Technische Universität Chemnitz-Zwickau
Sonderforschungsbereich 393
Numerische Simulation auf massiv parallelen Rechnern

Wolfgang Rehm (Ed.)

Portierbare numerische Simulation auf parallelen Architekturen

Preprint SFB393/96-07

Einband der zum gleichnamigen Workshop erschienenen
Computer Achitecture Technical Reports
des Lehrstuhls Rechnerarchitektur/Informatik

Preprint-Reihe des Chemnitzer SFB 393

SFB393/96-07 **April 1996**

Contents

Preface

The workshop "Portierbare numerische Simulationen auf parallelen Architekturen" ("Portable numerical simulations on parallel architectures") was organized by the Faculty of Informatics/Professorship Computer Architecture at 18 April 1996 and held in the framework of the Sonderforschungsbereich (Joint Research Initiative) "Numerische Simulationen auf massiv parallelen Rechnern" (SFB 393) ("Numerical simulations on massiv parallel computers") (*http://www.tu-chemnitz.de/~pester/sfb/sfb393.html*)

The SFB 393 is funded by the German National Science Foundation (DFG).

The purpose of the workshop was to bring together scientists using parallel computing to provide integrated discussions on portability issues, requirements and future developments in implementing parallel software efficiently as well as portable on Clusters of Symmetric Multiprocessorsystems

I hope that the present paper gives the reader some helpful hints for further discussions in this field.

April 1996 Wolfgang Rehm

Benchmarking

Sven Schindler

svsc@informatik.tu-chemnitz.de
http://noah.informatik.tu-chemnitz.de/members/schindler/schindler.html

Zusammenfassung

Nach einer kurzen Erläuterung theoretischer Grundlagen werden stellvertetend die Benchmarks des Performance Database Servers sowie ein eigener Benchmark vorgestellt.Im weiteren werden Theorie und Probleme des parallelen Benchmarking behandelt.

1 Einführung

1.1 Historisches

Der Begriff **BENCHMARK** tauchte erstmals 1840 bei nordamerikanischen Landvermessern auf. Er bezeichnet eine Erhebung, welche als fester Punkt zur Landvermessung verwendet wurde.

Heute versteht man unter dem Begriff Benchmark einen Referenzpunkt, welcher die Leistungsbestimmung von verschiedenen Rechnern ermöglicht. Ein Beispiel für einen solchen Referenzpunkt sind die 1-MIPS der VAX–11/780.

Benchmarks sind ein wichtiges Hilfsmittel zur Leistungsbestimmung bzw. Leistungsvorhersage von Rechnersystemen.

1.2 Begriffe

Runtime(Laufzeit):

- Laufzeit eines Programm auf einem konkreten Rechner
- bildet Grundlage zur bestimmung anderer Performancegrößen

MIPS = **M**illion **I**nstructions **P**er **S**econd

MFLOPS = **M**illion **Fl**oating–Point **Op**erations Per **S**econd

1.3 Theoretische Grundlagen

1.3.1 Berechnung des MIPS–Wertes

Der MIPS–Wert berechnet sich, wie schon aus seinem Namen zu schlußfolgern, mittels Division der ausgeführten Befehle(in Million Instructions) durch die Laufzeit(in sec). Die Bestimmung der Anzahl der ausgeführten Befehle erfolgt meist durch sogenannte Befehlszähler.

1.4 Berechnung des MFLOPS–Wertes

Vorraussetzung für die Berechnung des MFLOPS–Wertes ist die Bestimmung der ausgeführten Floating–Point–Operations.Hierbei gibt das McMahon Schema eine Hilfe.

Nach dem McMohanSchema werden die einzelnen Arithmetikoperationen wie folgt gewichtet:

add,sub,mul	1 flop
div,sqr	4 flop
exp,sin ...	8 flop
if(x compare y)	1 flop

Nun kann der MFLOPS–Wert problemlos mittels Division der ausgeführten Floating–Point–Operations durch die Laufzeit ermittelt werden.

1.5 Zeitmessung

Während bei Singleprozessbetriebssystemen, wie z.B. DOS, die Zeitmessung fast problemlos möglich ist muß bei Multiprozessbetriebssystemen, wie z.B. UNIX, einiges beachtet werden.

So darf in diesem Fall keinesfalls die Systemzeit als Bewertungsgrundlage eingehen, vielmehr muß sich der Nutzer(Implementierer) mit dem Ablesen der Prozess– bzw. Userzeit vertraut machen.

2 Analyse der sequentielle Benchmarks des PDS

In diesem Kapitel sollen die auf dem PDS[1] benutzten Benchmarks vorgestellt werden. Fuer diese Benchmarks existieren auf dem oben genannten PDS eine Vielzahl von Testergebnissen. Als weiterer Vorteil kann die freie Erhältlichkeit der Quellen angesehen werden(im Gegensatz z.B. zu SPEC). Die meisten der hier vorgestellten Benchmarks sind, eventuell in leicht abgewandelter Form, ebenfalls Bestandteil anderer Benchmarksuiten.

2.1 Ergebnislisten

Die im Anhang beiliegende Ergbnisliste soll am Beispiel des Dhrystone–Benchmarks kurz erläutert werden. Es ist zu beachten, daß die im Anhang vorliegende Ergebnisliste stark gekürzt wurde.

- ### ⇒ Platz innerhalb der Rangliste
- System ⇒ verwendetes System
- OS/Compiler ⇒ verwendetes Betriebssystem
- CPU ⇒ verwendete CPU
- CPU(Mhz) ⇒ Taktfrequenz der CPU
- MIPS V1.1(V2.1) ⇒ MIPS–Rate der Dhrystoneversion 1.1 bzw. 2.1
- REF ⇒ Referenz : Mit Hilfe dieser Referenznummern werden am Ende der Liste(im Anhang nicht dargestellt) Compiler und Compileroptionen angegeben.

2.2 Dhrystone

Der Dhrystone–Benchmark wurde 1984 von Reinhold P. Weicker in ADA entwickelt (Version 1) und später in C übertragen . Es handelt sich hierbei um einen kleinen Integer CPU–Benchmark,der keine Aussage über das Cacheverhalten ermöglicht. Der Dhrystone gehört zu den synthetischen Benchmarks, d.h. er versucht die mittlere Häufigkeit von Operationen und Operanden eines typischen Programmes zu simulieren. 1988 entwickelte Weicker die Version 2.0, auf dei sich die folgende Analyse bezieht.

Unterschiede zur Version 1:

[1] Performance Database Server'
WWW-Adresse:http://performance.netlib.org/performance/PDS.op.html

- Ausgabe aller benutzen Variablen am Ende
  ⇒ keine toten Variablen
  ⇒ keine Codeunterdrückung

- seperates Compilieren meherer C–Files

Daraus folgt, daß die Version 2 des Dryhstone–Benchmarks nicht mehr so sehr compilerabhängig ist und somit einen genaueren Schluß auf die eigentliche Rechenleistung der CPU (allerdings nur bei Integeroperationen) ermöglicht. Der Benchmark arbeitet mit Hilfe von Zählschleifen und enthält 8 Prozedur- und 3 Funktionsaufrufe.

Befehlsverteilung:

Zuweisungen	52	51.0%
Kontrollflußanwisungen	33	32.4%
Funktions- und Prozeduraufrufe	17	16.7%

Operatorenverteilung:

Arithmetische Op.	32	50.7%
Vergleichsop.	27	42.8%
Logische Op.	4	6.3%

Operandentypen:

Integer	175	72.3%
Character	45	18.6%
Pointer	12	5.0%
String	6	2.5%
Array	2	0.8%
Record	2	0.8%

Einige typische Resultate(Version 2):

Rechner	Betriebssystem	MIPS
VAX 11/780 5 Mhz	UNIX 5.0.1	0.93
VAX 8650 18 Mhz	4.3 BSD	6.20
80486DX2/66 Mhz	MSDOS 3.0	30.20
Pentium 60 Mhz	LINUX 1.2.3	43.58

2.3 Fhourstone

Der Fhourstone-Benchmark ist die Implementierung des Spiels Vier-Gewinnt.

Auf einem Spielfeld(6 Reihen X 7 Spalten) lassen 2 Spieler abwechselnd jeweils einen Spielstein auf eine Spalte fallen. Sieger des Spiels ist der Spieler, welcher als erster 4 Spielsteine in einer horizontalen, vertikalen oder diagonalen Linie erhält. Gibt es nach 42 Zügen noch keinen Sieger, so endet das Spiel unentschieden.

Es wurde bei diesem Benchmark eine Alpha–Beta–Suche unter Nutzung von History-Heuristik und platzoptimierten Hashtabellen verwendet. Der Fhourstone nutzt über 5 MB Hauptspeicher, so daß Rechner mit großem Cache nicht allzu stark bevorzugt werden. Es ist zu beachten, daß es sich bei Fhourstone um einen reinen Integer-Benchmark handelt. Als Ergebnis erhält man die Anzahl der besuchten Positionen pro Sekunde[2].

Der Benchmark gibt Aufschluß über:

- recursive Alpha-Beta-Calls
- Hashing
- Historytabellenerneuerung

Einige typische Resultate:

Rechner	Betriebssystem	Kpos/sec
80486DX2/66 Mhz	MSDOS 5.0	56.5
Pentium 60 Mhz	LINUX 1.2.3	67.6

2.4 Flops

Der Benchmark dient, wie schon dem Namen zu entnehmen ist, der MFLOPS–Bestimmung für die Operationen FADD,FSUB,FMUL und FDIV auf der Basis spezieller Befehlsmixe. Das Programm ermöglicht auf Grund der starken Nutzung von Registervariablen eine Schätzung der PEAK–MFLOPS.

Der Benchmark besteht aus folgenden 8 unabhängigen Modulen(Angaben in %):

[2] Angabe in Kpos/sec = Kilopositionen pro Sekunde

Modulnr.	Aufgabe	FADD	FSUB	FMUL	FDIV
1	Berechnung eines Integrals	50.0	0.0	42.9	7.1
2	Berechnung der Zahl π	42.9	28.6	14.3	14.3
3	Berechnung von $\int \sin x\,dx$ in den Grenzen $[0, \pi/3]$	35.3	11.8	52.9	0.0
4	Berechnung von $\int \cos x\,dx$ in den Grenzen $[0, \pi/3]$	50.0	0.0	50.0	0.0
5	Berechnung von $\int \tan x\,dx$ in den Grenzen $[0, \pi/3]$	46.7	0.0	50.0	3.3
6	Berech. von $\int \sin x \cos x\,dx$ in den Grenzen $[0, \pi/4]$	46.7	0.0	53.3	0.0
7	Berech. der best. Integrale $\int \frac{1}{x+1}dx, \int \frac{x}{x^2+1}dx, \int \frac{x^2}{x^3+1}dx$ in den Grenzen $[0, 102.332]$	25.0	25.0	25.0	25.0
8	Berech. von $\int \sin x \cos^2 x\,dx$ in den Grenzen $[0, \pi/3]$	46.7	0.0	53.3	0.0

Aufbauend aus diesen Modulen werden folgende 4 Testergebnisse erstellt:[3]

Testnr.	Aufbau des Tests	FADD	FSUB	FMUL	FDIV
1	$2 * MO2 + MO3$	40.4	23.1	26.9	9.6
2	$MO1 + MO3 + MO4 + MO5 + MO6 + 4 * MO7$	38.2	9.2	43.4	9.2
3	$MO1 + MO3 + MO4 + MO5 + MO6 + MO7 + MO8$	42.9	3.4	50.7	3.4
4	$MO3 + MO4 + MO6 + MO8$	42.9	2.2	54.9	0.0

Zu diesen Tests nun einige Bemerkungen:

- Test2 enthält die schlecht vektorisierbare Berechnung der Zahl π und ist somit ebenfalls schlecht vektorisierbar

- Test4 enthält keinerlei Divisionsoperationen und ermöglicht, so daß bei diesem Test unterschiedliche Implementierungen des FDIV Befehls nicht ins Gewicht fallen

Bei der Berechnung des MFLOP–Wertes mit Hilfe des McMohanschemas wurden folgende Ergebnisse erzielt:

Testart	Test1	Test2	Test3	Test4
ohne McMohan	6.8948	7.8076	10.7670	13.8918
mit McMohan	8.4917	8.8906	11.7123	14.1428

Es ist eine Angleichung der Werte festzustellen.

[3] MO steht im folgenden für Modul

Einige typische Resultate:[4]

Rechner	Betriebssystem	Test1	Test2	Test3	Test4
VAX 8650 18 Mhz	4.3 BSD	1.1380	1.0810	1.3710	1.5994
80486DX2/66 Mhz	MSDOS 3.0	3.4166	3.3628	4.0868	4.6655
Pentium 60 Mhz	LINUX 1.2.3	5.3068	6.7619	9.3710	11.9521

2.5 Hanoi

Bei diesen Benchmark handelt es sich um das bekannte Problem der Türme von Hanoi.

Ziel: bringe alle Scheiben von einem Turm auf einen anderen

Regeln: bewege jeweils nur eine Scheibe,lege stets eine kleinere Scheibe auf eine größere Scheibe

Dieses Programm besteht praktisch nur aus dem rekursiven Aufruf der Funktion mov (Anzahl Scheiben,von,nach). Die Funktion mov besteht aus einer Vergleichsoperation und abhängig davon aus 2 Inkrementierungen und einer Dekrementierung bzw. aus 1 Addition,1 Subtraktion und 3 Aufrufen von mov. Zur Lösung des Problems sind $2^N - 1$ mov–Aufrufe notwendig. N bezeichnet hierbei die Anzahl der Scheiben. Es handelt sich um einen kleinen Integer–Benchmark, der das Verhalten bei stark rekursiven Verschachtelungen testet.

Einige typische Resultate:

Rechner	Betriebssystem	Züge in 25 μs
VAX 8650 18 Mhz	4.3 BSD	3.745
80486DX2/66 Mhz	MSDOS 3.0	16.19
Pentium 60 Mhz	LINUX 1.2.3	33.71

2.6 Linpack

Der Linpack–Benchmark wurde ursprünglich von Jack Dongarra in FORTRAN entwickelt, und löst ein dicht besetztes lineares Gleichungssystem mittels LU–Zerlegung und Gaußscher Elimination. Es handelt sich um einen stark Floating–Point–Operation geprägten Benchmark, der zum Großteil aus der Funktion saxpy($a[i] = a[i] + x * b[i]$) besteht.Er existiert als single– oder doubleprecision bzw. unrolled[5] oder rolled[6] Variante.

Ein Gleichungsystem der Größe n benötigt für seine Lösung $2n^3/3 + 2n^2$ Operationen. Daraus läßt sich der MFLOPS–Wert des Linpack bestimmen. Es ist zu beachten, daß

[4] Testergebnisse in MFLOPS
[5] mit Schleifenauflösung
[6] ohne Schleifenauflösung

dieser Wert nur Schlüsse auf die Leistung beim Lösen von Gleichungssytemen zuläßt. Aufgrund des relativ geringen Codeteils ist bei größeren Matrizen die Performance stark vom Datencache abhängig.

Auf einem P60 unter Linux 1.2.3 wurden für die verschiedenen Varianten folgende Resultate ermittelt:

Variante	Performance in MFLOPS
single precision rolled	7.529
double precision rolled	6.288
single precision unrolled	7.957
double precision unrolled	6.765

2.7 Nsieve

Nsieve berechnet Primzahlen nach dem Prinzip des Siebs des Eratothenes. Die Orginalversion sieve dieses Benchmarks stammt von Gilbreath. Nsieve entscheidet sich von seinem Vorgänger vor allem durch die Verwendung von Registervariablen und größerer Bytefelder.Diese Verbesserungen möge folgender Vergleich verdeutlichen:[7]

Programm	Laufzeit in Sekunden
sieve	0.133
Nsieve	0.022

Es handelt sich bei nsieve um einen reinen Integer–Benchmark. mit Hilfe einer internen Befehlszählung wird der MIPS–Wert für die Bytefeldgröße bestimmt. Diese Bytefeldgrößen beginnen bei 8191[8] Byte und enden bei 2.56 MByte, woraus zu erkennen ist, daß es sich bei Nsieve um einen etwas größeren Benchmark handelt. Für den Performancevergleich relevante Werte sind High MIPS und Low MIPS. Des weiteren wird ein Wert Linear_Time berechnet, der die erwarte Laufzeit bei linearen Aufwandswachstum[9] angibt.

Einige typische Resultate:

Rechner	Betriebssystem	High MIPS	Low MIPS
VAX 8650 18Mhz	4.3 BSD	6.0	3.2
80486DX2/66 Mhz	MSDOS 6.0	34.5	19.9
Pentium 60 Mhz	LINUX 1.2.3	75.0	31.5

[7] Als Grundlage dient die Berechnung von 1899 Primzahlen bei einem Durchlauf auf einem P60
[8] dieser etwas krumme Wert stammt von seinem Vorgänger sieve
[9] relativ zur Bytefeldgröße

2.8 Test FFT Double Precesion

Der TFFTdp[10]–Benchmark nutzt den Duhamel-Hollmann split–radix Fast Fourier Transformation Algorithmus. Er berechnet die Fourier–Transformation in double precision angefangen von 16 Punkten bis hin zu 262144 Punkten. Dazu werden 4MB Hauptspeicher benötigt. Der TFFTdp–Benchmark ist stark floating–point geprägt.

Es ist zu beachten, daß dies nicht der schnellstmögliche Fast Fourier Transformation Algorithmus ist.

Die Leistungsmessung erfolgt in VAX_FFT's. Diese berechnen sich wie folgt:

$$VAX\_FFT = \frac{Laufzeit\_auf\_zu\_messenden\_Rechner}{Laufzeit\_auf\_VAX8650}$$

Auf der VAX 8650 benötigte das Programm 140.658 Sekunden.

Einige typische Resultate:

Rechner	Betriebssystem	VAX_FFT's
VAX 8650 18 Mhz	4.3 BSD	1.000
80486DX2/66 Mhz	MSDOS 5.0	4.065
Pentium 60 Mhz	LINUX 1.2.3	6.319

3 Gaussbench

Dieser vom Autor entwickelte Benchmark löst dicht besetzte Gleichungsysteme nach dem Gaussalgorithmus. Zu Testzwecken werden 100x100, 200x200, 300x300, 400x400 und 500x500 Matrizen bearbeitet. Es handelt sich um ein reinen Floating–Point Benchmark; die Matrizen bestehen aus double–werten. Des weiteren steht ein Programm zur Generierung der Testdaten zur Verfügung.

Dieser Benchmark soll im weiteren Verlauf der Arbeit parallelisiert werden, um mit ihm auch Aufschlüsse auf parallele Performance zu erhalten.Eine aktuelle Version ist beim Autor erhältlich.

Auf einem P60 wurden unter LINUX 1.2.3 folgende Resultate ermittelt:

DIM	RUNTIME
100	0.11
200	1.02
300	3.87
400	9.65
500	20.33

TOTAL RUNTIME: 34.89

[10] Test Fast Fourier Transformation Double Precesion

4 Paralleles Benchmarking

4.1 Beschreibungsmittel

Ein wichtiges Leistungsmerkmal stellt das Speedup S_p eines parallelen Systems dar. Es ist ein Maß für die Leistungsteigerung eines Algorithmus bei der Verwendung von meheren Prozessoren. S_p ist nach Hockney wie folgt definiert:

$$S_p = \frac{t_{sequentiell}}{t_{nprocessor}}$$

$t_{sequientiell}$ = Laufzeit des schnellsten sequentiellen Algorithmus

$t_{nprocessor}$ = Laufzeit des Algorithmus auf n Prozessoren

Es ist zu beachten, daß es sich beim Speedup um eine relative Größe handelt, bei der vor allen Aussagen über die Skalierbarkeit eines Algorihmus gemacht werden, und die weniger dem Vergleich verschiedener Rechner dient.

Die folgenden Größen dienen der weiteren Beschreibung von Parallelrechnern bzw. darauf aufbauenden Applikationen(Hockney):

- r^S_∞(in MFLOPS) = max. Rechenleistung eines Knotens
- r^S_∞(in Mword/s) = max. Transferrate zwischen 2 Knoten bei Paketgröße $\rightarrow \infty$
- $n^C_{1/2}$(in word) = Paketgröße bei $r^S_\infty/2$
- S^S(in MFLOPS) = Anzahl der Floating–Point–Operationen
- S^C(in word) = Menge der zu sendenden Daten
- N^C(in word) = durchschnittliche Paketgröße

4.2 Theoretische Betrachtungen

Die Benchmarkperformance $R_B(N;p)$ berechnet sich aus der Anzahl der ausgeführten Arithmetikoperationen $F_B(N)$ und der dafür benötigten Zeit $T(N;p)$ wie folgt:

$$R_B(N;p) = \frac{F_B(N)}{T(N,p)}$$

Sie ist abhängig von der Problemgröße N und der Prozessorzahl p.

Die Zeit $T(N;p)$ setzt sich aus der Zeit für die Berechnung $T_{calculation}$ und der Zeit für den Datenaustausch $T_{communication}$ zusammen.

$$T(N;p) = T_{calculation} + T_{communication}$$

$T_{calculation}$ läßt sich aus S^S und r_∞^S berechnen:

$$T_{calculation} = \frac{S^S}{r_\infty^S}$$

Die Zeit für den Datenaustausch splittet sich widerum in die Startupzeit und in die reine Übertragungszeit.

Sie berechnet sich mittels:

$$T_{communication} = T_{startup} + T_{transfer} = \frac{q^C n_{1/2}^C}{r_\infty^C} + \frac{S^C}{r_\infty^C}$$

q^C = Anzahl der zu startenden Kommunikationen.

4.3 Probleme paralleler Benchmarks

Aufgrund der sehr stark differierenden Architekturen ist es schwierig, wenn nicht gar unmöglich einen allgemeingültigen Satz von Benchmarks zu entwickeln. Die Probleme der Portierung paralleler Benchmarks werden hoffentlich in der Zukunft durch das Durchsetzen einheitlicher Standarts, z.B. MPI oder PARMACS, gelöst.

Literatur

[DON95] Jack J. Dongarra : Performance of Various Computers Using Standart Linear Equations Software

[HOC88] Roger W. Hockney, Cristoffer R. Jesshope: Architecture, Programming and Algorithmns
Adam Hilger/IOP Publishing, Bristol & Philadelphia

[HOC94] Roger W. Hockney : Performance Parameters and Results for the Genesis Parallel Benchmarks
Portability and Performance for Parallel Processing , John Wiley & Sons

[HP90] John L. Hennessy, David L. Patterson : Computer Architecture : A Quantitative Approach
Morgan Kaufmann Publishers, Inc., San Mateo, California

[KLE94] Andreas Kleber: Entwicklung,Implementierung und Evaluierung eines parallelen Benchmarksatzes für DM–MIMD–Rechner

[LAR93] Brian Howard LaRose: The Development and Implementation of a Performance Database Server
Computer Science Department CS-93-195

[WEI84] Reinhold P. Weicker: Dhystone: A Synthetic Systems Programming Benchmark
Comm. ACM 27,10

[WEI88] Reinhold P. Weicker: Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules
SIGPLAN Notices vol. 23 no. 8

Anhang(Ergebnisliste des Dhrystone)

Results as of 15 Nov 1995:

###	System	OS/Compiler	CPU	CPU (MHz)	MIPS V1.1	MIPS V2.1	REF
001	DEC Alpha 600 5/266	OSF/1 V3.2c	21164-EB5	266.0	------	366.8	79
002	DEC Server 2100 5/250	UNIX V3.2b	DEC 21064	250.0	------	360.4	70
003	DEC 3000/900 AXP	OSF/1 V3.0	DEC 21064	275.0	------	291.9	63
004	DEC Alpha 600 5/266	OSF/1 V3.2c	21164-EB5	266.0	------	290.0	79
005	DEC 200 4/233	OSF/1 V3.2	DEC 21064A	233.0	189.3	245.8	73
006	DEC Alpha 250 4/266	OSF/1 V3.2c	-----------	266.0	------	226.2	81
007	DEC 10000/610 AXP	OpenVMS V1.0	DEC 21064	200.0	194.9	214.8	6
008	DEC 7000/600 AXP	OSF/1 V1.3a	DEC 21064	200.0	195.5	203.3	33
009	DEC Alpha 250 4/266	OSF/1 V3.2c	-----------	266.0	------	196.0	81
010	DEC 7000/610 AXP	OpenVMS V1.0	DEC 21064	182.0	177.3	195.6	6
011	DEC 3000/800 AXP	OSF/1 V1.3a	DEC 21064	200.0	192.9	189.7	33
012	DEC 4000/710 AXP	OSF/1 V1.3a	DEC 21064	190.0	188.7	189.7	33
013	HP 9000/735	HP-UX 9.03	PA-RISC7150	125.0	224.5	189.2	64
014	Sun Ultra 1	Solaris 2.5	UltraSPARC	167.0	189.7	179.0	80
015	DEC 4000/610 AXP	OpenVMS V1.0	DEC 21064	160.0	159.0	173.0	6
016	Sun Ultra	Solaris 2.5	UltraSPARC	167.0	------	170.8	82
017	DEC 3000/600 AXP	OSF/1 V1.3a	DEC 21064	175.0	168.5	167.4	33
018	PowerMac 9500/120	System 7.5.2	PowerPC 604	125.0	------	167.4	71
019	DEC 3000/500 AXP	OpenVMS V1.0	DEC 21064	150.0	146.7	160.1	6
020	Sun Ultra 1	Solaris 2.5	UltraSPARC	143.0	162.2	153.0	80
021	Sun Ultra	Solaris 2.5	UltraSPARC	167.0	------	151.8	82
022	HP 9000/735	HP-UX 9.01	PA-RISC7100	99.0	------	146.5	52
023	HP 9000/735	HP-UX 9.01	PA-RISC7100	99.0	------	143.0	46
024	DEC 3000/400 AXP	OpenVMS V1.0	DEC 21064	133.0	129.9	142.1	6
025	DEC 3000/500 AXP	OSF/1 T1.3-3	DEC 21064	150.0	211.0	137.1*	25
026	SPARCstation 20/HS21	Solaris 2.4	HyperSPARC	125.0	150.2	136.8	78
027	Mac PowerPC 604	MacOS 7.5.2	PowerPC 604	120.0	------	136.6	74
028	DEC 3000/500 AXP	OSF/1 T1.3-3	DEC 21064	150.0	157.1	133.7	25
029	DEC 2000/300 AXP	OSF/1 V1.3a	DEC 21064	150.0	140.9	129.4	33
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
062	IBM RS/6000 250	AIX 3.2.5	PowerPC 601	66.0	96.1	83.6	54
063	IBM RS/6000 250	AIX 3.2.5	PowerPC 601	66.0	96.1	82.8	54
064	Sun SPARCserver 20/612	Solaris 2.3	SuperSPARC	60.0	93.6	82.2	58
065	IBM RS/6000 Model 340	AIX 3.2	Power Risc	33.0	------	76.3	60
066	Gateway Pentium P5-90	LINUX 1.1.35	Pentium	90.0	80.6	75.9	59
067	HP 9000/712	HP-UX 9.05	PA-RISC	80.0	------	70.7	77
068	DATEL Pentium P5-90	MS DOS 6.22	Pentium	90.0	73.0	70.0	62
069	ZEOS Pentium P5-90	MS DOS 6.22	Pentium	90.0	72.5	70.0	62
070	HP 9000/750	HP-UX 9.0	PA-RISC	66.0	76.1	69.6	5
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
124	Intel 486DX2/66	OS/2 2.1	80486DX2	66.7	32.9	30.2	35
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
213	VAX 8650	4.3 BSD	-----------	18.0	6.3	6.2	2
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
239	VAX 11/780	UNIX 5.0.1	-----------	5.0	0.93	-----	1
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
282	Commodore 64	C64 ROM	6510	1.0	0.0205	---	1

Architectural Development Tracks in Parallel Computing – A Brief Overview

Wolfgang Rehm

rehm@informatik.tu-chemnitz.de
http://noah.informatik.tu-chemnitz.de/members/rehm/rehm.html

Abstract

Recent results in parallel computing confirm that highly parallel, general-purpose shared-memory computers can in principle be built. The architectural development tracks follow no simple foreseeable path. This paper gives a brief overview about some architectures have been recently emerged.

