

Abbildung 1: Mögliche Wege einer Nachricht aus dem Speicher des Prozessors i in den Speicher des Prozessors j. stellt einen Datenpuffer dar. sind Wegmakierungen.

Abbildung 2: Knoten (•), physikalische und logische Prozessornummern und Netzwerk (—) eines rechteckigen 4x4-Prozessorfeldes mit eingezeichnetem „nearest neighbor ring“ (- - - -)