1 Introduction

In spite of the tremendous increase in computing power in the last years, nobody has ever felt a glut in computing power. The main technique computer architects are using to achieve speedup is to do parallel processing.

The parallism present in programs can be classified into different types - regular (data parallelism) versus irregular, coarse-grain versus fine-grain (instruction level) parallelism, etc. Coarse grain parallelism refers to the parallelism between large sets of operations such as subprograms, and is best exploited by multiprocessor systems.

The key open architectural question is the nature of a parallel architecture fitting best a wide range of applications.

Everybody should recognize the importance of what is called "mapping of problem architecture (spatio-temporal data access and communication patterns)" for getting an efficient implementation on a certain parallel computer architecture.

Parallel systems encompass a full spectrum of size and prizes, from a collection of workstations that happen to be attached to the same local-area network, to an expensive high-performance machine with hundreds or thousands of CPUs connected by ultra-high-speed switches.

Obviously, the speed and capacity of the CPUs and their communication medium constrain the performance of any application. But from the perspective of the programmer, the way in which the multiple CPUs are controlled and the way they share information may have even more impact, influencing not just the ultimate performance results but also the level of effort needed to parallize an application.

2 SIMD and MIMD

Dated from the early days of parallel computing distinctions of architectures are made in how processors are controlled and how they can access memory. Both characteristics are still in evidence, although these are no longer the only distinguishing features of parallel computers (see Fig.1, which is an extension of a slide of [Ash95].). The *control model* dictates how many different instructions can be executed at the same time. The *memory model* indicates how many CPUs are capable of directly accessing a given memory location.

On *SIMD (Single Instruction Multiple Data)* computer -sometimes called a processor array- all CPUs execute the same instructions in "lockstep" fashion. MasPar's MP-2 and Thinking Machine's Connection Machine CM-2 are examples of this type of architecture.

In contrast with SIMD machines, in a *MIMD (Multiple Instructions Multiple Data)* multicomputer many instruction streams are concurrently applied to multiple data sets; heterogenous processes may execute at different rates.

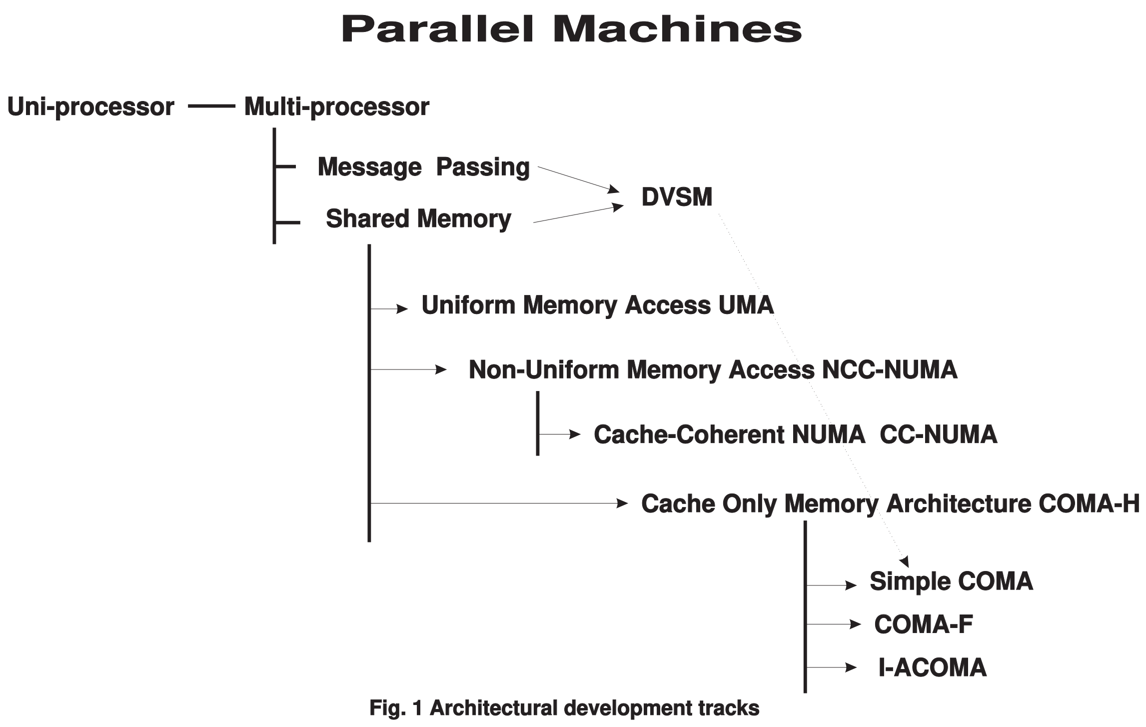


Figure 1: Architectural developments tracks

3 Shared Memory and Distributed Memory

On a *shared-memory multicomputer*, the CPUs interact by accessing memory locations in a single, shared address space. Traditional supercomputers (e.g. Cray Y/MP and C-90, IBM ES/9000, and Fujitsu Vector Processor Series) and the recently emerging so-called *symmetric multiprocessor systems* (*SMP*s at the market place are e.g. 4-CPU SUN-Hypersparc, 4-12-CPU DEC Alpha Server, 16-30-CPU SGI PowerChallenge Line) are examples of this approach.

On *distributed-memory multicomputers*, as the name implies, there is no shared memory. Each CPU has a private memory and executes ist own instruction stream. Most current high-performance paralllel machines -due to their high processor count also refered as *massively parallel processors* (*MPP*s)- are of this type, e.g. Cray T3E, IBM SP-2, and Meiko CS-2. Since they are based on workstation microprocessors technology, these systems are versatile and very cost-effective. From some point of achitectural view they are close related to so-called *network of workstations* (*NOW*s).

4 Message Passing and Shared Memory

Parallel computing on NOWs, also referred as *workstations clusters* (*WSC*s), has been gaining more attention in recent years. Because such workstation clusters use "of-the-shelf" products, they are cheaper than supercomputers. Furthermore, very powerful workstation processors and high-speed general-purpose networks are narrowing the performance gap between workstation clusters and supercomputers. All communication in such a system must be performed by explicitly sending and receiving messages over the network, since no physical memory is shared. This is too one reason why currently the prevailing programming model for parallel computing is message passing and a widely accepted message passing standard (library) *MPI* (*Message Passing Inerface*) [MPI93] has been developed and is still under development.

Message-passing communication as well as *shared-memory communication* style via shared data structures are the major paradigms for parallel programming and, indeed, different schools of thought. Each method has ist own strengths and shortcomings. Message passing seems to be more convenient for distributed-memory-type computers and shared-memory-type communication more convenient for multiprocessor systems with a "naturally" global shared memory.

I believe, that any programming model that relies entirely on message passing or shared memory communication is very likely to fail because of inherent limitations of both. Instead of providing complete solutions for communication, providing simple RISC-like primitives, which expose the full range of hardware communication facilities (and thus the full hardware performance) to higher layers, will be a future solution to this controversy (see also the Active Message approach).

5 Distributed Virtual Shared Memory

Currently there is a growing consensus in the parallel computing community that a shared memory interface is more desirable from the application programmer's viewpoint. This is why it is believed that such an abstraction allowing the programmer to focus on algorithmic development rather than on mapping communication. Consequently recently much effort has been done to provide shared memory (SM) abstraction on top of in fact distributed memory (DM) architectures.

A shared memory abstraction on a distributed memory machine is performed by means of *distributed virtual shared memory* (*DVSM*) techniques. DVSM can be implemented based on (conventional) virtual memory (see IVY [Li88]), with various forms of hardware support (such as the KSR-1 [Ken93] and DDM[Henk93]) and with compiler technology.

Today the challenge in large-scale parallel computing is not to facilitate such an abstraction as such rather than to do so efficiently. On a lower hardware level that means minimize communication overhead (decrease latencies, increase bandwidth), allow communication to overlap computation (use latency hiding or tolerance techniques as prefetching, distributed caching, multithreading, and weaker memory consistency models) and coordinate the two with without sacrificing processor cost/performance.

6 CC-NUMA and COMA

Two interesting variants of large-scale shared-address-space parallel architectures are *cache-coherent non-uniform-memory-access* machines (*CC-NUMA*) and *cache-only memory* architectures (*COMA*s); both have distributed shared (main) memory . Examples of CC-NUMA are Sun's research project S3.mp [Now95] and the Stanford DASH [Len91] while examples of COMAs are te Kedall Square Research KSR-1[Ken92] and the Swedish Institute of Computer Science's *Data Diffusion Machine* (*DDM*) [Hag90].

7 SCI

To provide a hardware-supported shared memory abstraction in a CC-NUMA-style on top of workstation clusters recently was developed the so-called *Scalable Coherent Interface* (*SCI*). By using that an application programmer can write the program as if it is executing in a shared memory multiprocessor and access data by ordinary read and write operations. Nonetheless CSI offers full message passing capabilities.

Clearly, the predominant communication model for programs, that exhibit dynamic communication behaviour or fine-grain sharing, is shared memory. Thus therefore an efficient support of short messages is required; one of the design goals of SCI.

On the other hand the advantages of using message-passing over shared memory for certain types of communication is an undisputed fact. That is why one can notice that message passing machines are moving to efficient support short messages and uniform address space and vice versa DSM computer are starting to provide support for message-like block transfer (SCI too).

There is a convergence of both architectures in hardware and software mechanism to implement the communication abstarctions. A research projact dedicated to efficient integration and support of both cache coherent shared memory and low-overhead user-level message passing is the *FLASH* (*FLexible Architecture for Shared memory*) multiprocessor [Kus94].

8 SMP

So-called *symmetric multiprocessor* (*SMP*) machines are a recent addition to the parallel computing marketplace. They also use workstation microprocessor technology but additionaly join a small number of CPUs (typically 4, 8 or some more up to 30) with certain level of memory hierarchy, e.g. on the main memory level or/and on second-level caches, to achieve more power than high-end uniprocessor workstations offer. Examples include Suns's SPARCServer, HP/Convex's Exemplar , and SGI's PowerChallenge lines.

In a typical simple structure all processors share a global memory via a common bus. If all processors additionaly have the same access capabilities to all I/O (interrupt) resources such a SMP is called *symmetric*. The cache-coherence problem can easily be handled by a *bus-snooping* protocol. Another advantage of a SMP is that all processes have equal access time to shared (main) memory .That is why this architecture is called a *uniform memory-access* (*UMA*) architecture. The main drawback of an SMP is the restricted scalibility due to the limited bandwidth of the common bus, which must be shared by all processors.

9 SMP-Cluster

A simple strategy for implementing a scalable and high-performance computing facility is to cluster SMPs, that means to build *SMP-cluster* (*SMPC*), via very high speed communication systems, e.g. like HIPPI switches or SCI interfaces. The resulting configuration behaves much like a distributed-memory multicomputer, exept that each node actually has multiple CPUs sharing a common memory.

The problem arises with the emergence of SMPCs is how to program them. To date, the major performance success have been scored by programmers who treat SMPCs exactly as what they are: a collection of distinct, smal-scale shared-memeory systems. The communication within a SMP is carried out via shared memory (variables) and between SMPs by message passing.

To exploit parallelism within a SMP an appropriate programming model is that of using the abstraction of threads additionaly to processes (tasks). Whereas the threads (of a process) can only communicate via shared variables processes are able to use other communication primitives, particularly message passing. Remote process communication is restricted to message passing if no technology to provide global memory addressing is available. This is why a programmer must take care how to partitioning and communication is established for a special machine.

Handling different parallelism (processes, threads) and communication styles (shared memory, message passing) depending on the machine configuration is an unacceptable way for a programmers view.

One solution seems to take thoroughly a flat paradigm of concurrent processes each one with his own local memory communicating by messages. A more elaborated version enables a hierachy in the sense that on the upper level appear only processes (e.g. the tasks with ports like in the MACH kernel) and downwards a process can contain several threads of control. The threads of a process communicate via the process-local memory, threads of different processes have to use the communication surface of a process (e.g. ports). Both paradigms are convenient for SMPCs although the latter makes changes in partitioning of parallel units more complicated.

10 UMA, NUMA and COMA

However, probably the primary issue for portable parallel programming (beside uniform access to the secondary storage system) is the to provide uniform access to memory.

One of several software approaches to this problem is the experimental compiler Split-C from the University of California Berkeley; which is a parallel extension of the C laguage that supports access to a global address space on distributed multiprocessors.

From the viepoint of a computer architect the main issue in maintain the abstraction of a global shared memory is the effective design of a DVSM.

SMPCs with a DVSM are a prototype of an architecture where the memory access times varying depending on if a local or a remote access (between SMPs) is performed. Such architectures are called*non uniform memeory access* (*NUMA*) machines.

Typically DSVM systems reqiure additionaly to the maintainance of the consistency of cache-copies in the memory hierarchy of one node to maintain the coherence of multiple cache copies among caches of different nodes (clusters of SMP). A basic concept of cache coherence uses *directory-based* protocols. Machines as described are called *cache-coherent non-uniform-memory-access* (*CC-NUMA*) architectures. In contrast NUMAs with no cache-coherence support are called *NCC-NUMAs*.

Beneath the variants of large-scale shared-address-space architectures are *cache-only memory* architectures (*COMA*s) an interesting solution.

Both the CC-NUMA and the COMA type have distributed main memory and use directory-based cache coherence. Both too migrate and replicate data at the cache level automatically under hardware control, but COMAs do this at the main memory level as well. In a special sense COMA models are NUMAs, in which the distributed main memories are converted to caches. All the caches form the global address space.

The primarey advantage of a conventional COMA architecture is that the memory acts like an "*attraction memory*". That means that the data migrate to the locations where they are needed. No longer a programmer must dealing with data mapping. Again, this advantage is achieved by the penalty of a higher hardware comlexity and an on average longer delay in accessing memory.

11 COMA-F

COMA architectures enable a reduced capacity miss rate due to the large memory caches in each processing node. The primary disadvantage is an increased internode miss penalty due to their hierarchical directory structure which is fundamental to COMAs coherence protocol. The hierarchy efficiently solves the problem of locating copies of memory blocks that may be resident in an arbitrary location in the system.

In contrast CC-NUMA uses a non-hierachical directory structure. There is an explicit home node for each memory block. The directory of this node keeps track of all copies of that block. A copy can be located without traversing a directory hierarchy by searching a single directory memory. The result lies in fewer directory access and a lower internode miss penalty.

A new architecture called *COMA-Flat* (*COMA-F*) is proposed by [Tru95]. It combines the advantages of both CC-NUMA and COMA by retaining the cache-only organization found in COMA but utilizes a non-hierarchical directory structure. Applications where data access by different processes are finely interleaved in memory space and where capacity misses dominate over coherence misses are more convenient for a COMA whereas applications where coherence misses dominate will have better performance on CC-NUMA. Further research will show how powerful COMA-Fs will do the work for various applications.

In contrast to a COMA-F the conventional COMA is called *COMA-H* (COMA-Hierarchical).

12 Simple COMA

Another ongoing research is the so-called *Simple-COMA* architecture [Ash95]. Simple COMA exhibits the conventional COMA properties, but with a reduced hardware and protocol complexity. Therefore the key is using the Memory Menagment Unit on a commodity processor to build the *"attraction memory"at a page granularity.* The operating system becomes responsible for managing the allocation and replacement of the data space in the attraction memory. Complex hardware support is not needed. Here should be noticed that *Suns's Scalable Shared memory MultiProcessor (S3.mp)* research projekt [Now95] now supports both a Simple-COMA implementation as well as its originally intended CC-NUMA.

13 Other models

There are several other research projekts to provide efficient, scalable, shared memory with minimal hardware support, e.g. the *CASHMERe (Coherence Algorithms for Shared Memory aRchitectures)* at the University of Rochester, the *Spark*-project at USC, *SHRIMP* at Princton University, *I-ACOMA* at University of Illinois, etc.

14 Summary

I have tried to give a (much to) brief introduction into the state-of-the-art research and some trends in the field of large-scale parallel computing. Towards the development of truly scalable computers, much research needs to be done.

Finally I like to state some problems the researchers must dealing with in the future:

- Memory-access-latency reduction
- Scalable adaptive cache coherence protocols
- Adaptive granularity of coherence-unit sizes
- More support for weaker memory concistency models
- Realizing distributed shared virtual memory in combination with a broad range of communication abstractions
- Integrating multithreaded and multiscalar architectures for improved processor utilization
- Integration of hardware support for system monitoring, e.g. informing memory operations or performance prediction support
- Expanding software portability at all levels
- Universal parallel overall architecture supporting a wide range of programming models

References

[1] Lenoski, D. , *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor*, PhD Dissertation, Stanford University, December 1991.

[2] Saulsbury, A. et.al. *An Argument for Simple COMA.* 1st IEEE Symposium on High Performance Computer Architecture. January 22-25th 1995, Rayleigh, North Carolina, USA, pages 276-285.

[3] Message Passing Interface Forum. *Document for a standard message-passing interface.* Technical Report. No-CS-93-214, University of Tennessee, Nov.1993.

[4] A. Nowatzyk. et.al. *The S3.mp Scalable Shared Memory Multiprocessor.* (sorry, incompletely, see WWW).

[5] Erik Hagersten et.al. *The cache-coherence protocol of the data diffusion machine.* In Michel Dubois, editor. Cache and Interconnect Architectures in Multiprocessors. Kluwer Academic Publishers 1990.

[6] Jeffrey Kuskin et.al. *The Stanford FLASH Multiprocessor.* In Proceedings of the 21st International Symposium on Computer Architecture, pages 302-313. April 1994.

[7] L.Henk et.al. *The Data Diffusion Machin with a Scalable Point-to-Point Network.* Technical Report CS TR-93-17, Department of Computer Science, University of Bristol, Oct. 1993.

[8] Kai Li. IVY: *A Shared Virtual Memory System for Parallel Computing.* Proceedings of the 1988 International Confernce on Parallel Processing, 2: 94-101, August 1988.

[9] Kendall Sqare Research. *KSR-1 Technical Summary.* Kendall Sqare Research 1992.

[10] Silicon Graphics. *POWER CHALLENGEarray.* Technical Report, July, 1995.

[11] Joe Truman, COMA-F: *A non-hierarchical cache only memory architecture.* Thes.,Department of Electrical Engineering. Stanford 1995.

[12] Ashley Saulsbury et.al. *COMAs can be easily built.* Proceedings of the 1994 International Symposium on Computer Architecture Shared Memory Workshop. Chicago, Illinois, USA

Parallel FEM Implementations on Shared Memory Systems

Lothar Grabowsky

grabowsk@informatik.tu-chemnitz.de
http://noah.informatik.tu-chemnitz.de/members/grabowsky/grabowsky.html

Wolfgang Rehm

rehm@informatik.tu-chemnitz.de
http://noah.informatik.tu-chemnitz.de/members/rehm/rehm.html

Abstract

In the field of parallel FEM methods a number of highly efficient solutions for distributed memory systems are existing. The passage to the 3D–case, especially for non–trivial domains, enforces the use of adaptive techniques. The efficient realization of those techniques on DM–computers is an essentially unsolved problem today.
On the other hand there is an increasing importance of symmetric multiprocessor systems, and in the future clusters of SMP–systems will be a part of parallel computers, which cannot be neglected and so the examination of the efficient port of parallel FEM systems to SMP–systems or –clusters is an important task. We considered a single SMP–system as a first step. The most simple solution is the formal replacement of message passing routines by using a message passing library. The result is the necessity of transmitting local datafields between the processors, although the hardware properties would allow a direct access to global data. The consequence is an unnecessary data transfer, so that an increase of efficiency can probably be reached by making explict use of shared memory for parallization. That suggestion should be verified by an implementation.

1 FEM substructure technique

In this section we will give a short summary of the needed theoretical results. For more details see [HLM91] and the references within.

We consider a symmetric, uniformly elliptic boundary value problem for a partial differential equation of second degree on a bounded domain $\Omega \subset \mathbb{R}^d$, $(d = 2, 3)$ with a piecewise smooth boundary Γ. The weak formulation of this problem leads to a symmetric, $\mathbb{V}_0$– elliptic and $\mathbb{V}_0$–bounded variation problem of the form:

$$\text{find } u \in \mathbb{V}_0 : \qquad a(u, v) = < F, v > \qquad \forall v \in \mathbb{V}_0 \tag{1}$$

We divide Ω into p non–overlapping subdomains, called also substructures or superelements, such that

$$\overline{\Omega} = \bigcup_{i=1}^{p} \overline{\Omega}_i \qquad \text{and} \qquad \Omega_i \bigcap \Omega_j = \emptyset \quad \text{for} \quad i \neq j$$

Let us divide the subdomains $\overline{\Omega}_i$ into finite elements $\overline{\delta}_r$, such that the discretisation process results in a conform triangulation of $\overline{\Omega}_i$. In the following the indices "C" and "I" denote quantities corresponding to the coupling boundary $\Gamma_C = \bigcup_{i=1}^{p} \partial\Omega_i \backslash \Gamma_D$ and to the interior of the subdomains $\Omega_1, \ldots, \Omega_p$, respectively, where Γ_D denotes the Dirichlet–boundary.

Let

$$\Phi = \left\{\varphi_1, \ldots, \varphi_{N_C}, \varphi_{N_C+1}, \ldots, \varphi_{N_C+N_{I,1}}, \ldots, \varphi_{N=N_C+N_I}\right\} \tag{2}$$

the usual nodal basis, where the first N_C functions belong to nodes from coupling boundary Γ_C, the next $N_{I,1}$ functions to inner nodes from Ω_1, and so on. The FE–subspace

$$\mathbb{V} = \mathbb{V}_h \subset \mathbb{V}_0 \tag{3}$$

is now defined by the finite dimensional space:

$$\mathbb{V} = span(\Phi V) \tag{4}$$

with

$$V = (V_C \, V_I) = I = \begin{pmatrix} I_C & 0 \\ 0 & I_I \end{pmatrix}_{N \times N}$$

Once the basis Φ for $\mathbb{V}$ is chosen, the FE–approximation

$$\begin{aligned} &\text{find } u = \Phi V \underline{u} \in \mathbb{V} : \\ &a(\Phi V \underline{u}, \Phi V \underline{v}) = < F, \Phi V \underline{v} > \qquad \forall v = \Phi V \underline{v} \in \mathbb{V} \end{aligned} \tag{5}$$

to (1) results in the system

$$K \underline{u} = \underline{f} \tag{6}$$

where K and f are defined by

$$(K \underline{u}, \underline{v}) = (V^T K V \underline{u}, \underline{v}) = a(\Phi V \underline{u}, \Phi V \underline{v}) \quad \forall \underline{u}, \underline{v} \in \mathbb{R}^N \tag{7}$$

$$(\underline{f}, \underline{v}) = (\underline{f}, V \underline{v}) =< F, \Phi V \underline{v} > \quad \forall \underline{v} \in \mathbb{R}^N \tag{8}$$

The f.e. isomorphism between $u \in \mathbb{V}$ and $\underline{u} = \left(\underline{u}_C^T \, \underline{u}_I^T\right) \in \mathbb{R}^N$ is given by

$$\mathbb{V} \ni u = \Phi V \underline{u} = \Phi \underline{u} \quad \xleftrightarrow{\Phi} \quad \underline{u} \in \mathbb{R}^N \tag{9}$$

Taking into account the arrangement of the basis function given in (2) we can rewrite the system (6) in the block form

$$\begin{pmatrix} K_C & K_{CI} \\ K_{IC} & K_I \end{pmatrix} \begin{pmatrix} \underline{u}_C \\ \underline{u}_I \end{pmatrix} = \begin{pmatrix} \underline{f}_C \\ \underline{f}_I \end{pmatrix} \tag{10}$$

with

$$K_I = diag(K_{I,i})_{i=1,2,\ldots,p} \tag{11}$$

The well known FE substructure technique gives a factorization of K

$$K = \begin{pmatrix} I_C & K_{CI}K_I^{-1} \\ O & I_I \end{pmatrix} \begin{pmatrix} S_C & O \\ O & K_I \end{pmatrix} \begin{pmatrix} I_C & O \\ K_I^{-1}K_{IC} & I_I \end{pmatrix}$$

containing the Schur complement $S_C = K_C - K_{CI}K_I^{-1}K_{IC}$.

In the next section we will apply a via Domain Decomposition parallized and preconditioned CG–method to system (10).

2 A parallized CG–method for shared memory systems

The method described here is derived from a parallel algorithm, developed for distributed memory computers (see [HLM91]).

The aim of this implementation was to make explicit use of shared memory for parallization. From this it seemed to be naturally to work with global datafields and not to split the matrices and vectors in local parts. The base to realize this was the multithreading programming model.

The domain decomposition underlies the parallization as in the message passing case, but contrary to this the coupling nodes (which are not splitted in local parts) are used by several processors, what causes problems for the parallization.

A consequence is that every thread uses a part of the global matrix K and not the superelementmatrices K_i. For all components assigned to inner nodes this obviously makes no difference. For nodes assigned to the coupling boundary a unique assignment to the threads needs to be found. For example the matrix K can be divided as follows

$$\left(\widetilde{K}_s\right)_{ij} = \begin{cases} k_{ij} & : \quad \omega_i, \omega_j \in \overline{\Omega}_s \\ & \quad\;\; \wedge \left(s = \min\limits_r \left\{\omega_i \in \overline{\Omega}_r\right\} \vee s = \min\limits_r \left\{\omega_j \in \overline{\Omega}_r\right\}\right) \\ 0 & : \quad \text{else} \end{cases} \tag{12}$$

where $\wedge$ and $\vee$ denotes logical "and" and "or", respectively. The minimum condition can be replaced by any condition that guarantees an unique assignment of components to threads.

Then the equation

$$K = \sum_{i=1}^{p} \widetilde{K}_i \tag{13}$$

obviously holds.

The corresponding partial vectors can be defined by

$$(\underline{x}_s)_i = \begin{cases} x_i & : \quad \omega_i \in \overline{\Omega}_s \wedge s = \min\limits_r \left\{ \omega_i \in \overline{\Omega}_r \right\} \\ 0 & : \quad \text{else} \end{cases} \tag{14}$$

Additionally we define the following vectors

$$(\overline{\underline{x}}_s)_i = \begin{cases} x_i & : \quad \omega_i \in \overline{\Omega}_s \\ 0 & : \quad \text{else} \end{cases} \tag{15}$$

Remark

Since all threads have access to the whole matrix K other divisions of K are possible too. So a distribution by rows can be accomplished. This guaranties, that every vector component is changed by only one thread and therefore these operations can be performed unprotected. On the other hand this leads to a loss of locality with respect to the data distribution in the assembly phase. The following considerations on the synchronization expense can be done analogously in this case.

With the notations above we can formulate the parallel algorithm:

for $i = 1, 2, \ldots p$ do in parallel
0. <u>Start Step</u> Choose an initial guess $\underline{u}$, e.g. $\underline{u} = O$ $\underline{r}_i = \underline{f}_i$ $\overline{\underline{r}}_i = \overline{\underline{r}}_i - \widetilde{K}_i \overline{\underline{u}}_i$ $\underline{w} = \sum C^{-1} \underline{r}_i$ $\underline{s}_i = \underline{w}_i$ $\sigma_i = \underline{w}_i^T \underline{r}_i$ $\sigma = \sigma^0 = \sum \sigma_i$
Iteration
1. $\widetilde{\underline{v}}_i = 0$ $\overline{\widetilde{\underline{v}}}_i = \overline{\widetilde{\underline{v}}}_i + \widetilde{K}_i \overline{\widetilde{\underline{s}}}$ $\delta_i = \widetilde{\underline{v}}_i^T \widetilde{\underline{s}}_i$ $\delta = \sum \delta_i$ $\alpha = \widetilde{\sigma} / \delta$ 2. $\underline{u}_i = \widetilde{\underline{u}}_i + \alpha \widetilde{\underline{s}}_i$ $\underline{r}_i = \widetilde{\underline{r}}_i - \alpha \widetilde{\underline{v}}_i$ 3. $\underline{w} = \sum C^{-1} \underline{r}_i$ 4. $\sigma_i = \underline{w}_i^T \underline{r}_i$ $\sigma = \sum \sigma_i$ $\beta = \sigma / \widetilde{\sigma}$ 5. $\underline{s}_i = \underline{w}_i + \beta \widetilde{\underline{s}}_i$ 6. $\sigma \leq \epsilon^2 \cdot \sigma^0$? $\xrightarrow{\text{no}}$ goto 1 $\downarrow$yes stop

In the algorithm above the tilde symbol "~" marks the old iterates.

After steps 1 and 4 a global synchronization is necessary in order to make sure that the scalar products are completely evaluated. Additionally you need a synchronization between step 5 an step 1. Here threads that have common nodes have to be synchronized, and this synchronization is really additional compared to the message passing algorithm described in [HLM91]. Step 3 will be discussed in the next section. We only note here that in the case $C = I$ this step results in the local operation $\underline{w}_i = \underline{r}_i$.

2.1 The hierarchical DD-preconditioning

In the message passing algorithm described in [HLM91] a requirement is, that a preconditioner should not increase the communication amount significantly. Various preconditioners satisfying this condition were presented in [HLM90, HLM91]. One example that was used for the practical experiments is described here. This method is equivalent to the hierarchical preconditioning by H. Yserentant (see [Yse86, Yse90]). For this reason we give only the basic theoretical facts here, a detailed decription can be found in the publications above.

Starting point is a coarse grid (user triangulation). By this triangulation (consisting of as few triangles as possible) the geometry of the domain Ω should be sufficiently described. Additionally we assume that these triangles are used to define the subdomains Ω_i. Hence it is favourable if the number of triangles is a multiple of the number of processors. We call the nodes in this start triangulation nodes in level 0. Each node in level 0 is assigned a usual f.e. basis function $\psi_i = \varphi_{h_0,i}$, i.e. these functions form a nodal basis for the level 0 triangulation. Now the triangulation is refined hierarchically in l steps. Again we assign the usual basis functions $\psi_i = \varphi_{h_j,i}$ to nodes supervened in level j. At the end of this process we have defined N_0 basis functions in level 0, N_1 in level 1 an so on. The totality of all $N = N_0 + N_1 + \ldots + N_l$ basis functions is called hierarchical basis

$$\Psi = \{\psi_1, \ldots, \psi_{N_0}, \psi_{N_0+1}, \ldots, \psi_N\} \tag{16}$$

of the f.e. subspace $\mathbb{V} = span(\Psi) = span(\Phi)$. If we use this basis instead of the usual nodal basis, then the associated matrix $\hat{K} = (a(\Psi_i, \Psi_j))_{i,j=1,\ldots,N}$ would have the following properties: (see [Mey90])

1. $\hat{K}$ is not a sparse matrix
2. The condition number $\kappa(\hat{K}) = O(|\ln h|^2)$

Because of 2. the conjugate gradient method (without preconditioning) would be a fast solver for sytems with the system matrix $\hat{K}$. On the other hand 1. leads to an increased memory expense to store $\hat{K}$ and a higher computational expense.

But if we have a basis transformation

$$\Psi = \Phi\hat{V} \tag{17}$$

with a regular $N \times N$-matrix $\hat{V}$ from (17) follows:

$$\hat{K} = \hat{V}^T K \hat{V} \tag{18}$$

and hence

$$\kappa(\hat{K}) = \kappa(\hat{V}^T K \hat{V}) = \kappa(\hat{V}\hat{V}^T K)$$

Therefore the matrix

$$C^{-1} = \hat{V}\hat{V}^T$$

is a good preconditioner for the System $K\underline{u} = \underline{f}$ in the nodal basis Φ.

The matrix multiplication

$$\underline{w} = \hat{V}\hat{V}^T\underline{r}$$

can be performed very efficiently (only N multiplications and 2N additions are needed). We have to perform the multiplications

$$\underline{y} = \hat{V}^T\underline{r} \qquad \text{and} \qquad \underline{w} = \hat{V}\underline{y}$$

This is done by a factorisation

$$\hat{V} = \hat{V}^{(l)}\hat{V}^{(l-1)}\ldots\hat{V}^{(1)} \tag{19}$$

(see [Yse86]). Now we define matrices analogous to $\widetilde{K}_i$:

$$\left(\hat{V}_s^{(k)}\right)_{ij} = \begin{cases} v_{ij}^{(k)} & : \ \omega_i, \omega_j \in \overline{\Omega}_s \\ & \quad \wedge \left(s = \min\limits_r \left\{\omega_i \in \overline{\Omega}_r\right\} \vee s = \min\limits_r \left\{\omega_j \in \overline{\Omega}_r\right\}\right) \\ 0 & : \ \text{else} \end{cases} \tag{20}$$

$$k = 1, 2, \ldots, l$$

where

$$v_{ij}^{(k)} = \left(\hat{V}^{(k)}\right)_{ij}$$

Then step 3 of the parallel algorithm described in section 2 leads to

$$\underline{y}_i = 0, \quad \overline{\underline{y}}_i = \overline{\underline{y}}_i + \left(\hat{V}_i^{(1)}\right)^T \ldots \left(\hat{V}_i^{(l-1)}\right)^T \left(\hat{V}_i^{(l)}\right)^T \underline{r}_i \tag{21}$$

$$\underline{w}_i = \hat{V}_i^{(l)}\hat{V}_i^{(l-1)}\ldots\hat{V}_i^{(1)}\overline{\underline{y}}_i \tag{22}$$

If we rewrite (21) in the form

$$\begin{aligned} &\underline{y}_i^{(1)} = 0, && \overline{\underline{y}}_i^{(1)} = \overline{\underline{y}}_i^{(1)} + \left(\hat{V}_i^{(l)}\right)^T \underline{r}_i \\ &\underline{y}_i^{(k+1)} = \underline{y}_i^{(k)}, && \overline{\underline{y}}_i^{(k+1)} = \underline{y}_i^{(k+1)} + \left(\hat{V}_i^{(l-k)}\right)^T \underline{y}_i^{(k)} \quad (k = 1, \ldots, l-1) \\ &\underline{y}_i = \underline{y}_i^{(l)} \end{aligned}$$

and (22) as follows

$$\begin{aligned} &\underline{w}_i^{(1)} = \hat{V}_i^{(1)}\overline{\underline{y}}_i \\ &\underline{w}_i^{(k+1)} = \hat{V}_i^{(k+1)}\overline{\underline{w}}_i^{(k)} \quad (k = 1, \ldots, l-1) \\ &\underline{w}_i = \underline{w}_i^{(l)} \end{aligned}$$

it is obvious that a synchronization is needed after each level, i.e. $(2 \cdot l)$ synchronizations are needed to perform the preconditioning.

2.2 Results

In order to verify that you can increase the efficiency by using multithreading on shared memory systems we have implemented two versions of an example program, a message passing (according to the algorithm from [HLM91]) and a multithreading version. These program versions were evaluated on a pentium based system with 4 processors (Compaq Proliant 4000) and on a KSR1 system with 8 processors.

As described in 2.1 the synchronization expense increases with the number of levels, whereas the communication expense in the message passing program is independent from the number of levels. Thus it is to expect, that the advance of efficiency in the multithread program will be decreased if the number of levels is increased.

The results shown in fig. 1 confirm this.

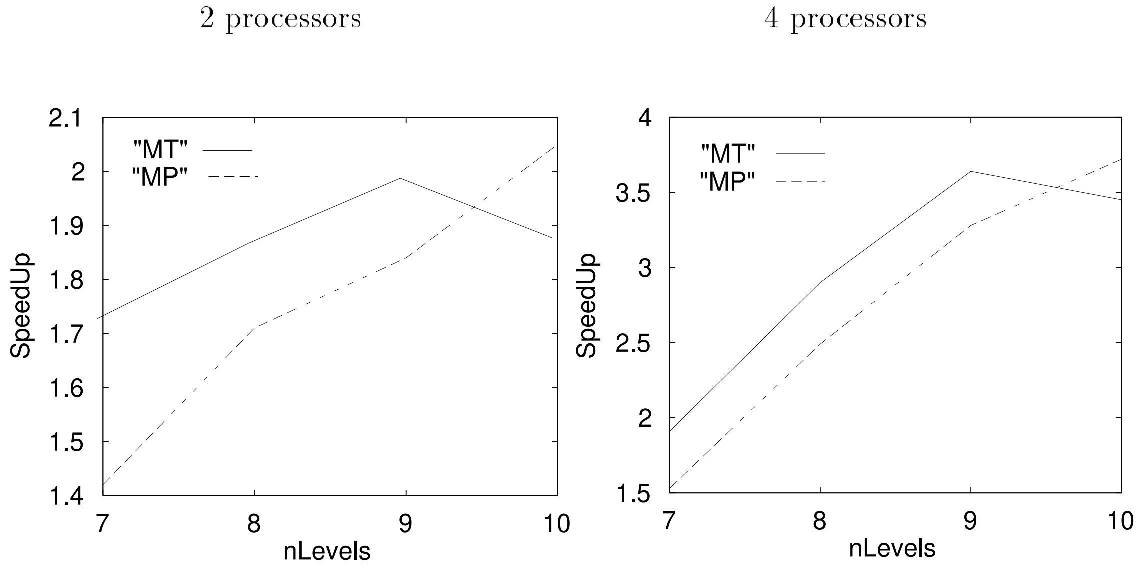


Figure 1: speedup results on the KSR1

3 Conclusions

The results show that the use of multithreading can lead to an increased efficiency on shared memory systems. On the other hand an increased synchronization expense can partial compensate this. Moreover the expense to transpose the algorithm to multithreading is relatively high. With more difficult preconditioners and in the 3d-case this expense will be additionally increased. From this reasons our actual work deals with the use of multithreading inside the message passing library. From this we expect similar increased efficiency but without expensive changes in the application.

References

[HLM90] G. Haase, U. Langer, and A. Meyer, *A new approach to the Dirichlet domain decomposition method*, Fifth Multigrid Seminar, Eberswalde, 1990 (S. Hengst, ed.), Karl–Weierstrass–Institut, 1990, Report R–MATH–09/90, pp. 1–59.

[HLM91] G. Haase, U. Langer, and A. Meyer, *Parallelisierung und Vorkonditionierung des CG-Verfahrens durch Gebietszerlegung*, Proceedings of the GAMM-Seminar "Numerische Algorithmen auf Transputersystemen" held at Heidelberg, 1991, Teubner Verlag, Stuttgart, 1991.

[Mey90] A. Meyer, *A parallel preconditioned conjugate gradient method using domain decomposition and inexat solvers on each subdomain*, Computing **45** (1990), 217–234.

[Yse86] H. Yserentant, *On the multi–level splitting of finite element spaces*, Numer. Math. **49** (1986), no. 4, 379–412.

[Yse90] H. Yserentant, *Two preconditioners based on the multi–level splitting of finite element spaces*, Numer. Math. **58** (1990), 163–184.

Leistungsvergleich ausgewählter Funktionen verschiedener MPI-Implementierungen

Jörg Werner

jwern@informatik.tu-chemnitz.de
http://noah.informatik.tu-chemnitz.de/members/werner/werner.html

1 Leistungsvergleich ausgewählter PARIX- und MPI-Kommunikationsfunktionen

1.1 Aufbau des Testprogramms

Das Testprogramm wurde als MPI- und als PARIX-Version implentiert. Beide Version nutzen das gleiche Rahmen-Programm und ein identisches Testfunktions-Interface. Alle Testroutinen stellen unabhängig von der Implementierung die gleiche Funktionalität zur Verfügung. Das Programm umfaßt Leistungsmessungen für Punkt-zu-Punkt-Kommunikation (blockierend, nichtblockierend), globale bzw. kollektive Kommunikation (Barrier, Broadcast, globale Reduktion) und topologiebezogene Kommunikation.

Da PARIX keine Routinen für globale Kommunikation bereitstellt, wurden diese mit den verfügbaren elementaren Sende- und Empfangsfunktionen nachgebildet.

Art und Parameter des durchzuführenden Test werden dem Hauptprogramm als Parameter übergeben. Je Programmlauf kann für einen spezifischen Einzeltest eine Meßreihe erstellt werden. Eine Meßreihe umfaßt eine vom Nutzer bestimmbare Anzahl von Einzelmessungen mit verschiedenen Datenpaketgrößen. Pro Einzelmessung wird die für den Test benötigte Zeit bei einer festen Datenpaketgröße mit einer bestimmten Anzahl Wiederholungen protokolliert.

1.1.1 Zeitmessung

In der MPI-Version basiert die Zeitmessung auf `MPI_Wtime()`. Diese Funktion liefert die aktuelle Systemzeit in Form von Sekunden. In der PARIX-Version wurde die Funktion `TimeNowLow()` benutzt, welche die Systemzeit knotenlokal mit einer Auflösung von 64 μs zurückgibt.

Die Anzahl der Wiederholungen pro Einzeltest wurde im Allgemeinen so gewählt, daß die akkumulierte Gesamtzeit im Sekundenbereich lag.

1.1.2 Knotenidentfikation

Sowohl PARIX als auch MPI bieten Routinen an, um die eigene Knotennummer als eindeutigen Identifikator innerhalb der angeforderten Partition abzufragen. Die ermittelten Knotennummern stimmen für PARIX und MPI überein. Diese Prüfung war insofern notwendig, da einige der Tests eindeutig festlegte Kommunikationspaare erfordern.

1.2 Punkt-zu-Punkt-Kommunikation

1.2.1 Einführung

Dieser Teil des Test beschäftigt sich mit der Untersuchung grundlegender Kommunikationsprimitiven. Basis dieses Test ist der Austausch von Datenpaketen verschiedener Größe zwischen zwei ausgewählten Knoten unter Nutzung elementarer Sende- und Empfangsroutinen. Die gewonnenen Meßergebnisse dienen als Grundlage zur Ermittlung der Startup-Zeit und der maximalen Übertragungsrate. Um eventuelle architekturspezifische Abhängigkeiten der letztgenannten Beschreibungsgrößen von der Distanz der Kommunikationsknoten zu berücksichtigen, wurde eine gezielte Auswahl der Kommunikationspartner ermöglicht.

1.2.2 Einfache blockierende Punkt-zu-Punkt-Kommunikation

Der Begriff blockiernde Kommunikation entstammt der MPI-Terminologie und beschreibt ein Transferverhalten, bei dem die Kommunikationspartner so lange blockiert sind, bis die entsprechende Aktion (Senden, Empfangen) lokal abgeschlossen ist. MPI bietet neben einem explizit synchronen Modus noch drei weitere Modi an, die mit der Übertragung der Daten in bestimmte Puffer einen Sendevorgang als beendet werten. Einfache Punkt-zu-Punkt-Kommunikation bedeutet in diesem Zusammenhang, daß nur ein eindeutig festgelegtes Paar von Knoten (Prozessen) in den Datenaustausch einbezogen ist. Unter diesen Randbedingungen wurde folgendes Kommunikationsschema verwendet (Ping-Pong-Test):

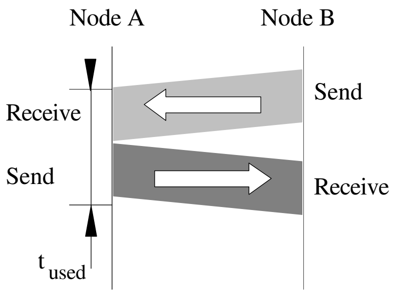


Abbildung 1: Kommunikationsschema - einfache blockierende Punkt-zu-Punkt-Kommunikation

Für eine Paketgröße wird die oben gezeigte Kommunikation n-mal in einer Schleife wiederholt. Die benötigte Zeit zur Übertragung von m Bytes ergibt sich dann wie folgt :

$$t_{mBytes} = t_{used}/(2*n) \tag{23}$$

1.2.3 Blockierende Mehrfach-Punkt-zu-Punkt-Kommunikation

Eine Erweiterung des oben gezeigten Kommunikationsschemas ergibt sich, wenn nacheinander mehrere Knoten in den Datenaustausch einbezogen werden (Stern als Kommunikationsgerüst). Dadurch lassen sich zusätzliche Aussagen ableiten.

Für echte synchrone Kommunikation besitzen die Resulte zur Transferdauer einen allgemeineneren Charakter, da in diesem Fall nicht nur eine (gegebenenfalls minimale) Distanz bewertet wird, sondern mehrere verschiedene. Der so ermittelte Wert könnte als mittlere Kommunikationsdauer für eine bestimme Paketgröße bezeichnet werden. Parallelrechner, deren Startup-Zeit und Kommunikationsbandbreite abhängig sind von der Distanz der einbezogenen Knoten, werden bei dieser Messung Resultate aufweisen, die sich von den Werten einer einzelnen und eindeutigen Punkt-zu-Punkt-Verbindung unterscheiden.

Bei Verwendung puffernder MPI-Modi (Standard, Ready, Buffered) besteht die Möglichkeit der teilweisen Überlagerung des Datenaustausches. Mit der Rückkehr von der Senderoutine kann so bereits vom nächsten Knoten empfangen werden, obwohl der letzte Sendevorgang physisch noch nicht beendet wurde (siehe Abb. 2).

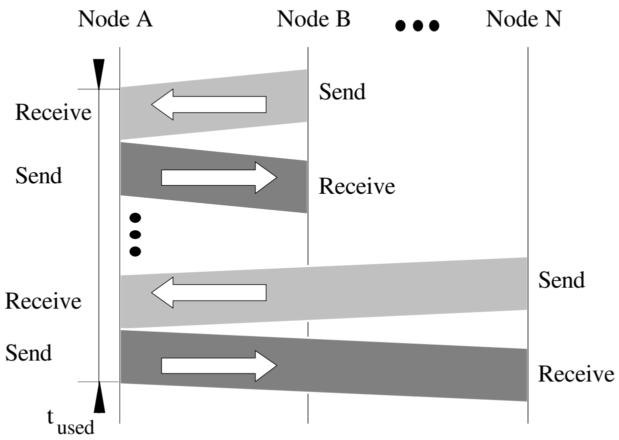


Abbildung 2: Kommunikationsschema - blockierende Mehrfach -Punkt-zu-Punkt-Kommunikation

Für eine Paketgröße wird die oben gezeigte Kommunikation n-mal in einer Schleife wiederholt. Die benötigte Zeit zur Übertragung von m Bytes unter Beteiligung von k Knoten (1 Master(Node A), k-1 Slaves(Node B bis Node N)ergibt sich dann wie folgt :

$$t_{mBytes} = t_{used}/(2 * n * k - 1) \tag{24}$$

1.2.4 Nichtblockierende Punkt-zu-Punkt-Kommunikation

Die Anwendung nichtblockierender Kommunikationsroutinen ermöglicht Aussagen darüber, wie effektiv sich verschiedene Kommunikationsanforderungen überlagern lassen bzw. wie groß ein damit verbundener zusätzlicher Overhead ist. Alle nichtblockierenden Kommunikationsroutinen erfordern den separaten Aufruf von Funktionen, die die Beendigung einer Datenübertragung testen bzw. auf diese warten (sogenannte Complete-Rufe). Analog zum vorangegangenen Abschnitt erstrecken sich die Untersuchungen auf einzelne und mehrere Kommunikationspaare. Bei Einbeziehung mehrerer Knoten ergibt sich unter diesen Voraussetzungen das folgende mögliche Kommunikationsschema (siehe Abb. 3) :

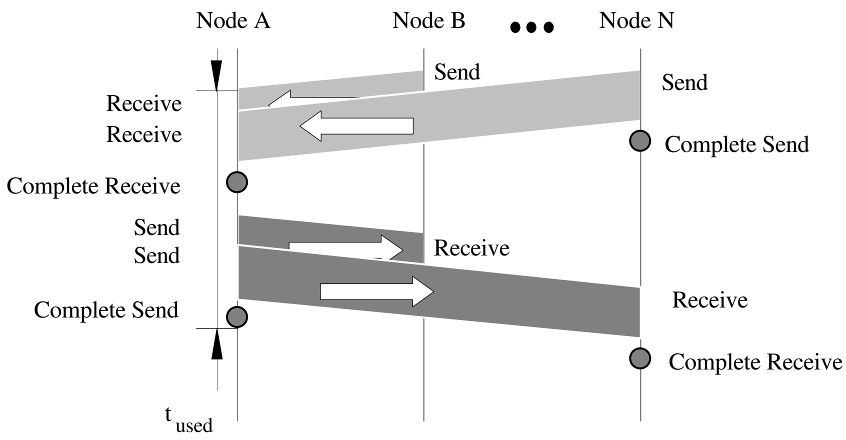


Abbildung 3: Kommunikationsschema - nichtblockierende Mehrfach -Punkt-zu-Punkt-Kommunikation

Für eine Paketgröße wird die oben gezeigte Kommunikation n-mal innerhalb einer Schleife wiederholt. Die benötigte Zeit zur Übertragung von m Bytes unter Beteiligung von k Knoten ergibt sich auch hier wie folgt :

$$t_{mBytes} = t_{used}/(2 * n * k - 1) \tag{25}$$

1.2.5 Spezifika der MPI-Implementierung

MPI bietet für Punkt-zu-Punkt-Kommunikation eine ganze Palette von Routinen an. Senderseitig existieren zu jedem der beiden Übermodi (blocking, nonblocking) vier weitere Untermodi, die nachfolgend kurz beschrieben werden.

Blockierender Modus :

Die Routinen kehren erst vom Aufruf zurück, wenn die zu sendenden Daten in einen Puffer übertragen wurden oder die Daten vollständig vom Empfänger abgenommen sind. Eine Empfangsroutine kehrt zurück, wenn alle angeforderten Daten lokal verfügbar sind (Empfang physisch abgeschlossen).

Standard - Falls empfängerseitig kein korrespondierendes Receive aufgerufen wurde, erfolgt eine Zwischenspeicherung der Daten in vom System bereitgestellte Puffer. Mit der Übertragung in den Puffer ist der Sendevorgang abgeschlossen. Ist die Kapazität dieser Puffer erschöpft, wird auf die Bereitschaft des Empfängers gewartet.

Buffered - Die Daten werden in einen vom Nutzer bereitzustellenden Puffer übertragen. Die Sendeoperation kehrt nach der Zwischenpufferung unverzüglich zurück. Unterdimensionierter Puffer führen zu einem Fehler und nicht zum Warten auf Empfangsbereitschaft.

Synchron - Es wird solange gewartet, bis eine korrespondierende Empfangsroutine gerufen wurde. Die Datenübertragung erfolgt ohne Zwischenspeicherung direkt in den vom Empfänger angegebenen Datenbereich.

Ready - Das MPI-Forum hat diesen Modus so definiert, daß ein Ready-Send unverzüglich zurückkehrt, wenn das passende Receive nicht bereits aktiv ist. Die meisten Implementierungen setzen diesen Modus aufgrund der schlechten Handhabbarkeit mit dem Standard-Mode gleich.

Nichtblockierender Modus :

Mit dem Ruf der entsprechenden Routine werden nur noch Sende- bzw. Empfangsanforderungen (Requests) gestellt. Nach Abgabe des Requests kehren die Routinen unverzüglich zurück. Mit Hilfe von Statusabfragen läßt sich der Zustand einer laufenden Kommunikation ermitteln oder auf dessen Ende warten. Auch hier stehen die vier oben genannten Modi mit den dort erläuterten Spezifika zur Verfügung.

Empfangsseitig beschränkt sich die Anzahl der zur Verfügung stehenden Routinen auf eine blockierende und eine nichtblockierende Funktion. Es sind beliebige Kombinationen zwischen den 8 Sende- und 2 Empfangsroutinen zulässig. Ein blockierendes synchrones Send darf demnach von einem nichtblockierenden Receive bedient werden. Auf diese Möglichkeit wurde innerhalb des Testprogrammes verzichtet. Somit stehen für die Punkt-zu-Punkt-Kommunikation acht Modi zur Verfügung (4 blockierende, 4 nichtblockierende).

1.2.6 Spezifika der Parix-Implementierung

PARIX bietet für synchrone linkgebundene Punkt-zu-Punkt-Kommunikation lediglich eine Empfangs- und eine Senderoutine an. In PARIX-Terminologie bedeutet synchron, daß eine erfolgreiche Kommunikation unbedingt die Bereitschaft beider Partner erfordert. Eine Verletzung dieser Bedingung führt zu einem Deadlock.

Im Unterschied zu MPI sind in PARIX die bidirektionalen Kommunikationskanäle (Links) vor der ersten Benutzung explizit zu eröffnen. Nach der Eröffnung einer Link ist die weitere Handhabung während des Datenaustausches weitgehend identisch zu MPI. Unterschiede ergeben sich nur im Adressierungsschema. PARIX verwendet zur Bestimmung des Kommunikationspartners den Namen der dorthin errichteten Link, MPI benutzt zur Adressierung ein Knotennummer-Kommunikationskontext-Paar (Rank-Communicator).

Asynchrone Kommunikation (gleichbedeutend mit nichtblockierend in MPI) läßt sich in PARIX nur in Zusammenhang mit Topologien verwirklichen. Auch hier werden nur noch Anforderungen gestellt, die mit Hilfe von Threads unabhängig bearbeitet werden. Informationen zum Kommunikationsforschritt sind mit Hilfe von Statusabfragen möglich.

Somit bietet PARIX für Punkt-zu-Punkt-Kommunikation nur die beiden Modi blockierend (synchron) und nichtblockierend (asynchon).

1.3 Globale Kommunikation

Bei diesen Tests stehen Kommunikationsroutinen im Vordergrund, die eine ganze Gruppe von Prozessen bzw. Knoten einschließen. Aus der Fülle möglicher Kommunikationsbeziehungen wurde eine Gruppe von vier Routinen ausgewählt, die zum einen besonders häufig in der parallelen Programmierung Anwendung finden, zum anderen sowohl in MPI als auch in PARIX einfach realisierbar sind (siehe Abb. 5).

Barrier - Synchronisation einer Gruppe von Prozessen. Alle einbezogenen Prozesse bleiben nach dem Eintritt in das Barrier solange blockiert, bis alle teilnehmenden Prozesse das Erreichen des Synchronisationspunktes (Eintritt in das Barrier) signalisiert haben.

Broadcast - Die lokalen Daten eines Prozesses (Master) werden an alle anderen Prozesse gesendet.

Reduce - Globale Reduktionsoperation über eine Gruppe von Prozessen (z.B. Summenbildung über mehrere Knoten), deren Endergebnis in nur einem Knoten verfügbar ist.

Allreduce - Globale Reduktionsoperation über eine Gruppe von Prozessen (z.B. Summenbildung über mehrere Knoten), bei der das Endergebnis in allen Knoten verfügbar ist.

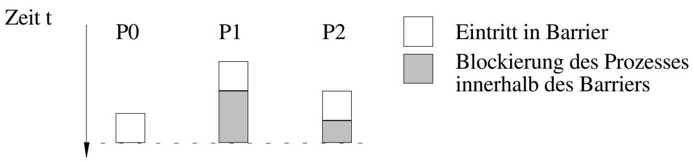


Abbildung 4: Barrieroperation

Innerhalb von Reduce und Allreduce wurde als Reduktionsoperation einheitlich eine Summenbildung auf der Basis von Doubles durchgeführt. Auf die optionale Festlegung eines beliebigen Masters wurde verzichtet, so daß diese Aufgabe immer dem Knoten 0 übertragen wird.

Alle kollektiven Kommunikationsrufe sind blockierend. Für einen einzelnen Knoten gilt die Ausführung einer globalen Operation als beendet, wenn alle Teiloperationen,

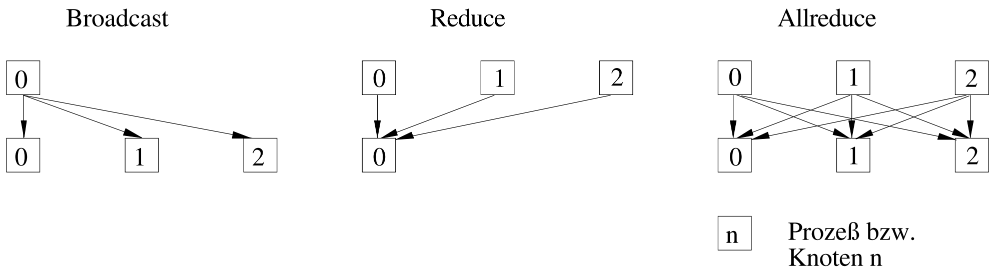


Abbildung 5: logisches Kommunikationsschema globaler Operationen

die der Knoten beizusteuern hat, lokal beendet sind. Diese vom MPI-Forum getroffene Festlegung zur Verweildauer eines Prozesses innerhalb einer globalen Operation erweist sich jedoch bei vergleichenden Leistungsmessungen als nicht geeignet. Die meisten MPI-Implementierungen verwenden zur Propagierung globaler Operationen eine Baum-Topologie. Würde z.B. bei einem Broadcast nur die Zeit erfaßt werden, die der Master zur Bedienung seiner eigenen Kindknoten benötigt, wäre nur der Beitrag des Masters und nicht die Gesamtzeit der globalen Operation erfaßt worden. Da jeder Knoten in der Regel nur wenige Kindknoten (1..4) besitzt, würde sich eine verzerrte Sichtweise ergeben, die einen Vergleich mit anderen Implementierungen kaum noch zuläßt. Dies gilt insbesondere dann, wenn zur Propagierung andere Topologien benutzt werden, bei denen der Master eventuell in mehrere Kommunikationsschritte einbezogen ist (Hypercube).

Die Messung der benötigten Zeit zur Ausführung der globalen Operationen Broadcast, Reduce und Allreduce basiert aus den genannten Gründen auf folgendem Schema.

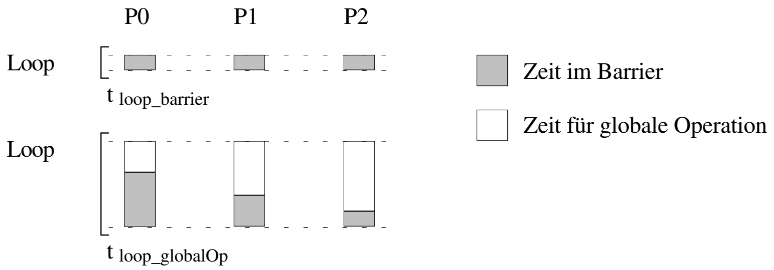


Abbildung 6: Zeitmessung globaler Operationen

Die tatsächlich für eine einzelne globale Operation benötigte Zeit t_{used} bei n Wiederholungen (loop) ergibt sich dann aus :

$$t_{used} = (t_{loop\_globalOp} - t_{loop\_barrier})/n \tag{26}$$

1.3.1 Spezifika der MPI - Implementierung

Für alle Teiltests wurden die von MPI bereitgestellten Routinen benutzt.
(`MPI_Barrier()`, `MPI_Broadcast()`, `MPI_Reduce()`, `MPI_Allreduce()`)

1.3.2 Spezifika der PARIX-Implementierung

PARIX bietet keinerlei vordefinierte Bibliotheksfunktionen für kollektive Kommunikation. Benötigt man als Anwender Operationen auf Gruppenniveau, ist die entsprechende Funktionalität mit Hilfe von Punkt-zu-Punkt-Kommunikation nachzubilden. Die Effizienz bzw. Leistungsfähigkeit solch kollektiver Routinen ist dabei vorrangig von der verwendeten Kommunikationstopologie abhängig. Dabei sollte die benötigte Zeit zum Verteilen bzw. Sammeln von Daten nicht (idealerweise) oder nur in möglichst geringen Maße von der Anzahl der beteiligten Knoten abhängen. Das Idealziel der Entkopplung von Leistungsfähigkeit und Prozessorzahl läßt sich oft nur über zusätzliche Spezialhardware lösen und wurde nur ein wenigen Architekturen realisiert (Cray T3D).

In der PARIX-Implementierung wurde die Funktionalität von Barrier, Broadcast, Reduce und Allreduce auf der Basis topologie- und linkgebundener Kommunikationsroutinen (`Send()`, `Recv()`) verwirklicht. Für jeden der genannten Teiltests läßt sich die zugrundeliegende Topologie aus einer der folgenden Alternativen auswählen :

Stern-Topologie : Der Master besitzt zu jedem einbezogenen Knoten eine separate Link. Die teilnehmenden Knoten werden nacheinander bedient. Für die Bedienung von n Prozessoren werden somit n Zeitschritte benötigt. Da dieses Vorgehen keinerlei Kommunkationsparallelität ermöglicht, ist diese Topologie die uneffektivste und am schlechtesten sklarierungsfähigste.

Hypercube-Topologie : Diese Topologie läßt sich nur für Prozessorzahlen $n = 2^k$ ($k = 1..m$) errichten. K wird als Dimension des Cubes bezeichnet. Die Anzahl benötigter Kommunikationsschritte ist mit der Dimension identisch. In einem Zeitschritt laufen 2^{k-1} Kommunikationen zueinander parallel ab.

Baum-Topologie : Die Knoten bilden einen unbalancierter Baum. Jeder Knoten kann maximal einen Elternknoten und n Kindknoten besitzen. Zur Durchführung einer Broadcast-Operation sind z.B. folgende Schritte Knoten-lokal auszuführen.

1) Wenn es einen Elternknoten gibt, empfange Daten von diesem Knoten.

2) Leite die Daten an alle eigenen Kindknoten weiter

Abbildung 7 zeigt die Reihenfolge der Kommunikationsschritte in einen unbalancierten Baum mit 8 Knoten.

1.4 Topologien

Im Unterschied zum vorangegangenen Abschnitt dienen Topologien hier nicht als internes Mittel zum Erzielen einer gewünschten Funktionalität. In diesem Abschnitt sind Topologien als eine dem Nutzer offenstehende Methode zu sehen, Kommunikation auf einer systematischeren Ebene durchzuführen. Für einen Ring besteht diese abstraktere Sicht darin, Kommunikationspartner nicht mehr konkret durch ihre Prozessornummer zu adressieren, sondern allgemeiner als linken und rechten Nachbarn anzusprechen. Ein weiterer wichtiger Vorteil bei der Benutzung von Topologien liegt in der systematischen Integrationsfähigkeit von Nutzerkenntnissen zum physisch vorhandenen Kommunikati-

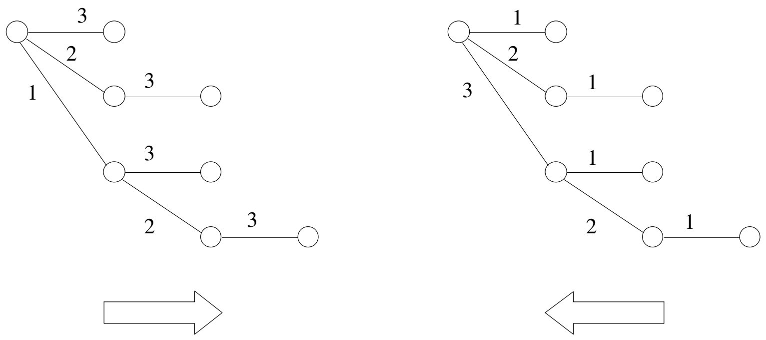


Abbildung 7: Unbalancierter Baum für 8 Knoten

onsnetzwerk. Diese Möglichkeit ist besonders bei Distributed Memory Parallelrechner mit dedizierten Kommunikationsnetzwerk und distanzabhängigen Kommunikationsparametern wertvoll. Ein optimales Mapping der benutzten Softwarekanäle auf die realen Hardwareverbindungen wird bei Maschinen der obengenannten Architektur in der Regel zu einer besseren Performance führen.

Sowohl PARIX als auch MPI bieten diese Möglichkeit, nutzerspezifische Topologien aufzubauen und zu nutzen. Für beide Versionen wurden die zwei folgenden topologiebezogene Tests implementiert:

Optimaler Ring : Über alle verfügbaren Knoten wird eine optimale Ringtopologie aufgebaut. Jeder Knoten kommuniziert dabei nur mit unmittelbar physisch benachbarten Knoten (Distanz = konstant 1 Hops). Die Informationen zur angeforderten Partition werden vom Laufzeitsystem abgefragt (Root-Struktur).

Einfacher Ring : Der Aufbau einer einfachen Ringtopologie erfolgt ohne weitere Berücksichtung von Randinformationen als reine Aneinanderreihung der Prozessornummern (bei n Prozessoren -> 0, 1, 2, ... , n-1, 0).

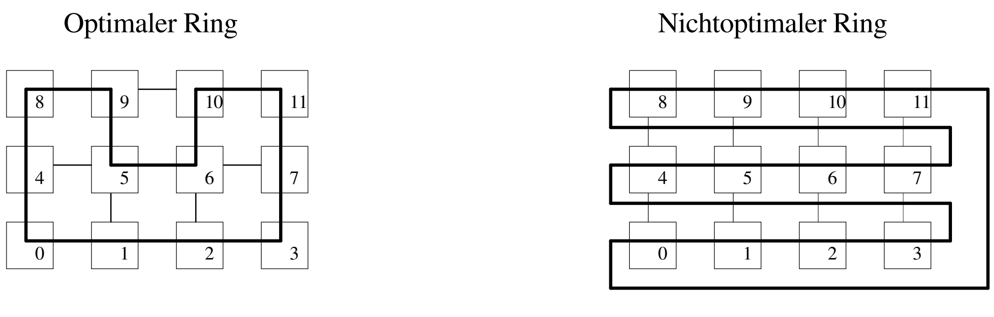


Abbildung 8: Kommunikationsschema - optimaler und einfacher Ring

Die Gegenüberstellung beider Topologogietests erlaubt Rückschlüsse, wie gut sich ein optimales Mapping in der Praxis auswirkt. Gemessen wird jeweils die benötigte

Zeit für einen vollständigen Ringumlauf bei einer bestimmten Datenpaketgröße.

1.4.1 Spezifika der MPI-Implementierung

MPI bietet für den topologiebezogenen Datenaustausch keine speziellen Kommunikationsfunktionen an. Es stehen alle unter Punkt-zu-Punkt-Kommunikation vorgestellten Routinen zur Verfügung. Die topologierelevanten Informationen sind dabei an einen Kommunikationskontext (Communicator) gekoppelt. Alle Topologiedaten werden vom Nutzer durch den Aufbau eines neuen Communicators integriert und können von den Knoten abgefragt werden. Jeder Knoten kann auf diese Weise ermitteln, wieviel Nachbarn er innerhalb der Topologie besitzt und welche Identifikatoren (Ranks) diese Knoten besitzen. Der Partner wird beim späteren Datenaustausch durch Communicator und Rank adressiert. Benutzt man das von MPI ermittelte Array der Nachbarn in Zusammenhang mit einem (Richtungs-) Index, ergibt sich eine PARIX-ähnliche Handhabung.

1.4.2 Spezifika der PARIX-Implementierung

Die Handhabung und Benutzung von Topologien ist in PARIX fest integriert. Für die synchrone Kommunikation auf Topologien existieren separate Befehle (`Send()`, `Recv()`). Während der Errichtung einer Topologie werden die eröffneten Kanäle in einer internen Struktur gespeichert, auf die der Nutzer über einen Kanalindex Zugriff besitzt. Die Adressierung eines Kommunikationspartners erfolgt über die Spezifizierung von Topologie und Kanalindex.

1.5 Resultate

1.5.1 Blockierende einfache Punkt-zu-Punkt-Kommunikation

Die beiden MPI-Modi Standard und Ready sind in ihrem Zeitverhalten identisch. Buffered benötigt im dargestellten Bereich bei allen Paketgrößen etwa 50 μs mehr Zeit. Synchron ist mit fast 500 μs Startup der langsamste Modus. Der erhöhte Zeitbedarf bei Synchron wird durch den höchsten Protokollaufwand verursacht.

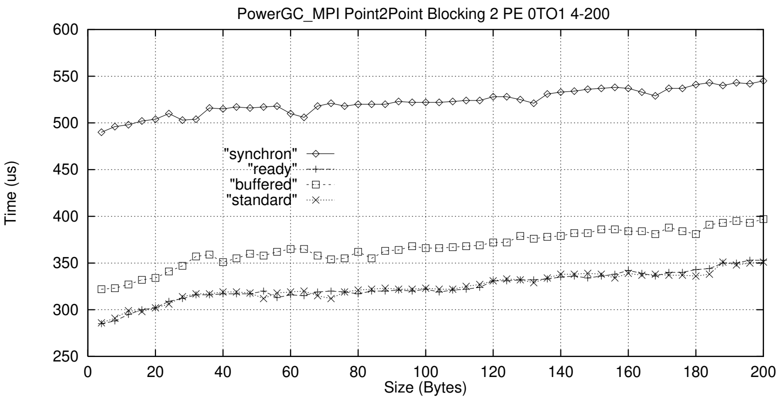


Abbildung 9: einfache blockierende Punkt-zu-Punkt-Kommunikation - MPI-Modi

Die Gegenüberstellung der erreichten MPI-Bandbreiten im Transferbereich von 1KByte bis 50 KByte zeigt, daß Buffered in diesem Bereich deutlich zurückbleibt. Offenbar wird bei Nutzung der Modi Standard und Ready die bei MPI übliche Pufferung der Daten den Kommunikationsprozessoren eines PowerGC-Knotens übertragen, während bei Buffered die Zwischenspeicherung der Transferdaten in den Nutzerpuffer von der CPU bewältigt wird.

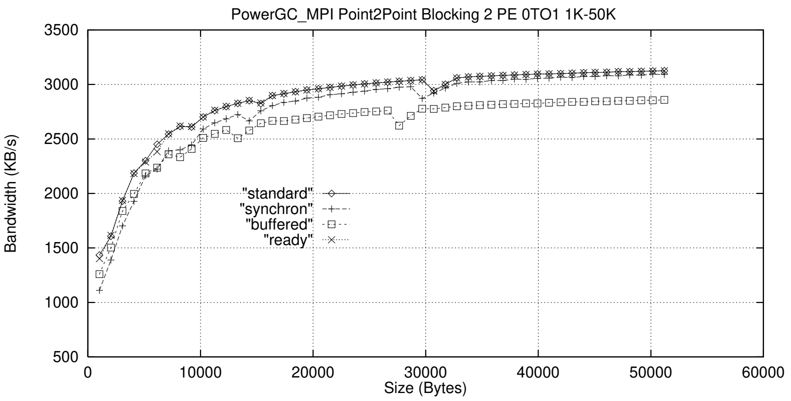


Abbildung 10: Bandbreiten der MPI-Modi - Bereich 1KByte-50KByte

Im Vergleich mit PARIX schneiden die blockierenden MPI-Modi generell schlechter ab. Da die benutzte MPI-Implementierung auf PARIX aufsetzt, liegen die erzielten Resultat im Erwartungsbereich und sind weigehend durch den erhöhten Verwaltungsaufwand in MPI verursacht. Die relativ hohen Startup-Zeiten des Power-GC steigen mit MPI um nahezu 100%.

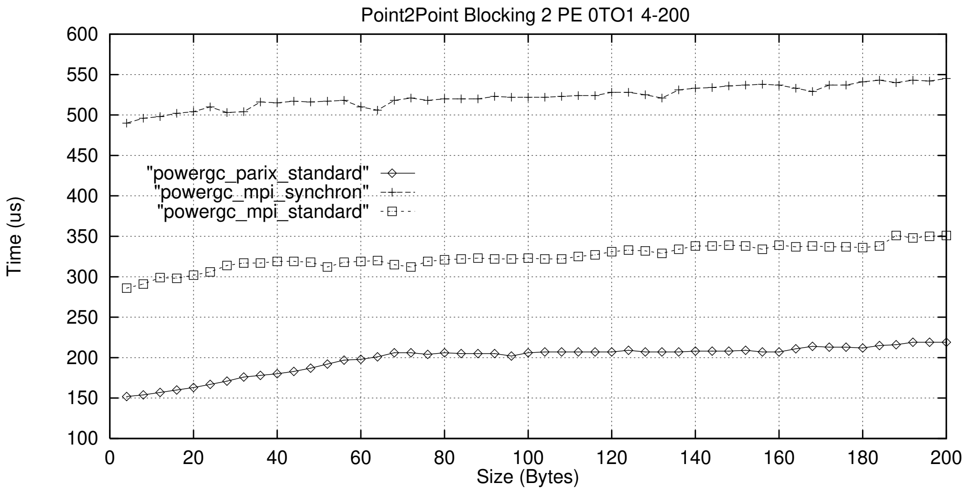


Abbildung 11: Vergleich PARIX - MPI, einfache blockierende Punkt-zu-Punkt-Kommunikation

Die folgende Grafik zeigt die benötigte Zeit zum Transfer von 4 Byte in Abhängigkeit von der physischen Distanz der Knoten im 2-D-Gitter. Das gesamte physische Routing von Nachrichten in PARIX und MPI basiert letztendlich auf diesem Gitter. Die verfügbaren 64 Knoten des an der TU-Chemnitz installierten Power-GC sind in einer Partition konfiguriert, deren X- und Y-Dimension jeweils 8 Knoten beträgt. Um das distanzabhängige Transferverhalten zu untersuchen, werden nur Kommunikationspartner ausgewählt, die auf der Außenkante der Partition liegen.

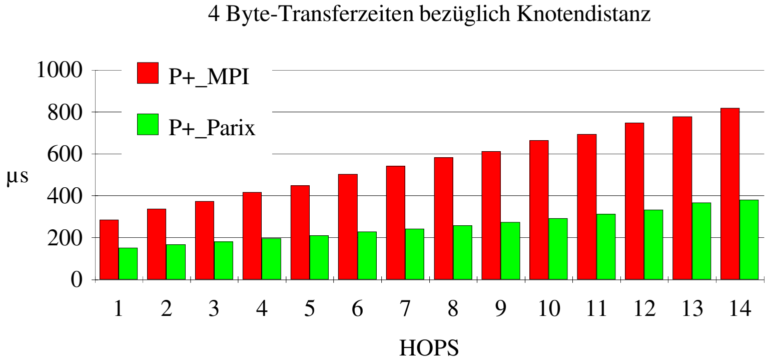


Abbildung 12: 4-Byte-Transferzeit in Abhängigkeit von der physischen Knotendistanz

Tabellarische Gegenüberstellung der Zeiten für einen Transfer von 4 Byte vom Knoten 0 zu verschiedenen Knoten der Partitions-Außenkante.

$$Knotenentfernung[HOPS] = abs(\Delta X) + abs(\Delta Y) \tag{27}$$

HOPS	MPI [μs]	PARIX [μs]	MPI-Bedarf [%]
1	286	152	+88
2	338	166	+104
3	374	180	+108
4	417	196	+113
5	449	211	+113
6	504	229	+120
7	541	243	+123
8	583	258	+126
9	612	273	+124
10	663	292	+127
11	694	312	+122
12	748	333	+125
13	777	366	+112
14	819	380	+116

Bei Darstellung der erzielten Kommunikationsbandbreite über einen Datenpaketbereich von 4 Byte bis 1 MByte bleiben die Vorteile für PARIX bis in den Bereich von 64 KByte deutlich erhalten. Für beide MPI-Modi ist bei 1 KByte ein Performanceknick sichtbar, der eventuell auf einen Wechsel des Übertragungsprotokolls zurückzuführen ist. Im Vergleich der beiden MPI-Modi untereinander bleibt Synchron stets hinter Standard zurück und schließt erst ab 64 KByte zu den anderen Modi auf. Die Messungen wurden zwischen physisch unmittelbar benachbarten Knoten (1HOPS) durchgeführt.

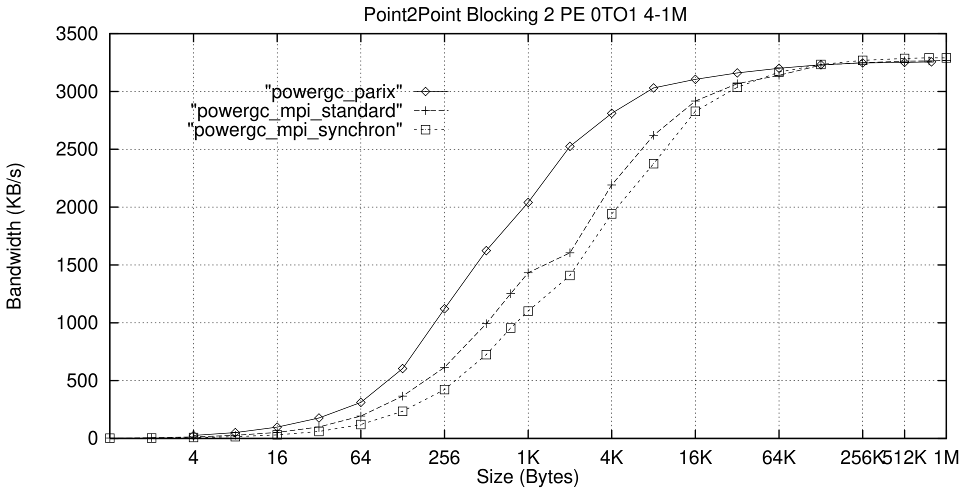


Abbildung 13: Vergleich der Bandbreiten für einfache blockierende Punkt-zu-Punkt-Kommunikation

1.5.2 Blockierende Mehrfach-Punkt-zu-Punkt-Kommunikation

Die im folgenden dargestellen Ergebnisse basieren auf einem Kommunikationsschema, bei dem Knoten 0 nacheinander mit verschiedenen anderen Knoten Daten austauscht. Bei einer Partitionsgröße von 8 Knoten, wird die gemessene Zeit durch die Anzahl der vom Master zu bedienenden Knoten (7) geteilt.

Für PARIX bleiben die Ergebnisse weitgehend gleich zum vorangegangenen Abschnitt, da der Datenaustausch wirklich synchron stattfindet.

Die Ergebnisse zu MPI zeigen den asychronen Charakter der blockierenden MPI-Sendemodi. Selbst der eigentlich synchrone Modus erreicht bei dem genannten Kommunikationsschema eine höhere Bandbreite als PARIX. Die erzielten Geschwindigkeitsvorteile beruhen dabei auf der Überlagerung mehrerer Transfers. Während PARIX auf die physische Beendigung einer Kommunikation tatsächlich wartet, kehren die MPI-Senderoutinen nach der Übertragung der Daten in Puffer unverzüglich zurück und sind bereit, vom nächsten Knoten zu empfangen.

Eine begrenzte Praxisrelevanz dieses synthetischen Kommunikationstests ergibt sich aus der Erfahrung, daß die Datenverteilung bei kleineren Partitionsgrößen häufig auf der Basis einer solchen Stern-Topologie realisiert wird.

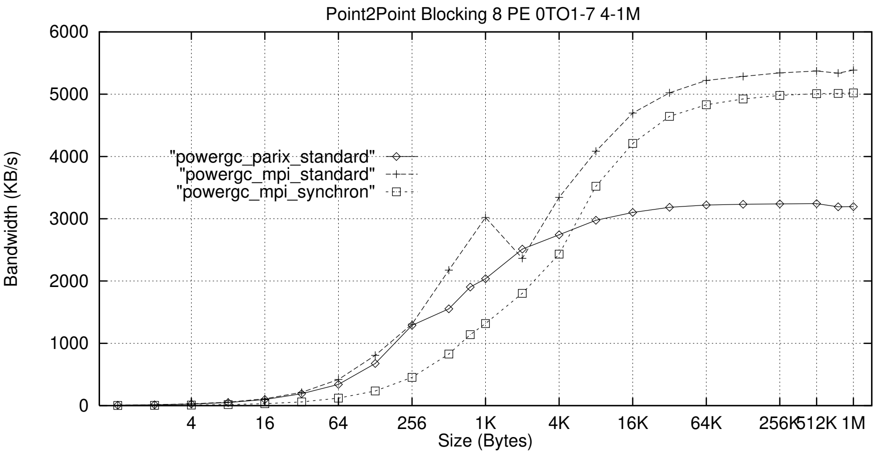


Abbildung 14: Blockierende Mehrfach-Punkt-zu-Punkt-Kommunikation - 8 Knoten

1.5.3 Nichtblockierende einfache Punkt-zu-Punkt-Kommunikation

Die folgenden Ergebnisse beziehen sich auf den Datenaustausch eines unmittelbar benachbarten Knotenpaares mit Hilfe nichtblockierender Kommunikation. Nichtblockierend bedeutet, daß mit dem Ruf der Sende- bzw. Empfangsroutine nur noch Kommunikationsanforderungen gestellt werden und die Routinen danach unverzüglich zurückkehren. Die Beendigung eines angestoßenen Transfers ist explizit zu testen. Analog zum blockierenden Modus bietet MPI die nichtblockierenden Untermodi Standard, Ready, Synchron und Buffered an.

Abbildung 15 zeigt PARIX im Vergleich mit MPI-Standard und MPI-Synchron.

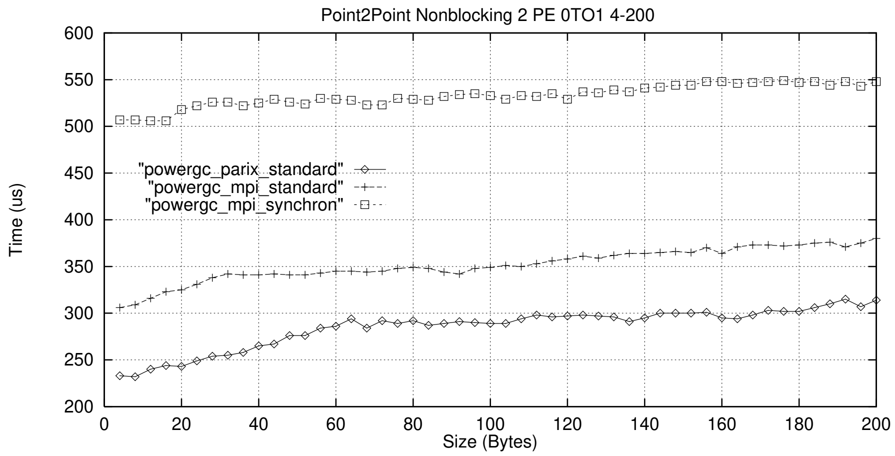


Abbildung 15: Nichtblockierende einfache Punkt-zu-Punkt-Kommunikation

Die Startup-Zeiten für MPI sind gegenüber dem blockiernden Mode verhältnismaßig gering gestiegen (Standard +20 μs, Synchron +10 μs). Für PARIX erhöht sich die Startup-Zeit von 150 μs auf etwa 230 μs. Die geringere Anstieg der MPI-Startup-Zeiten läßt sich damit erklären, daß wesentliche Basismechanismen zur Verwaltung des Message Passing in MPI ebenfalls asynchronen Charakter besitzen.

Aus der Darstellung zur Bandbreite der nichtblockierenden einfachen Punkt-zu-Punkt-Kommunikation (Abb. 16) ist ersichtlich, daß PARIX bis zu einer Paketgröße von etwa 8 KByte eine bessere Bandbreite aufweist. Ab dem genannten Punkt bietet MPI sowohl mit Standard als auch mit Synchron die bessere Performance.

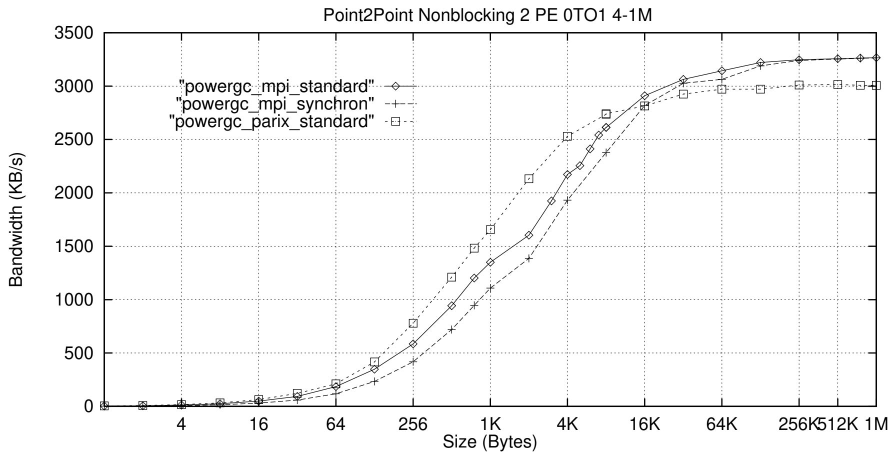


Abbildung 16: Bandbreite nichtblockierende einfache Punkt-zu-Punkt-Kommunikation

1.5.4 Nichtblockierende Mehrfach-Punkt-zu-Punkt-Kommunikation

Unter Verwendung nichtblockierender Punkt-zu-Punkt-Kommunikation tauscht Knoten 0 mit allen anderen Knoten einer Partition Daten aus. Die dafür benötigte Zeit wird durch die Anzahl der zu bedienenden Knoten (für diese Tests 7) geteilt und geht dann als Bewertungszeit ein.

Bei sehr kleinen Nachrichten nähern sich unter diesen Voraussetzungen die Zeiten von PARIX und MPI-Standard sehr stark an (4 Byte Transfer, PARIX 127 μs, MPI-Standard 143 μs).

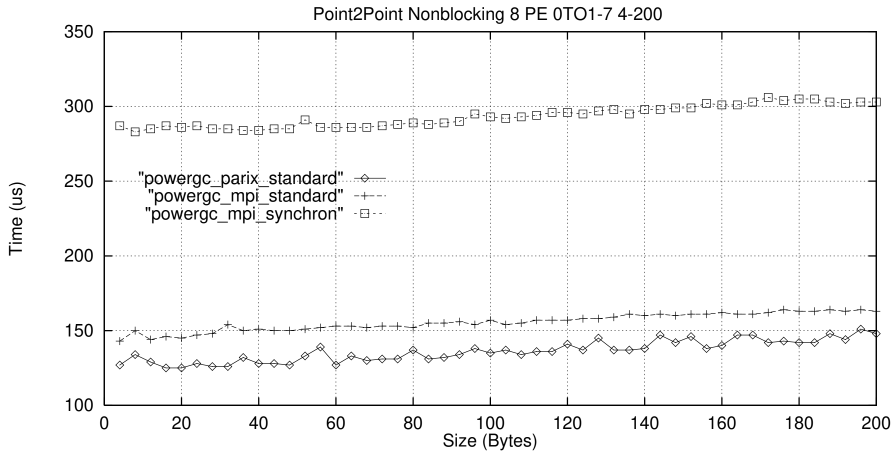


Abbildung 17: Latenzzeit nichtblockierende Mehrfach-Punkt-zu-Punkt-Kommunikation

Die Darstellung der Bandbreite zeigt, daß die oben genannte Annäherung von PARIX und MPI-Standard nur bis zu einer Pakergröße von 1 KByte relevant ist. Dort erfolgt für beide MPI-Modi ein relativ starker Leistungseinbruch, der MPI deutlich gegenüber PARIX abfallen läßt.

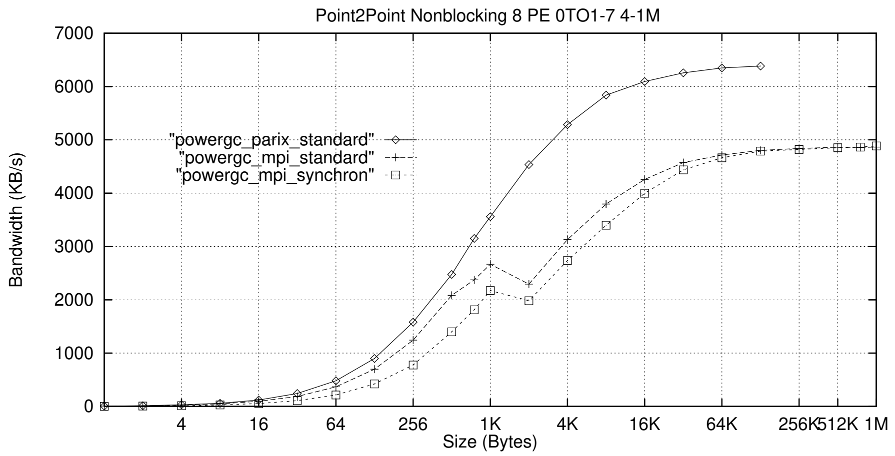


Abbildung 18: Bandbreite nichtblockierende Mehrfach-Punkt-zu-Punkt-Kommunikation

1.6 Resultate kollektive Kommunikation

Wie unter 1.3 beschrieben wurden alle kollektiven Kommunikationsfunktion für PARIX durch Punkt-zu-Punkt-Kommunikation auf Topologien nachgebildet. Als Topologien stehen Stern- und Hypercube-Topologie sowie unbalancierter Baum zur Auswahl. In den nachfolgenden Diagrammen wurde die Ergebnisse für die Stern-Topologie nicht mit aufgenommen. Im Vergleich zu den anderen Topologien sind die erreichten Werte als unakzeptabel zu bewerten.

1.6.1 Barrier

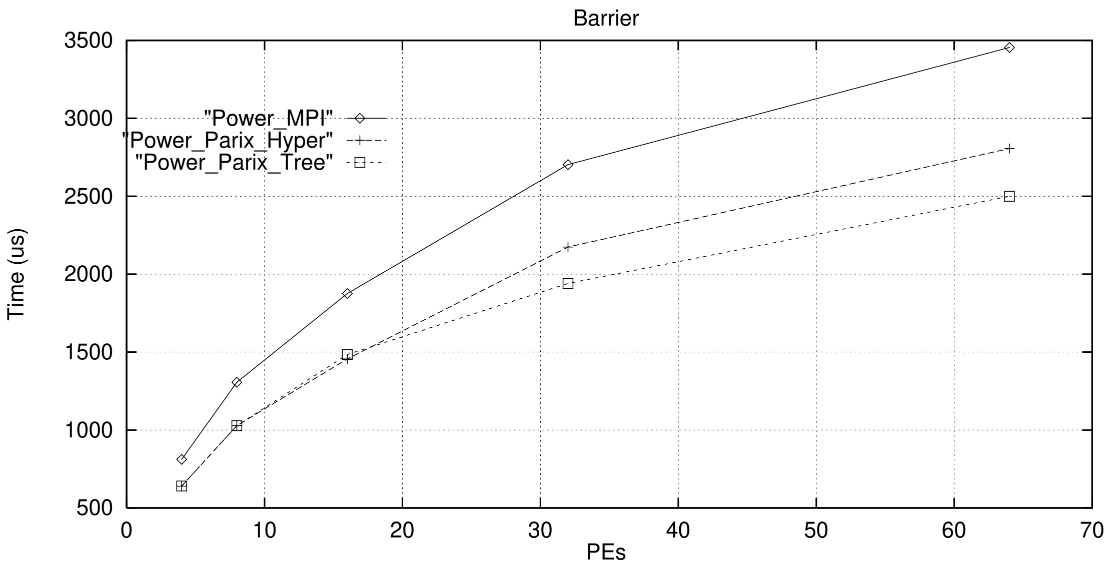


Abbildung 19: Barrier 4-64 PE

1.6.2 Broadcast

Ab mehr als 32 Knoten läßt die Effektivität des Hypercubes nach. Bei 64 Knoten ist MPI-Broadcast bereits schneller. Die beste Performance liefert die baumbasierte PARIX-Version. Abgesehen vom einem erhöhten Zeitbedarf weist MPI annähernd den gleichen Kurvenverlauf auf wie Parix_tree. Dadurch ist anzunehmen, daß MPI die gleiche Topologie benutzt. Der MPI-Mehrzeitbedarf von etwa 800 μs bei 4 Knoten kann unter dieser Annahme auch rechnerisch nachvollzogen werden. Im unbalancierten Baum werden bei 4 Knoten 2 Kommunikationsschritte benötigt. Der Transfer von 2 KByte erfordert bei MPI etwa 400 μs mehr Zeit gegenüber PARIX.

Abbildung 20 zeigt, daß die Leistungsfähigkeit des Hypercubes bei relativ großen Datenmenge (16 KByte) deutlich hinter den anderen Alternativen zurückbleibt. Die baumbasierte PARIX-Version und MPI unterscheiden sich in ihrem Zeitbedarf um etwa 2 ms.

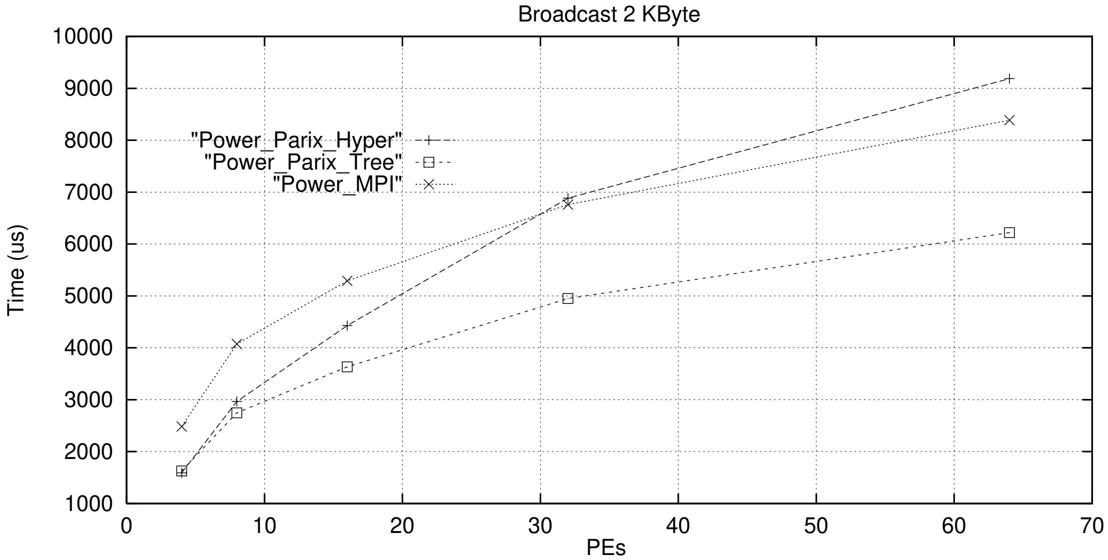


Abbildung 20: Broadcast von 2 KByte für 4-64 PE

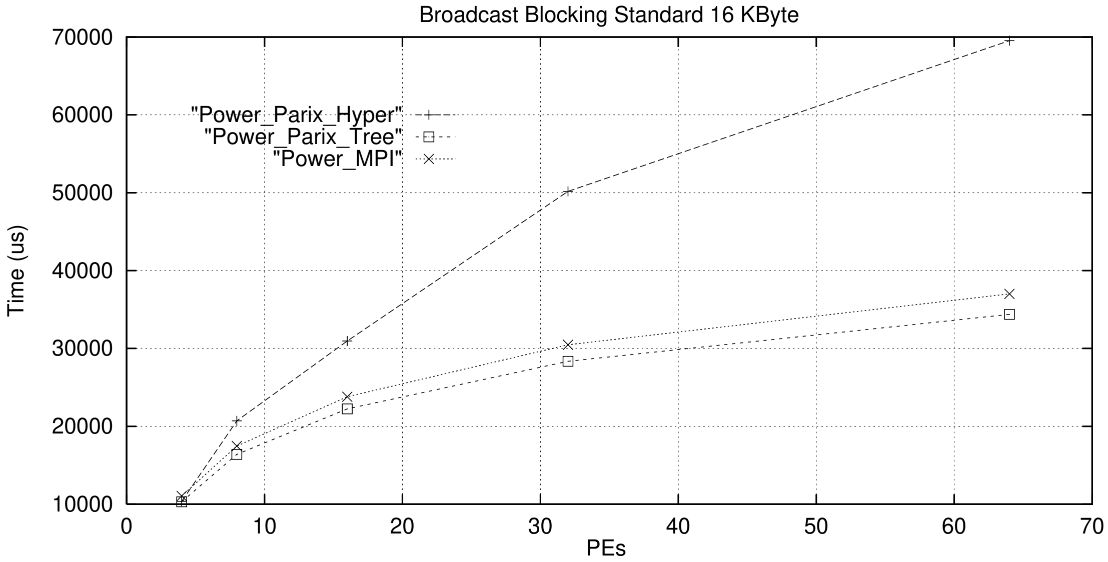


Abbildung 21: Broadcast von 16 KByte für 4-64 PE

1.6.3 Reduce

Abbildung 22 zeigt die Ergebnisse einer globalen Summenbildung über 32 Byte (4 Doubles). Im Bereich dieser Datenmengen sind beide PARIX-Topologien im Vorteil.

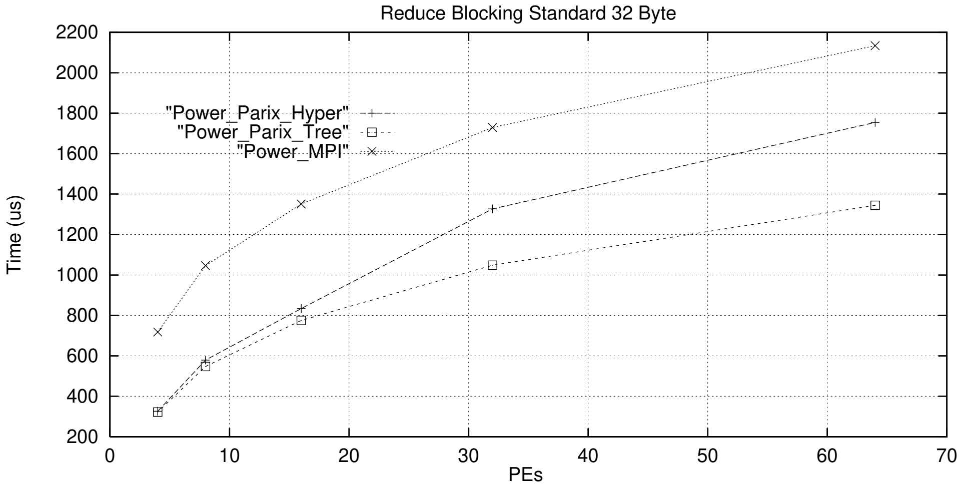


Abbildung 22: Reduce 32 Byte 4-64 PE

Die Ergebnisse der Reduktion über 2 KByte (512 Doubles) zeigen ein Abfallen der Hypercube-Effektivität bei höheren Prozessorzahlen. MPI und die baumbasierte PARIX-Version unterscheiden sich um etwa 1,5ms.

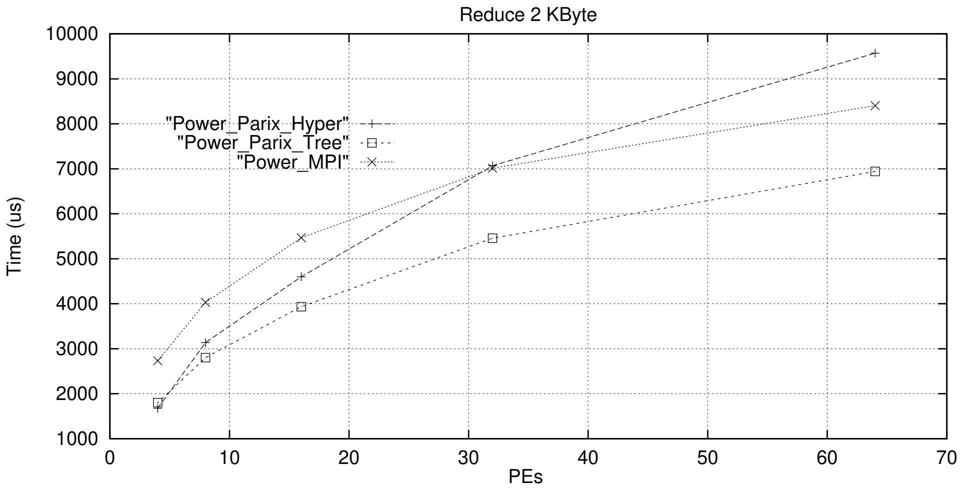


Abbildung 23: Reduce 2 KByte 4-64 PE

1.6.4 Allreduce

Allreduce liefert zu Reduce vergleichbare Ergebnisse. Bei kleinen Datenmengen ist MPI die langsamste Variante. Für größere Vektoren und größere Knotenzahlen wird der Hypercube zunehmend ineffektiver.

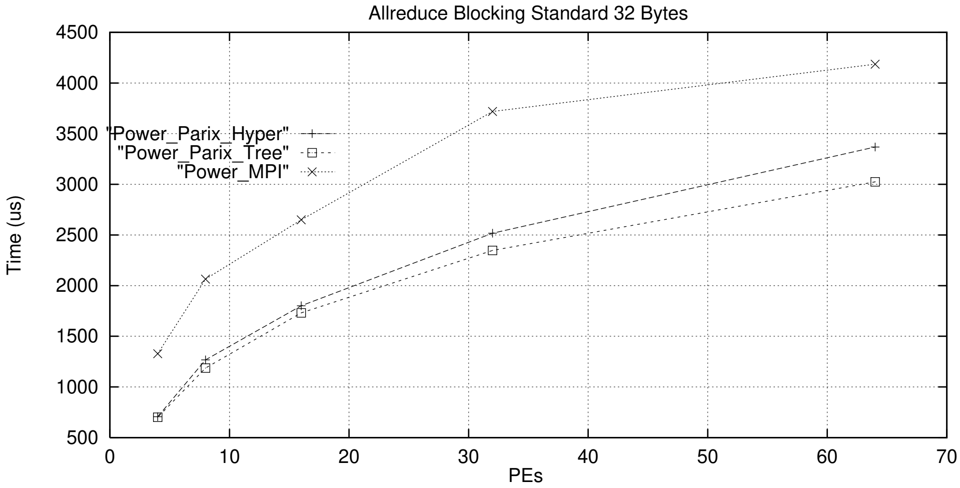


Abbildung 24: Allreduce 32 Byte 4-64 PE

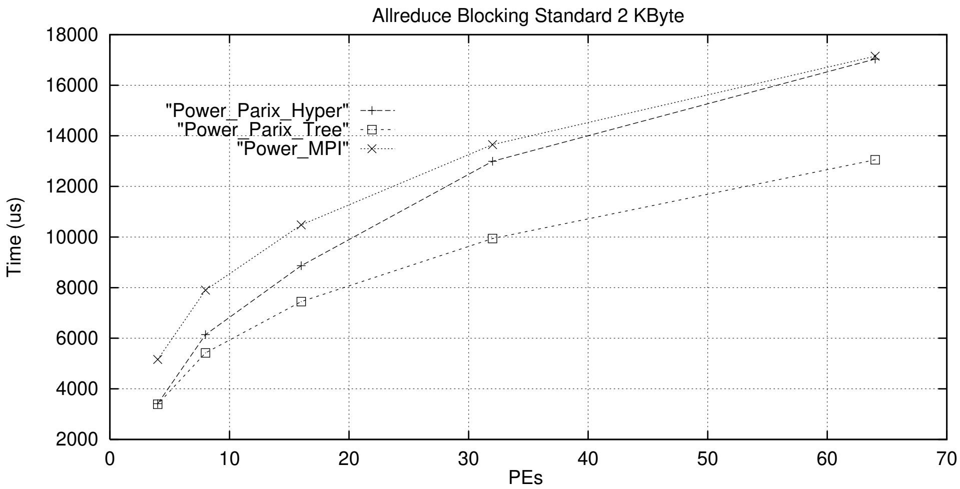


Abbildung 25: Allreduce 2 KByte 4-64 PE

1.6.5 Topologien

Die Gegenüberstellung der Ergebnisse zum Datenaustausch auf einem optimalen und einfachen Ring für 64 Knoten zeigt, daß die Berücksichtigung des physischen Kommunikationsnetzwerkes bei MPI zu einer deutlich stärkeren Verbesserung führt, als dies bei PARIX der Fall ist. Bei MPI ist die bessere Auswirkung eines optimalen Mappings auf die Startup-Zeiten zurückzuführen, die im Vergleich zu PARIX mit größer werdender Knotendistanz stärker anwachsen. Auch hier ist bei MPI oberhalb von 1 KByte ein sprunghaftes Anwachsen der benötigten Zeit zu beobachten.

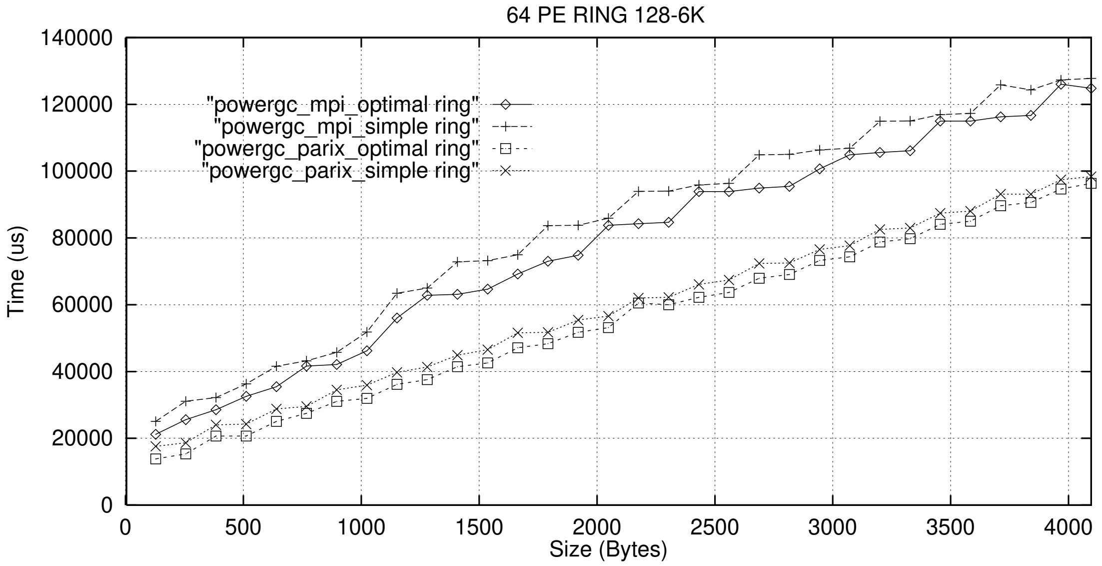


Abbildung 26: Optimaler und einfacher Ring unter PARIX und MPI für 64 PE

2 Architekturspezifische MPI-Implementierung des EPCC für die CRAY T3D

Zwischen 1994 und Mai 1995 entwickelten das Edinburgh Parallel Computing Centre (EPCC) und die Firma Cray Research eine eigene Implementierung des MPI-Standards für die CRAY T3D. Durch die Berücksichtigung spezieller Architektur- und Hardwareeigenschaften sollte dem Nutzer unter MPI ein Höchstmaß an Performance gesichert werden.

Auf der Basis der genannten MPI-Implementierung werden Leistungsmessungen für Punkt-zu-Punkt- und kollektive Kommunikation diskutiert und den theoretisch verfügbaren Leistungsparametern gegenübergestellt.

2.1 Architekturübersicht zur Cray T3D

Die Cray T3D ist eine Distributed Memory Multiprozessormaschine, die als natürliches Speicher- und Programmiermodell jedoch Shared Memory zur Verfügung stellt. Die Struktur des Kommunikationsnetzes entspricht einem 3D-Torus. Alle Knoten sind über 6 unabhängige Kanäle in dieses Netz integriert. Die hardwareseitig verfügbare Kommunikationsbandbreite wird mit 300 MByte/s angegeben. Jeder Knoten enthält zwei DEC Alpha 21064 Prozessoren, 64 MB RAM, eine Interconnect-Unit und eine Block-Transfer-Unit. Die Prozessoren werden mit 150 MHz getaktet, unterstützen 64 Bit Integer und IEEE Floating Point Operationen und liefern eine Peak-Performance von je 150 MFlops (64 Bit).

Die Logikbausteine außerhalb der CPU entlasten diese weitgehend von der gesamten Kommunikation und ermöglichen die prinzipielle Sichtweise eines durchgängigen Speichers. Praktisch erfolgt der Zugriff auf nichtlokale Speicher über vom System bereitgestellte Kommunikationsprimitive. (`shmem_get()`,
`shmem_put()`)

2.2 Low Level Performance

Für den Zugriff auf den Speicher entfernter Knoten werden die Basisfunktionen `shmem_put()` und `shmem_get()` bereitgestellt. Die Leistungsdaten liegen für `shmem_get()` bei ca 2μs Startup und 60 MByte/s Durchsatz, für `shmem_put()` bei 120 MByte/s Durchsatz bei ebenfalls 1-2 μs Startup. Die um den Faktor 2 geringe Bandbreite bei `shmem_get()` ist auf den verwendeten Übetragungsmechanismus zurückzuführen. Während `shmem_put()` die Daten direkt vom lokalen in den entfernten Speicher überträgt, bekommt bei `shmem_get()` der entfernte Prozessor zunächst eine Anforderung, durch die er veranlaßt wird, die gewünschten Daten zum lokalen Knoten zu übertragen.

2.3 Integration spezieller Architektureigenschaften

Nutzung atomarer Operationen - Die Behandlung von Anforderungen für Punkt-zu-Punkt-Kommunikation wird in Nachrichten-Queues verwaltet, die dem gemeinamen Zugriff aller Prozesse unterliegen. Der gegenseitige Ausschluß der Prozesse bei schreibenden Zugriffen auf diese Queue wird mit Hilfe atomarer Operationen realisiert, die durch Prozessor und Speicherverwaltungshardware unterstützt werden (Atomic Swap, Fetch&Increment). Atomic swap tauscht den Inhalt eines Registers unmittelbar mit einer Speicheradresse. Fetch&Increment liest innerhalb einer atomaren Operation den Wert eines speziellen, allgemein verfügbaren Registers, incrementiert diesen und schreibt das Ergebnis zurück.

Hardwareunterstützung für Barrier - Jeder Knoten der T3D besitzt neben den beiden Prozessoren weitere hochintegrierte Komponenten, die für die Speicherverwaltung und den gesamten Datentransport verantwortlich sind. Neben diesen Aufgaben liefern diese Bausteine auch eine effiziente Hardwareunterstützung für Barrier auf der Basis spezieller Register und kaskadierter AND-Gatter. Werden alle Knoten einer angeforderten Partition in ein Barrier einbezogen, ist die Auswertung des Barriers unabhängig von der Anzahl der teilnehmender Knoten innerhalb weniger Mikrosekunden (typisch 6 μs) vollzogen. Nehmen nicht alle Prozessoren einer Partition an einem Barrier teil, so wird die Barrieroperation softwaremäßig auf der Basis einer Baumtopologie bearbeitet.

Kommunikation auf Basis der Shared Memory Acces Library - Der MPI untergeordnete Kommunikationslayer benutzt für den Datenaustausch die Low Level Funktionen `shmem_put()` und `shmem_get()`.

Übertragung kleiner Paketgrößen im Nachrichtenheader - Sofern der reine Datenbereich einer Nachricht kleiner als 25 Byte ist, wird dieser innerhalb des Protokoll-Headers verschickt. Dadurch erübrigt sich die sonst anschließende eigentliche Kommunikationssphase.

Slotbasierte Verwaltung kollektiver Kommunikation - Für die Bearbeitung kollektiver Operationen wurde ein Protokollmechanismus implementiert, der auf separaten Slots basiert. Als Slot wird ein lokaler Datenbereich bezeichnet, der einem einzelnen remote Prozessor exklusiv zur Verfügung steht, um aktuelle Kommunikationsanforderungen zu deponieren. Jeder Prozessor richtet lokal für jeden weiteren verfügbare Prozessor einen solchen Slot ein. Durch die beschriebene Prozessor-Slot-Zuordnung können zeitaufwendige Lock-Operationen während der Behandlung eines Requestes weitgehend entfallen. Die eigentliche Kommunikation wird auf Punkt-zu-Punkt-Transfers abgebildet.

2.4 Ergebnisse unter MPI (EPCC)

Für die Messungen wurde das im vorangegangenen Abschnitt beschrieben MPI-Programm ohne weitere Modifikation verwendet. Somit gelten auch alle dort genannten Randbedingungen zur Zeitmessung.

2.4.1 Punkt -zu-Punkt-Kommunikation

Am Beispiel der Cray T3D-MPI-Implementierung kann sehr gut nochvollzogen werden, wie stark die Kommunikationsleistung vom verwendeten Softwareprotokolle abhängt. Folgende Übersicht stellt die Protokolle kurz vor.

Transfer (T) : Der Datenbereich der Nachricht ist so klein, daß er unmittelbar mit der Kommunikationsanforderung versendet wird.

Transfer-Acknowledge (TA) : Erweiterung des T-Protokolls um eine Bestätigung.

Request-Transfer-Acknowledge (RTA) : Der Empfänger veranlaßt den eigentlichen Datentransport nachdem er eine korrespondierenden Sende-Anforderung entdeckt hat (Empfängergetriebene Kommunikation - `shmem_get()`). Der vollständige Empfang wird durch eine Nachricht (Acknowledge) quittiert.

Request-Acknowlege-Transfer (RAT) : Der Empfänger signalisiert unmittelbar nach dem Erkennen eines zu ihm passenden Sende-Requests seine Bereitschaft (Acknowledge). Innerhalb des Acknowledge teilt er dem Sender mit, wo die Daten abzulegen sind. Daraufhin transportiert der Sender die Daten direkt ohne Pufferung in den angegebenen Zielbereich (Sendergetriebene Kommunikation `shmem_put()`). Nach dem vollständigen Transfer erhält der Empfänger vom Sender eine Bestätigung.

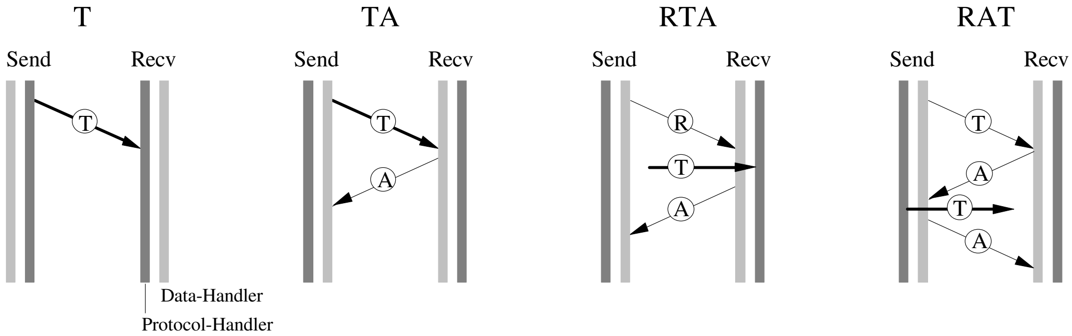


Abbildung 27: Punkt-zu-Punkt-Kommunikationsprotokolle unter MPI/EPCC

Für die Protokolle T und TA wird die "Nutzlast" der Nachricht ohne weitere Anforderung des Empfängers bereits in dort lokale Systempuffer übertragen (spekulativer Transfer). Ein solches Vorgehen ist aus Gründen der Speichereffizienz nur für relativ kleine Nachrichten geeignet.

Die Auswirkungen der verschiedenen Protokolle lassen sich sehr gut beobachten. Beim Transport sehr kleiner Nachrichten steigt ab einer Grenze von 24 Byte die benötigte Zeit sprunghaft an. Für die Modi Standard, Ready und Buffered erfolgt an dieser Stelle der Übergang vom T- zum TA-Protokoll, bei dem die Nachrichten spekulativ zum Empfänger gesandt werden. Bei Verwendung des synchronen Mode wird ab 24 Byte vom TA- zum RTA-Protokoll übergegangen.

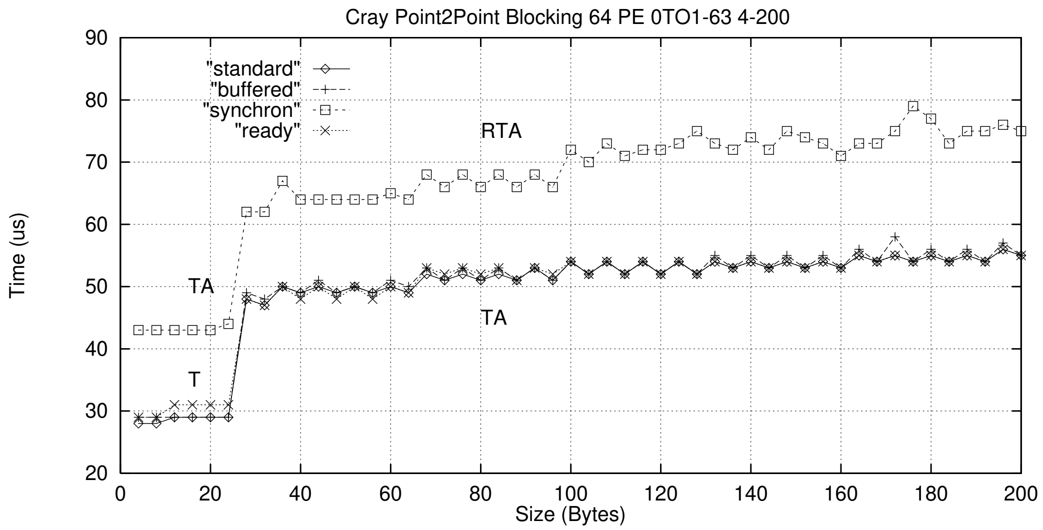


Abbildung 28: Punkt-zu-Punkt Kommunikation 4-200 Byte

Bis zu einer Datenmenge von 4 KByte sind die Unterschiede im Zeitverhalten der Modi Standard, Ready, Buffered und Synchron relativ gering. Ab der genannten Nachrichtengröße benötigt der synchrone Modus deutlich weniger Zeit als die anderen Modi und nähert sich bei sehr großen Datenmengen asymptotisch der Performance von `shmem_put()` an (ca. 120 MByte/s). Für das veränderte Zeitverhalten sind weitere Protokollwechsel verantwortlich. Der synchrone Mode wechselt von RTA- zum Sendergetriebenen RAT-Protokoll (`shmem_put()`). Die 3 verbleibenden Modi vollziehen ab 4 KByte Nachrichtengröße den Übergang zum RTA-Protokoll, bei dem die Nachrichten nicht mehr spekulativ übertragen werden, sondern explizit vom Empfänger anzufordern sind (Empfängergetrieben `sh_memget()`, max. 60 MByte/s).

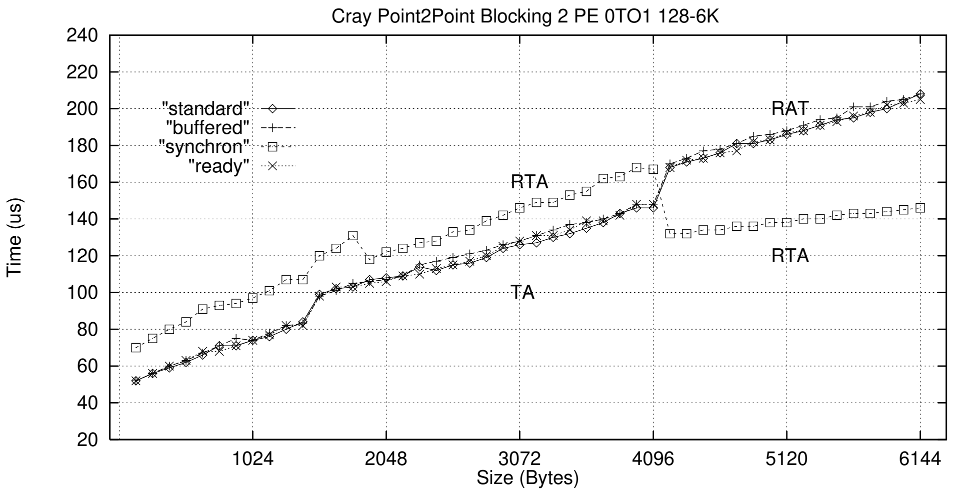


Abbildung 29: Punkt-zu-Punkt-Kommunikation 128 Byte - 6 KByte

Die Zeiten des pufferten Mode differieren nur minimal zu den Modi Standard und Ready. Das zeigt deutlich, das MPI auch bei den letztgenannten Modi eine Zwischenpufferung vornimmt. Im Gegensatz zu Buffered wird jedoch kein explizit vom Nutzer zugewiesener Puffer benutzt, sondern Systemspeicher. Bei dieser Implementierung verhalten sich die Modi Standard und Ready identisch.

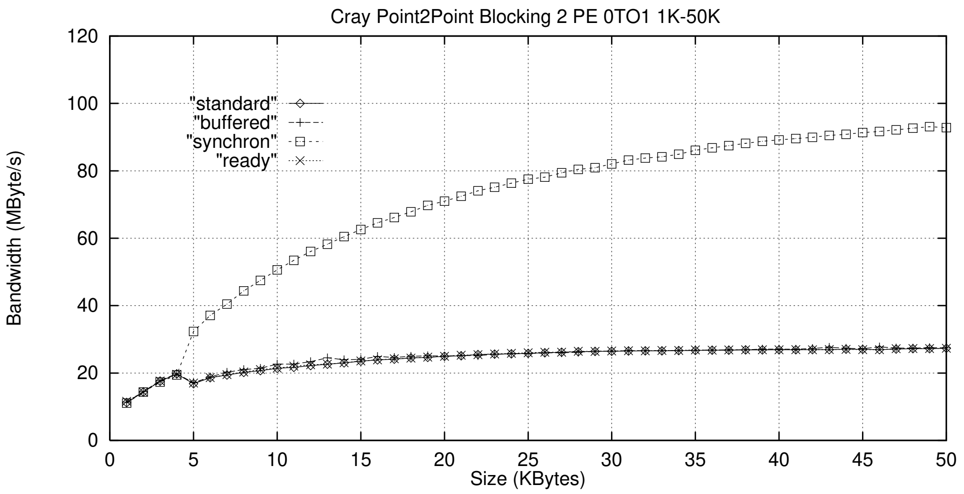


Abbildung 30: Bandbreite für Punkt-zu-Punkt Nachrichtengröße 1-50 KByte

Eine Abhängigkeit der Startup-Zeit und Kommunikationsbandbreite von der physischen Distanz der Knoten konnte nicht beobachtet werden.

2.4.2 Kollektive Kommunikation

Im Rahmen der kollektiven Kommunikation wurden als repräsentative Auswahl die MPI-Routinen für Barrier, Broadcast und Reduce untersucht.

2.4.3 Barrier

Ist eine Barriersynchronisation über die gesamte angeforderte Partition ausgedehnt, wird der Einfluß der Spezialhardware besonders offensichtlich. Im Bereich von 4 bis 64 Prozessoren wurden konstant 6μs zur Behandlung des Barriers ermittelt. Sobald ein Prozessor der Partition nicht in das Barrier einbezogen ist, erfolgt die Auswertung per Software auf Basis der unten beschriebenen Topologie.

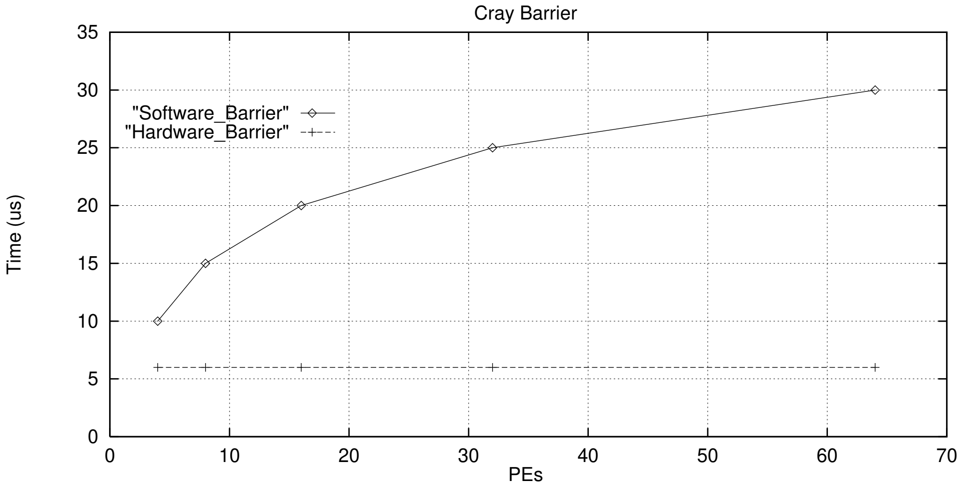


Abbildung 31: Software-Barrier 4 bis 64 Prozessoren

Sowohl Barrier als auch Broadcast verwenden einen unbalancierten Baum zur Propagierung der Daten. Durch die Struktur des Baumes verdoppelt sich bei einer Verteilung der Daten mit jedem Schritt die Anzahl gleichzeitig stattfindender Kommunikationen. Bei einer Verdoppelung der beteiligten Knoten ist somit nur ein weiterer zusätzlicher Schritt notwendig. Die folgende Grafik zeigt die Reihenfolge der Kommunikationsschritte für einen unbalancierten Baum mit 8 Knoten.

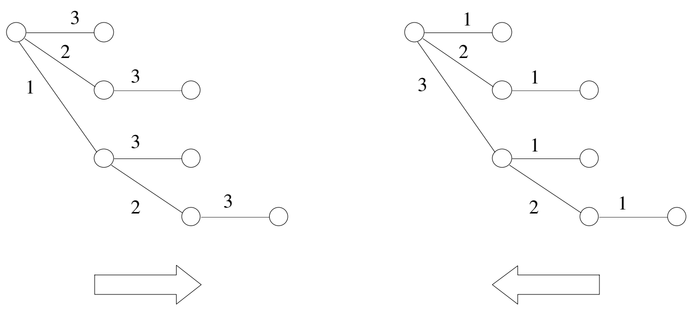


Abbildung 32: unbalancierter Baum für 8 Prozessoren

2.4.4 Broadcast

Das oben dargestellte Prinzip wurde durch Messungen mit `MPI_Broadcast()` praktisch bestätigt. Für kleine Nachrichtengrößen erhöht sich die benötigte Zeit bei Verdoppelung der Knotenzahl um einen konstanten Faktor.

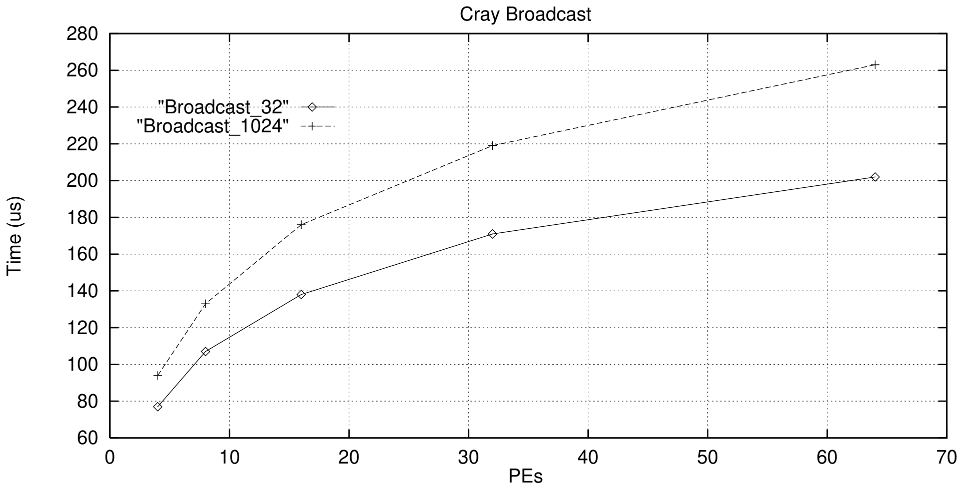


Abbildung 33: Latenzzeit Broadcast 32 Byte, 1 KByte

Eine Abweichung von diesem Schema wurde bei der Verwendung von 64 Prozessoren und Datenmengen über 1 Kbyte festgestellt. Dort stieg die benötigte Zeit stärker als erwartet an.

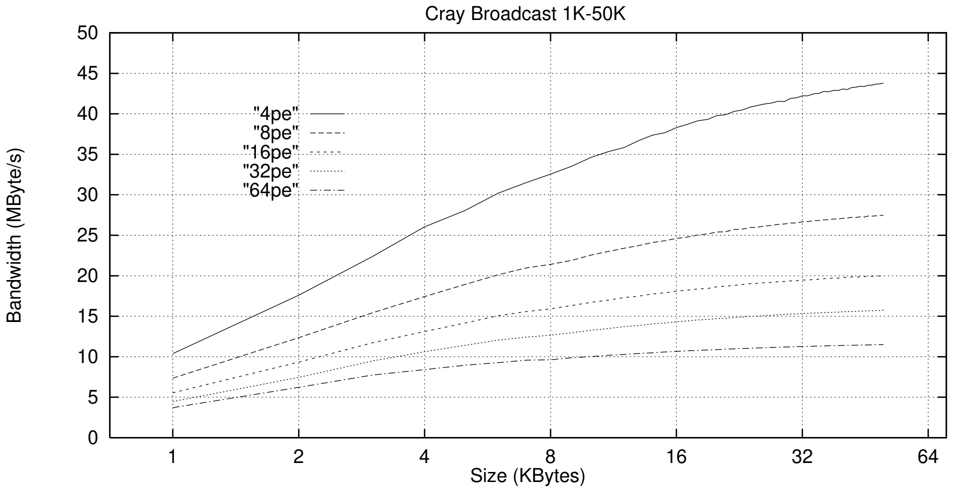


Abbildung 34: Bandbreite Broadcast 1-50 KByte, 4-64 PE

2.4.5 Reduce

Die Propagierung von Reduce basiert auf einem binären Baum. Die Anzahl der benötigten Kommunikationsschritte für eine vollständige Propagierung ist gegenüber einem unbalancierten Baum höher. Der Vorteil dieser Struktur liegt in einem geringeren Pufferbedarf, da jeder Knoten höchstens zwei Kindknoten und einen Elternknoten besitzen kann.

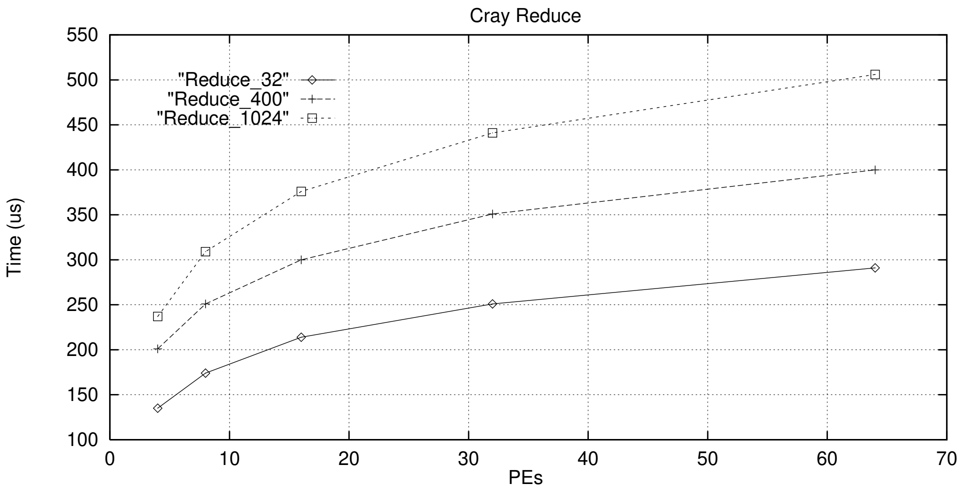


Abbildung 35: Latenzzeit Reduce 32,400,1024 Byte, 4-64 PE

Als Reduktionsoperation wurde eine Summenbildung auf der Basis von MPI_DOUBLE durchgeführt. Der gegenüber Broadcast erhöhte Zeitbedarf dürfte nur teilweise auf die Ausführung der Reduktionsoperation zurückzuführen sein.

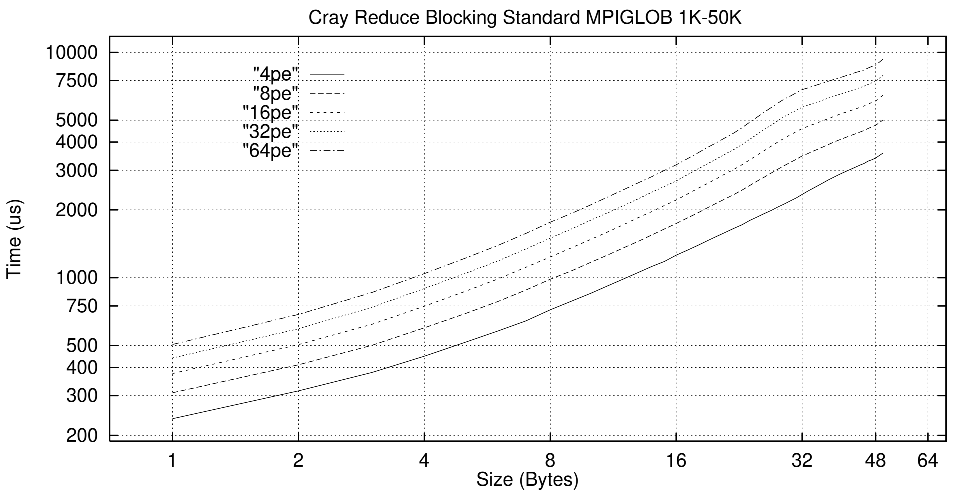


Abbildung 36: Latenzzeit Reduce 1-50 KByte, 4-64 PE

2.4.6 Schlußbemerkung

An dieser Stelle sei den Kollegen des Conrad-Zuse-Instituts Berlin gedankt, die innerhalb kurzer Zeit und erfreulich unbürokratisch den Zugang zur dort installierten Cray T3D-256 ermöglichten und halfen, erste Anfangshürden zu überwinden.

Literatur

[1] S. Burkhardt u.a., *Parallele Rechnersysteme - Programmierung und Anwendung*, Verlag Technik GmbH, Berlin München, 1993

[2] Message Passing Interface Forum, *DRAFT Document for a Standard Message Passing Interface*, Stand 30.6.93, 1993

[3] A. Geist u.a., *PVM 3.0 user's guide and reference manual*, Oak Ridge National Laboratory, Feb. 93

[4] V.Karamcheti u.a. , *FM: Fast Messaging on the Cray T3D*, University of Illinois at Urbana-Champaign, Department of Computer Science, 1995

[5] J.J. Dongarra, T. Dunigan, *Message-Passing Performance of various Computers*, Oak Ridge National Laboratory, Computer Science and Mathematics Division, 1996

[6] K. Cameron, L. C. Clarke, A.G. Smith, *Using MPI on the Cray T3D*, Edinburgh Parallel Computing Centre, 1995

[7] K. Cameron, L. C. Clarke, A.G. Smith, *CRI/EPCC MPI for Cray T3D*, Edinburgh Parallel Computing Centre, Edinburgh EH9 3JZ, 1995

[8] Cray Research, Inc. , *Cray T3D System Architecture Overview*, March 1993

Message Passing Efficiency on Shared Memory Architectures

Thomas Radke

tomsoft@informatik.tu-chemnitz.de
http://noah.informatik.tu-chemnitz.de/members/radke/radke.html

1 Introduction

Especially in the framework of the newly founded "Sonderforschungsbereich SFB 393 : Numerische Simulation auf massiv parallelen Rechnern", other parallel architectures – namely shared memory systems – are to be investigated for their ability to efficiently solve the addressed problems. Two principle ways are possible to implement parallel programs on shared memory architectures. First, one can use multithreading as the basic programming model for his/her implementation[1]. This guarantees the best use of the facilities of the underlying hardware. The other way is to use a message passing interface, that resides on top of multithreading, and write a CSP program.

The first way was evaluated at our department by porting a module of an existing message passing program (FEM solver) onto a shared memory system[2]. Tests showed that good efficiency results can be obtained for small problem sizes mapped onto few processors. But the port took a considerable amount of time, and writing a multithreaded program appeared to be even harder than defining the same problem with the message passing paradigm.

As there are comprehensive experiences at the University of Technology Chemnitz with message passing programming on the Parsytec massively parallel computer series, we concluded that the second approach should be favourized in the SFB 393.

This paper describes the potential facilities for message passing on 2 shared memory systems available at the Department of Computer Science: the KSR1 from Kendall Sqare Research, and Compaq's ProLiant 4000. The theoretical efficiencies of a message passing interface are given as communication bandwidth, communication latency, parallelity of communication and computation, and the implementation of global operations. Efficieny losses due to the implementation of the underlying multithreading functionality are not studied here in detail, the main intention was to evaluate the shared memory hardware architecture.

2 The KSR1 Architecture

The KSR1 parallel shared memory computer (built by Kendall Square Research) consists of 8 processor nodes. Each node has a KSR1 processor (20 MHz clock, RISC architecture, developed at Kendall Square Research) with a first-level cache of 0.5 MBytes, half for instructions, half for data. The node's main memory is 32 MBytes of size. It is called the local cache, but the local processor has access to local caches of other nodes too. This is realized by the KSR ALLCACHE architecture: all nodes are chained in a ring (called ALLCACHE engine). Any access to remote caches is done over this ring, with a physical bandwidth of 1 GByte/s. The coherency of global memory is ensured by a local cache directory in every node.

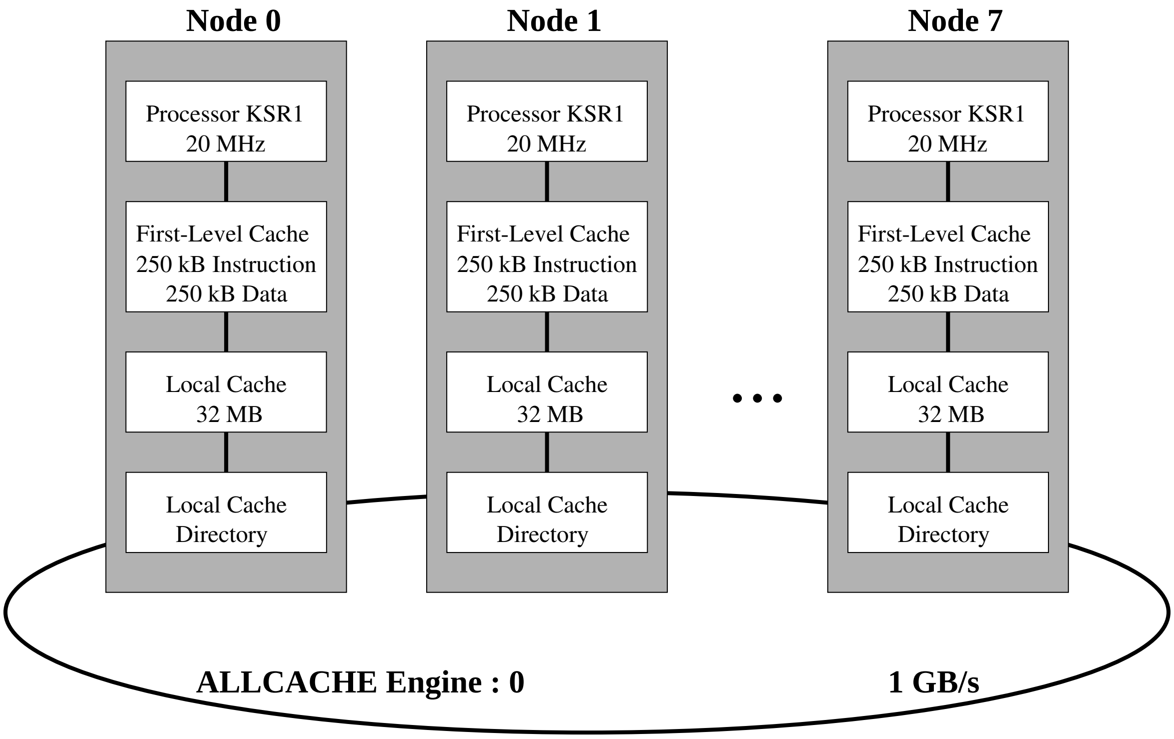


Figure 1: Architecture of the KSR1 computer [3]

Because there is no explicite global main memory, the KSR1 computer is classified as COMA architecture (cache only memory access).

3 The Compaq ProLiant 4000 Architecture

The ProLiant 4000 is a symmetric multiprocessor PC system, built by Compaq. It has 4 processors (Intel Pentium 66 MHz, 8 KBytes on-chip cache for instructions and data respecticely). Each processor module is supplied with its own second-level cache (256 KBytes). Access to main memory is realized by the Compaq TriFlex bus, with a peak bandwidth of 267 MBytes/s.

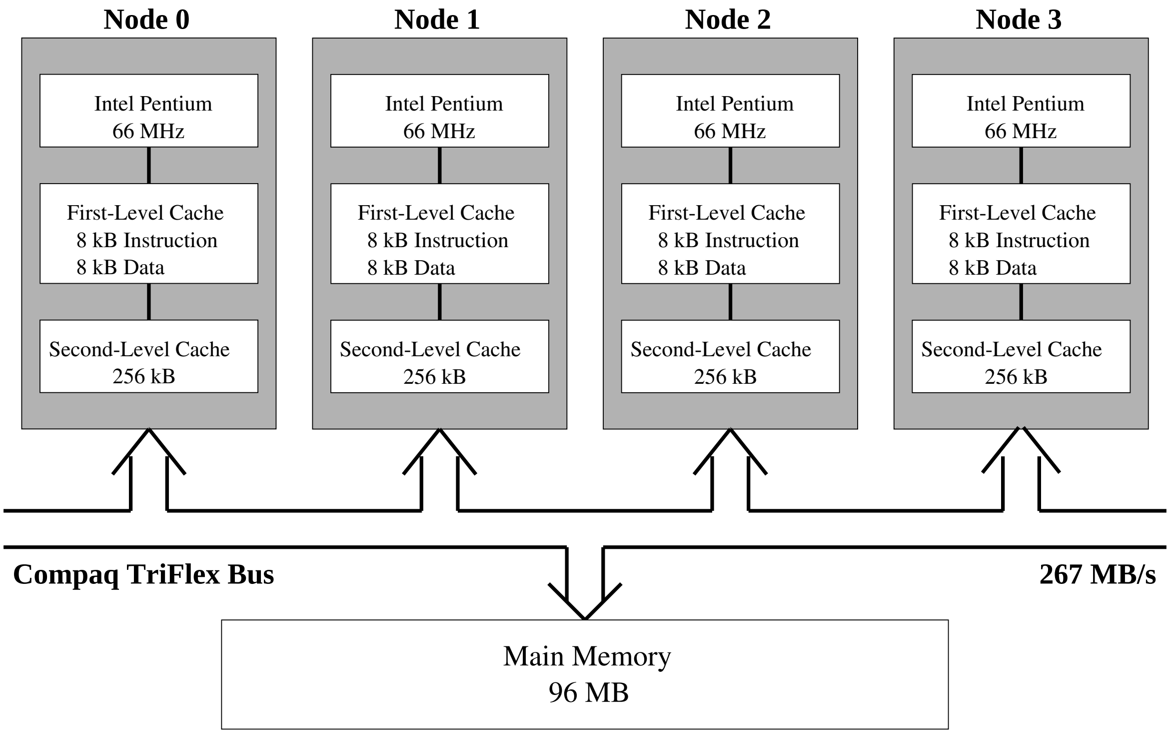


Figure 2: Architecture of the Compaq ProLiant 4000

The Compaq ProLiant 4000 has a UMA architecture (uniform memory access). All processors have the same access time to main memory.

4 The Multithreading Programming Model

Multithreading extends the process concept of classical UNIX operating systems by the opportunity, that a process itself can spawn several threads of control at runtime. The threads have an own stack and register set, so that they can run in parallel. The process address space is shared between all threads. Global variables can be used for inter-thread-communication. Synchronization facilities (mutexes, semaphores, condition variables, barriers) are used to implement controlled access to shared variables and to synchronize threads at specific program points.

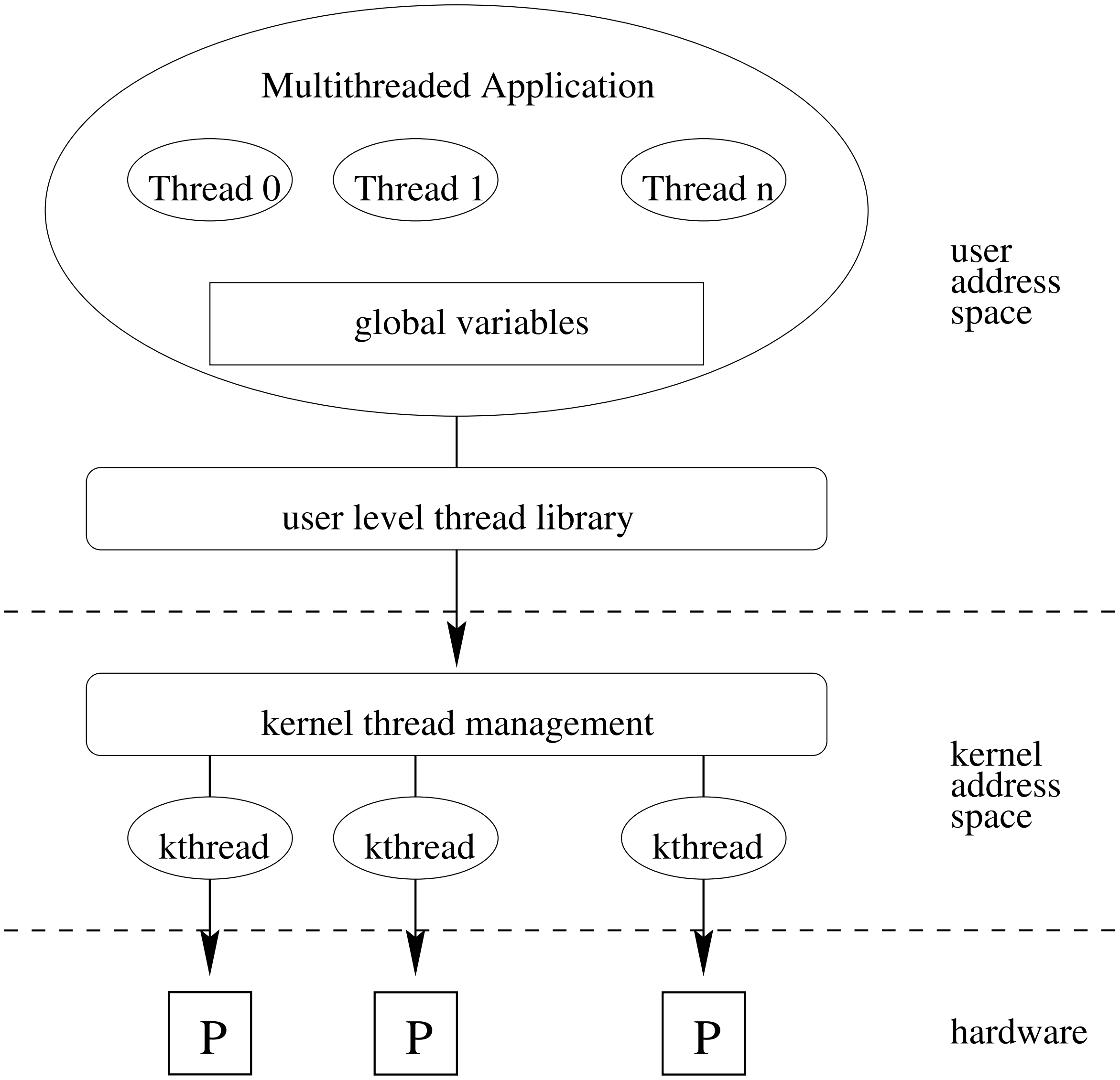


Figure 3: Threads in a multithreaded process

5 Message Passing on Top of Multithreading

The emulation of message passing in a multithreaded environment can be expressed as following:

- Threads become the nodes of the message passing program.
- Communication is done via synchronization of threads and simply copying of the data from the sending node to the receiver.

5.1 Threads as Nodes

In distinction to message passing programming under PARIX, where the whole main program is duplicated onto all processor nodes at load time, threads begin their execution in a start function from within the same process. This could be done implicitely by calling a function

```
int MP_Init (unsigned int num_nodes, void (*start_fn) (void *arg));
```

with *start_fn* specifying the function to start execution of the thread with the single argument *arg*, and *num_nodes* determining the number of nodes to create.

Another distinction is the declaration of global variables. In a PARIX program, all global variables are visible to the local node only. In multithreaded programs these variables are shared between all nodes. A possible way to solve this problem is the introduction of private data. Each variable with an attribute *private* becomes thread-specific, i.e., every thread gets a local copy of the variable. This solution requires the explicite declaration of those variables by the application programmer and some changes in the compiler to support the handling with thread-specific data.

5.2 Node-to-Node Communication

Communication between nodes is very easy to implement because of the common address space of all threads. A (synchronous) communication operation requires the synchronization of the participating nodes and a data transfer operation (a simple memcpy) from the sender to the receiver(s).

Prototypes for a basic `SendMessage() / RecvMessage()` function are introduced in the following:

```
typedef struct {
    sem_t sendReady, recvReady;
    void *message;
    unsigned int bytes;
    unsigned int *transfered;
```

```
} Channel_t;

unsigned int SendMessage (void *message, unsigned int bytes, Channel_t *channel)
{
    unsigned int transfered;

    channel->bytes = bytes;
    channel->message = message;
    channel->transfered = &transfered;
    sem_signal (&channel->sendReady);
    sem_wait (&channel->recvReady);
    return (transfered);
}

unsigned int RecvMessage (void *message, unsigned int maxBytes, Channel_t *channel)
{
    unsigned int transfered;

    sem_wait (&channel->sendReady);
    transfered = *channel->transfered = min (maxBytes, channel->bytes);
    memcpy (message, channel->message, transfered);
    sem_signal (&channel->recvReady);
    return (transfered);
}
```

A *channel_t* structure is used for synchronous uni-directional communication. It contains a semaphore for the synchronization of the sending and the receiving node respectively, a pointer to the message to be passed, the size of the message in bytes, and a help variable the store the actual transfered number of bytes.

The `SendMessage()` function prepares the channel structure (sets the pointer to the message and the size), signalles the receiver that it is ready to send, and waits for the receiver to fulfil the operation. The receiver waits until the sender is ready to send, and then copies the message into its own buffer. It sets the actual length of the transfered message (the minimum of the requested sizes of the sender and the receiver) and signalles the sender for completion.

The time needed to synchronize the sender and the receiver depends on the implementation of the semaphore functions. If it is neglected, only the time of the memcpy operation remains. So the memory bandwidth of the underlying hardware architecture determines the peak communication bandwidth for a message passing emulation.

This was examined on both the KSR1 and the ProLiant 4000, with the packet size and the number of active processors as parameters:

The test program does the following:
A packet of fixed size is copied from its source location to the destination. The source array was initialized before by processor i, so that its contents is held in the local

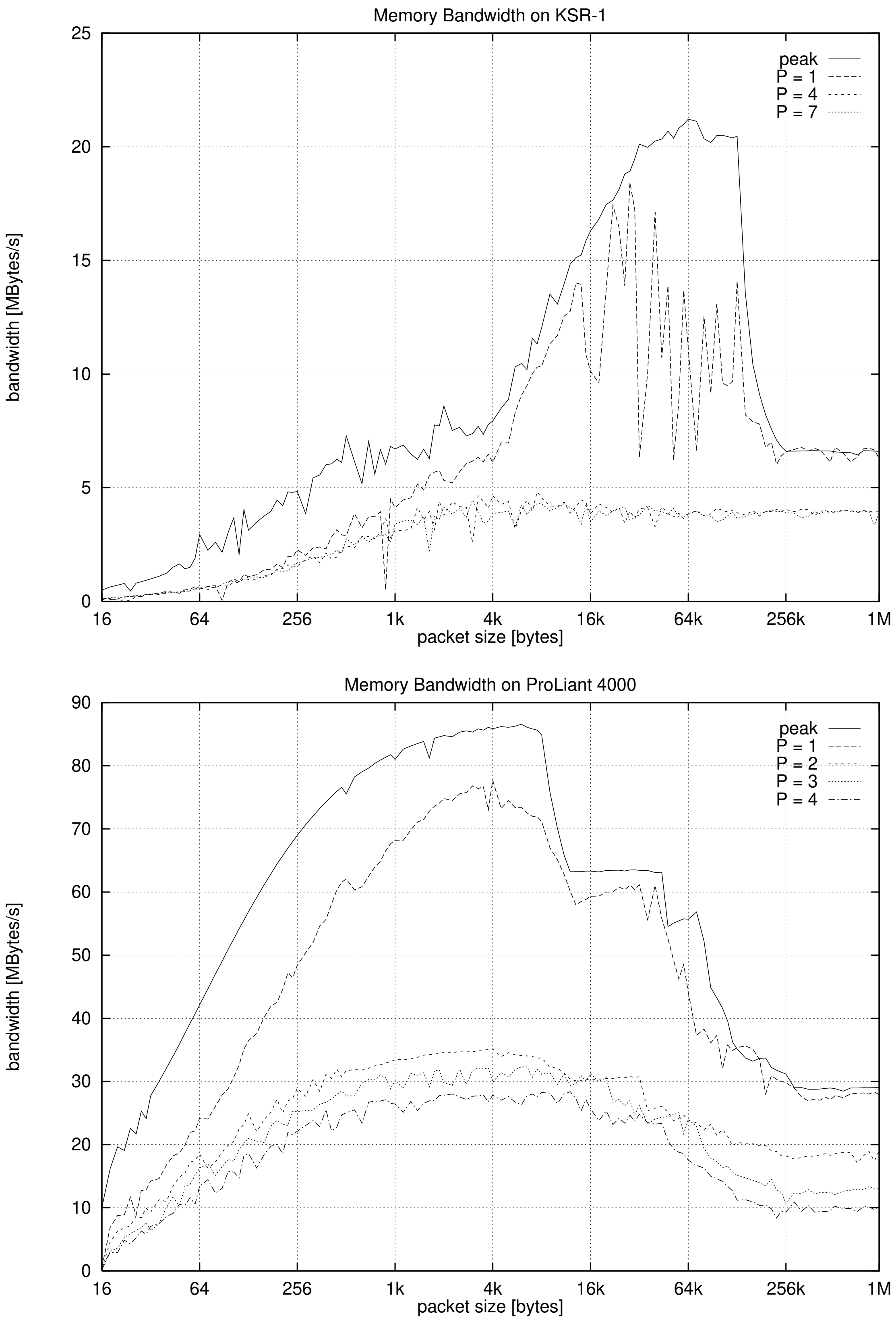


Figure 4: Memory bandwidth on the KSR1 and the ProLiant 4000

processor's cache (if the packet fits in). The target array is initialized too, but by processor *i + 1*, so the cache contents of this processor has to be invalidated during the memcpy operation.

Curves for a different number of parallel memcpy'ing processors are shown. On the KSR1 the curves for 2...7 processors are nearly equal (only 2 curves are shown here). That means, the ALLCACHE engine bandwidth is sufficient for all connected processor nodes. This is not the case on the ProLiant 4000: here the processors have to wait until they become the bus master to access the main memory. The more processors are used, the smaller is the achieved bandwidth.

The peak bandwidth curve was obtained over 100 iterations of the memcpy operation, i.e., the data got loaded into the cache at the first access and was fetched from there in all other operations.

The communication delays for small packet sizes are determined by the overhead of the function calls. To compare them with other machines, they are listed as the bandwidth in tables 1 and 2 again (for the single processor case):

packet size [bytes]	bandwidth [MBytes/s]
16	10.00
24	22.59
32	27.73
64	42.15
96	50.93
128	56.86
256	69.00

Table 1: bandwidth for small packet sizes on the KSR1

packet size [bytes]	bandwidth [MBytes/s]
16	0.49
24	0.45
32	0.98
64	2.93
96	3.03
128	3.13
256	4.83

Table 2: bandwidth for small packet sizes on the ProLiant 4000

6 Influence of Memory Latency on Computation

As seen in the previous chapter, the memory bandwidth of the ProLiant 4000 significantly decreases with the number of processors concurrently requesting the memory bus. This fact has some influence on the computation too: if the operands of a numerical operation have to be fetched from main memory, the computation can be delayed due to bus access conflicts.

A test program was written to examine the influence of the memory latency on computation. Several numerical operations were executed on each processor – fully independently from others – with the double vectors a and b and a constant C as operands. The measured execution times were set in proportion to the time on one processor.

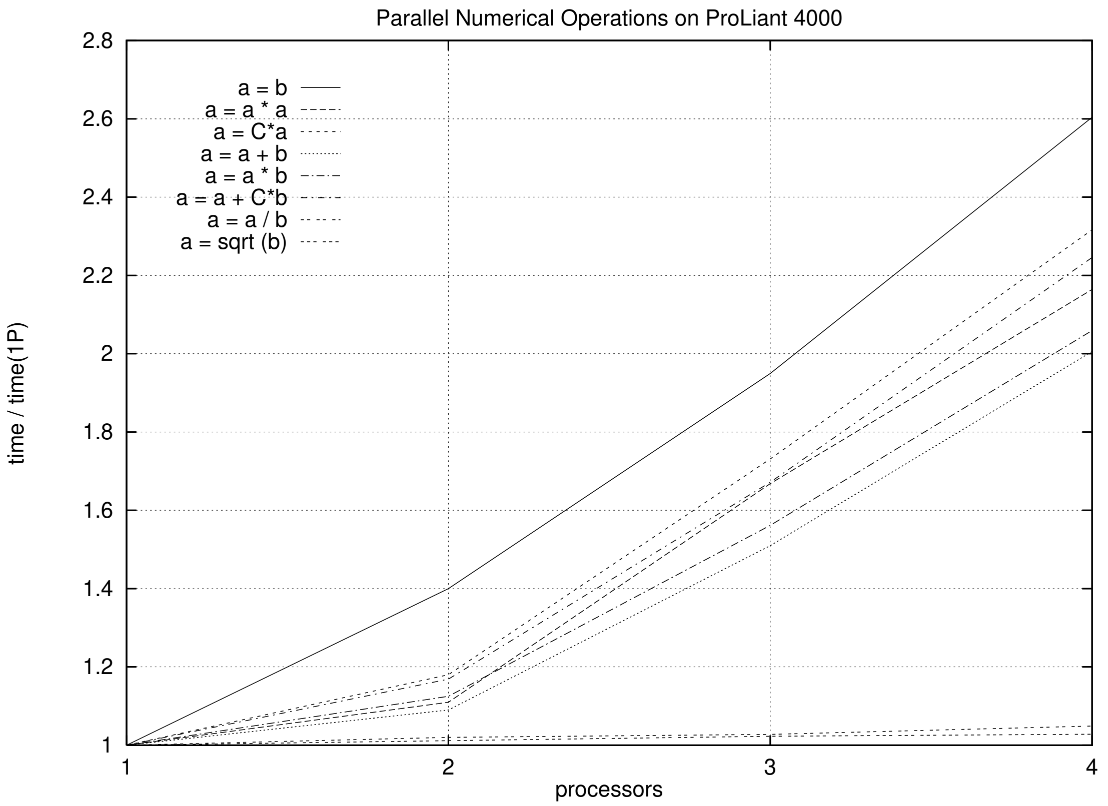


Figure 5: Memory latency influence on independent computations on the ProLiant 4000

A simple assignment of vector b to vector a takes 2.6 times on 4 processors in contrast to one processor. This operation is analogous to the memcpy curve depicted in the previous chapter. Other numerical operation, like the sum or the product of two vectors, scaling, or elimination, require twice the time of a serial computation. Only in heavyweight calculations (division, square root) the theoretical speedup is reached.

7 Global Operations

To examine the efficiency of global operations, two algorithms for the computation of a global sum of double vectors were implemented:

- a serial version with barrier synchronization
  The first thread that enters the barrier initializes a global operation structure by setting the pointer to its local vector to an intermediate result pointer, and suspends its execution to wait for the other threads. These add their own vector to the intermediate sum vector (the vector of the first thread is used for this) as they reach the barrier, and suspend themselfes too. The last thread computes the final sum vector and then awakenes all suspended threads by a broadcast on the barrier. Now all threads copy the result into its own local result vector.

- a parallel version via hypercube communication
  A hypercube topology is used to exchange the vectors between adjacent nodes. Both the local and the remote vector are added and sent to the next neighbour. This version has a parallelity degree of D (the dimension of the hypercube).

The following diagrams compare both versions on the KSR1 and the ProLiant 4000:

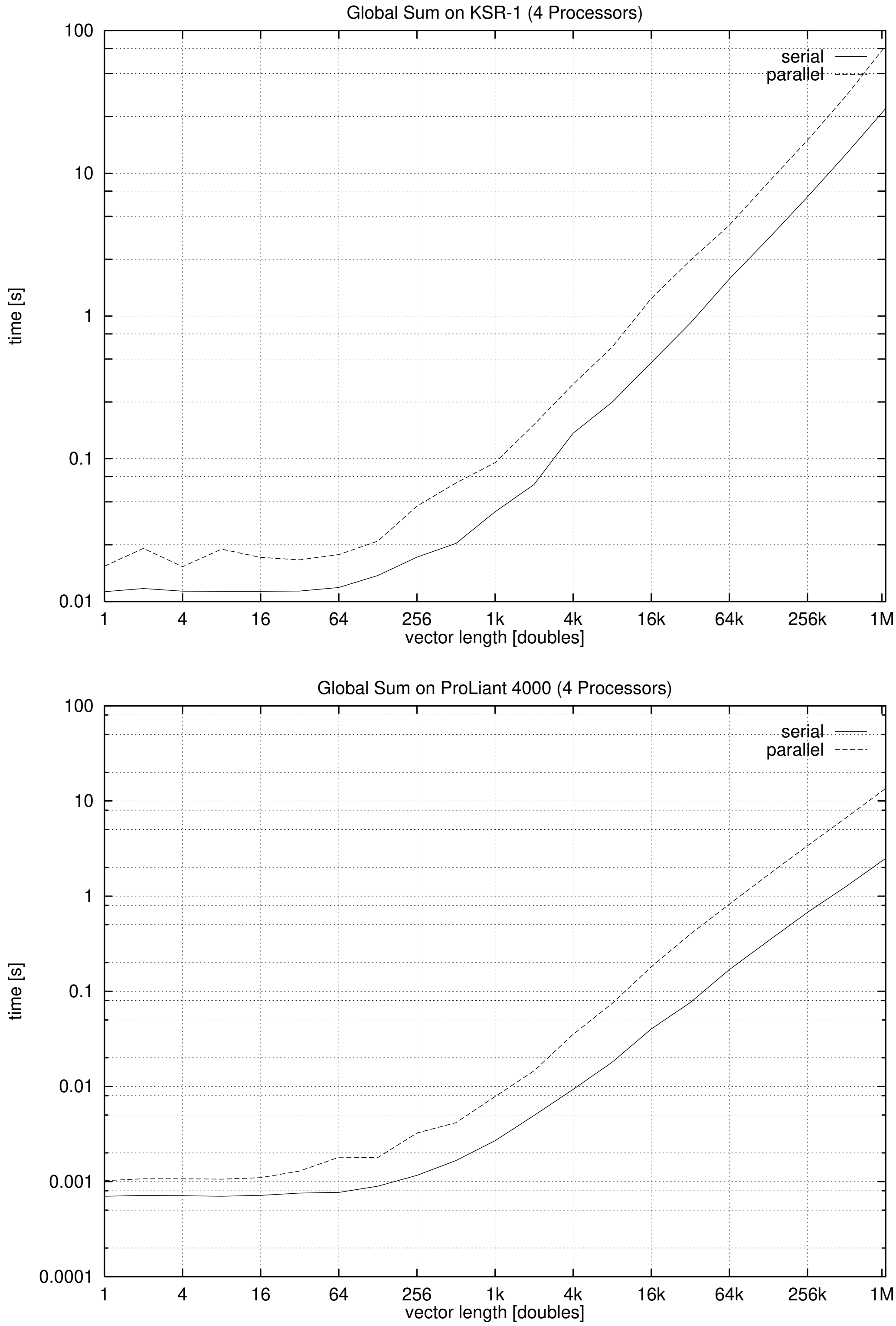


Figure 6: Serial and parallel computation of the global sum on the KSR1 and the ProLiant 4000

As can be seen, the serial algorithm works faster than the parallel one on both machines. There might be two reasons for that: at first, the global sum is a very cheap numerical operation (see figure 6). More complex operation should bring both curves nearer to each other. And secondly, only 4 processors were used (there were not more processors available on the KSR1 to build a hypercube of dimension 3). The more processors calculate in parallel the higher should be the speedup of the parallel algorithm.

8 Conclusions

The communication bandwidth achieved in a message passing emulation on shared memory architectures is mainly determined by their memory bandwidth and therefore is often much higher compared with distributed memory architectures. An important demand to the memory system is scalability: a processor's access to global memory should not be delayed by concurrent independent accesses of other processors. The use of local processor caches significantly increases the speedup, so caches should be as large as possible.

Computations should contain complex operations rather than simple, if the memory bandwidth of the underlying architecture is not sufficient for parallel work of all processors. Otherwise a decrease of speedup is the consequence, even if the operations are fully independently of each other.

For global operations there exists a break-even point, where a serial implementation is still faster than a parallel one (based on the model of true message passing implementations, e.g. hypercube communication). There are very efficient global synchronization methods for threads (e.g. barriers) which can overweigh the parallel versions with their several local synchronizations.

References

[1] T. Radke. *Parallel Programming with User-level Threads*, Technical Report TUCZ / RA-TR-96-3, Computer Science Department, University of Technology Chemnitz, 1996.

[2] L. Grabowsky. *Parallel FEM implementations on shared memory systems*, Technical Report TUCZ / RA-TR-96-4, Computer Science Department, University of Technology Chemnitz, 1996.

[3] *KSR1 Principles of Operations*, Kendall Square Research Corporation, Massachusetts, 1993.

[4] *Pentium^{TM} Processor User's Manual*, Intel Corporation, 1994.

MPI-Portierung eines FEM-Programmes

Uwe Beyer
ube@informatik.tu-chemnitz.de
http://noah.informatik.tu-chemnitz.de/members/beyer/beyer.html

Andreas Munke
ambu@informatik.tu-chemnitz.de
http://noah.informatik.tu-chemnitz.de/members/munke/munke.html

Zusammenfassung

Im Rahmen der Leistungsanalyse des Message Passing Interface (MPI) wurde ein Programm zur Lösung von 2-dimensionalen Potentialproblemen, welches auf der Finiten-Elemente-Methode (FEM) basiert, auf PowerMPI portiert. Ausgehend von dieser praktischen Erfahrung werden Erkenntnisse über die Handhabbarkeit von MPI, den entstandenen Programmieraufwand und das Laufzeitverhalten der MPI-Version des Programmes abgeleitet.

1 Zielstellung

Bei den Untersuchungen eines neuen Programmierwerkzeuges, einer verbesserten Rechnerarchitektur oder eines innovativen Standards hinsichtlich der Leistungsmerkmale, Funktionalität und Verwendbarkeit werden sehr oft spezielle und zugeschnittene Tests durchgeführt. Die Resultate einer solchen Analyse sind daher meist aussagekräftige und konkrete Werte, anhand derer es möglich ist, das untersuchte Objekt zu bewerten und eventuell mit vergleichbaren Objekten in Relation setzen zu können. Bei diesen Untersuchungen kommt es darauf an, definierte Bedingungen zu schaffen, um einzelne Parameter oder das Zusammenspiel mehrerer Parameter gezielt zu steuern und die Reaktionen des Systems zu ermitteln. Neben der direkt meßbaren Performance des zu untersuchenden Objektes stehen aber auch dessen Eigenschaften bei der Ausführung einer realen und komplexen Aufgabe im Vordergrund der Betrachtungen. Interessant und wichtig ist dies, da bei einem späteren praktischen Einsatz nicht mehr das Objekt selbst, sondern nur noch die Lösung eines Problems im Mittelpunkt steht. Das untersuchte System hat dann ausschließlich seine Funktion zu erfüllen und sich in das Gesamtprojekt zu integrieren. Eine Leistungsanalyse sollte daher auch den nötigen Aufwand bei einem Einsatz, die auftretenden Probleme und das Laufzeitverhalten in komplexen Applikationen ermitteln, um praktische Erfahrungen im Umgang mit dem Objekt zu gewinnen.

Vor diesem Hintergrund entstand im Rahmen der Leistungsanalyse des Message Passing Interface-Standards (MPI) die Aufgabe, die existierende PARIX-Version des Programmes SPC-PMPo2 auf MPI zu portieren. Diese Applikation ist ein Programm zur parallelen Lösung von 2-dimensionalen Potentialproblemen auf der algorithmischen Basis der Finiten-Elemente-Methode (FEM). Sowohl das Programm selbst als auch die unterstützenden Bibliotheken wurden an der Fakultät für Mathematik der Technischen Universität Chemnitz-Zwickau von der Arbeitsgruppe "Scientific Parallel Computing" entwickelt.

Von dem Erstellen der MPI-Version dieses Programmes wurden nun exemplarische Aussagen über die Handhabbarkeit von MPI und den nötigen Aufwand zur Lösung dieser Aufgabe erwartet. Desweiteren sollten Laufzeitvergleiche zwischen den beiden Varianten Auskunft über den Geschwindigkeitsnachteil der MPI-Realisierung geben. Zu beachten ist hierbei natürlich, daß die ermittelten Resultate nicht den Vorteil der Allgemeingültigkeit besitzen. Dafür stellen sie aber die Erfahrungen über den Umgang mit MPI bei der Lösung einer konkreten Aufgabe dar und können somit als Richtwert für vergleichbare Projekte dienen.

2 Portabilität paralleler Applikationen

Ein bedeutendes Problem bei der Entwicklung von parallelen Applikationen sind die Schwierigkeiten beim Schaffen von portablen Projekten. Das heißt von Programmen oder Bibliotheken, die ohne großen Aufwand auf mehreren unterschiedlichen, parallelen Rechnersystemen abgearbeitet werden können. Komplikationen bereiten hierbei die verschiedenen Konzepte für den Zugang der Applikation zum Kommunikationssystem des Rechners. Dieser Zugang ist insbesondere bei Systemen die auf dem Nachrichtenaustausch (Message-Passing) zwischen Prozessorknoten beruhen, von grundlegender Bedeutung. Aber gerade die Kommunikationrufe an die jeweilige Hardware unterscheiden sich durch das Fehlen von herstellerübergreifenden Standards. Damit muß eine portable Applikation oder ein Bibliothekspaket immer eine Schicht für die Anpassung an die Gegebenheiten der entsprechenden Parallelrechnersysteme enthalten. Diese Anpassungsschicht muß im Normalfall bei jeder Portierung auf einen anderen Parallelrechner neu implementiert werden. Um den dafür notwendigen Aufwand gering zu halten, wird oft die Funktionalität des architekturspezifischen "Bindegliedes" sehr klein gehalten und auf grundlegende Funktionen beschränkt. Alle benötigten komplexeren Routinen, wie zum Beispiel globale Kommunikationsrufe, müssen nun innerhalb der Applikation oder Bibliothek auf dieses kleine Interface abgebildet werden, welches mit den Kommunikationsprimiven des Parallelrechners arbeitet. Sollten vergleichbare komplexere Funktionen auch von dem Rechnersystem angeboten werden, so ist es dann nicht möglich diese zu verwenden. Dies verursacht einerseits einen deutlich erhöhten Programmieraufwand und andererseits bei den meißten Systemen einen Laufzeitverlußt, da die komplexen, systeminternen Funktionen effizienter auf die Hardware abgebildet werden können, als es bei den selbstgeschriebenen möglich ist.

Mit dem Formulieren und Implementieren des Message Passing Interface-Standards wurde ein systemübergreifendes Werkzeug geschaffen, mit dem es in Zukunft möglich sein wird, parallele Programme zu erstellen, die auf eine Anpassung an die spezielle Architektur verzichten können. Der Programmierer arbeitet hierbei mit architekturunabhängigen, von der MPI-Bibliothek zu Verfügung gestellten Kommunikationsfunktionen. Die Anpassung an die Spezifika der aktuellen Umgebung wird nun von MPI übernommen. Der Anwendungsprogrammierer muß lediglich die MPI-Bibliothek für die zu bearbeitende Architektur zur Verfügung stellen und kann die Applikation ohne Änderung des Quelltextes auf verschiedenen Parallelrechnerarchitekturen abarbeiten.

Zur Zeit existiert schon eine große Anzahl von MPI-Implementationen für die unterschiedlichsten Message-Passing- oder Virtual-Shared-Memory-Architekturen. So zum Beispiel für Workstation-Cluster oder auch den GC/PowerPlus.

Dieser Vorteil muß allerdings durch das Akzeptieren eines Nachteiles erkauft werden. So stützen sich die meisten MPI-Implementationen auf die vorhandenen, architekturspezifischen Programmierwerkzeuge und legen damit eine abstrahierende "Zwischenschicht" über all diese Tools. Dieses Einfügen von zusätzlichen Funktionsaufrufen und Rechenaufwand bringt natürlich einen gewissen Laufzeitverlust mit sich. So basiert zum Beispiel die MPI-Implementierung für den GC/PowerPlus (PowerMPI) auf der vorhandenen PARIX-Kommunikationsbibliothek. Ein Programm, welches nun PowerMPI nutzt, hat damit von Haus aus einen kleinen Geschwindigkeitsnachteil gegenüber einem direkt für PARIX geschriebenen Programm.

3 Die Struktur der Applikation

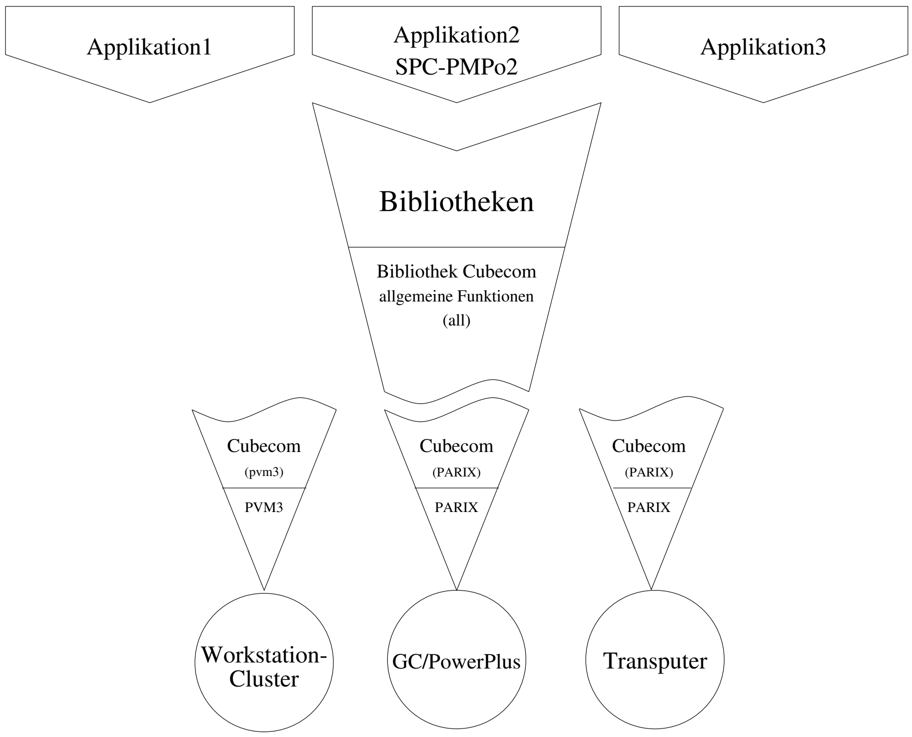


Abbildung 1: Die Applikation SPC-PMPo2 und die zugrundeliegenden Bibliotheken

In Abbildung 1 ist der hierarchische Aufbau der Applikation dargestellt. Die Kommunikation wird von der Bibliothek Cubecom übernommen. Diese Bibliothek besteht aus einem architekturunabhängigen Teil (all) mit höheren Funktionen (z.B. für den globalen Austausch von Werten oder für das Ausführen von systemweiten Operationen über gemeinsamen Daten) und einem Teil mit den Anpassungen an die jeweilige

Plattform. Der architekturspezifische Teil wird durch wenige, primitive Kommunikationsfunktionen übernommen, für die eine einheitliche Schnittstelle definiert wurde. Die Funktionalität dieses Teiles besteht hauptsächlich aus Funktionen für die Initialisierung und den Abbau der Kommunikation sowie aus Routinen für das Senden einer Anzahl von Worten an einen bestimmten, direktadressierten Knoten, oder über einen Link der zugrundeliegenden HyperCube-Topologie. Für die Portierung des FEM-Programmes, oder allgemeiner: der Bibliotheken, war es dadurch ausreichend, die Funktionen des spezifischen Teils zu modifizieren oder neu zu implementieren. Durch diese Herangehensweise wird die MPI-Version des entsprechenden Teils der Bibliothek CUBECOM zu einem mit den anderen Varianten gleichberechtigten Teil, welcher jedoch die Anpassung der Bibliothek an mehr Architekturen als nur den GC/PowerPlus bietet. Von den komplexeren Funktionen und Konzepten, wie z.B. virtuellen Topologien oder globalen Operationen, die MPI zur Verfügung stellt, macht dieser Lösungsansatz allerdings keinen Gebrauch, sondern beschränkt sich auf die Verwendung der grundlegenden Kommunikationsroutinen.

4 Die MPI-Implementation der Bibliothek CUBECOM

4.1 Die Portierung der PARIX-Funktionen

Wie jedes Kommunikationssystem, das auf dem Austausch von Daten in Form von Nachrichten beruht, stützen sich sowohl PARIX als auch MPI auf ein sehr ähnliches Konzept von Basisfunktionen. Bei beiden Systemen besteht dieses aus dem synchronen oder asynchronen Senden einer Menge von Bytes an einen bezeichneten Empfänger, welcher ebenfalls synchron oder ansynchron vom Sender eine Nachricht erwartet. Die Adressierung des Empfängers oder Senders beruht letztendlich auf der Angabe seiner eindeutigen Identifikation (z.B. seiner Knotennummer). Virtuelle Topologien, die mit beiden Systemen aufgebaut werden können, erleichtern dem Programmierer gegebenenfalls die Berechnung dieser Identifkation. Das die grundlegenden Kommunikationsprimitive beider Systeme eine sehr große Ähnlichkeit aufweisen wird in der Darstellung 2 gezeigt.

Die dargestellte MPI-Funktion `MPI_SEND` und die PARIX-Funktion `SendNode` übermitteln beide eine Anzahl von Bytes (PARIX) oder Elementen eines bestimmten Types (MPI) an den angegebenen Zielknoten. Die Kommunikation erfolgt bei diesen Funktionen synchron. Das heißt, daß auf dem Zielprozessor eine entsprechende Empfangsfunktion aufgerufen werden muß welche auf die gesendeten Daten wartet. Die grundlegende Funktionalität, das Senden von Daten an einen Zielknoten stimmt bei beiden Routinen überein. Die MPI-Funktion ist noch um die Option erweitert, mittels des Kommunikators die Prozessoren zu Gruppen zusammenzufassen. Desweiteren bietet MPI ein verbessertes Typkonzept an, welches es ermöglicht, Daten nicht mehr auf reine Bytefolgen abbilden zu müssen, sondern direkt mit deren Typen zu arbeiten. Die Funktionalität der PARIX-Routine ist damit vollständig in der MPI-Funktion enthalten und kann

PARIX

`SendNode(ProcID, RequestID, Buffer, Size)`

ProcID	Identifikator des Zielknotens
RequestID	Message-Kennung
Buffer	Adresse des Sendedatenpuffers
Size	Anzahl der zu sendenden Bytes

MPI

`MPI_SEND(buf, count, datatype, dest, tag, comm)`

buf	Adresse des Sendedatenpuffers
count	Anzahl der zu sendenden Elemente
datatype	Bezeichner für den Datentyp eines Elementes
dest	ID des Zielknotens
tag	Message-Kennung
comm	Kommunikator (bezeichnet die Umgebung der Kommunikation)

Abbildung 2: Vergleich einer synchronen Kommunikationsfunktion

deshalb ohne Probleme bei einer Portierung durch diese ersetzt werden.

Das ein derartiger Austausch von PARIX- und MPI-Funktionen ohne größere Komplikationen zum Portieren geeignet ist, verdeutlicht die Gegenüberstellung der PARIX- und MPI-Implementation der in CUBECOM enthaltenen Funktionen `Send_Chan_0` und `Recv_Chan_0` in Abbildung 3. Diese Funktionen sind Teil der Architekturanpassung der Kommunikation und senden bzw. empfangen eine Anzahl (N) von Worten aus dem Puffer (X) über den HyperCube-Link (NrLink). Die PARIX-Version greift hierbei auf die im Vorfeld aufgebaute virtuelle Topologie zurück, während in dieser MPI-Realisierung diese Topologie noch nach den bekannten Regeln emuliert werden muß indem die Knotennummer des Kommunikationspartners (IWHO) berechnet wird.

Wie aus den Quelltextbeispielen ersichtlich ist, unterscheiden sich die PARIX- und MPI-Funktionen nur in den Details der Funktionsparameter. Die Umrechnung der Linknummer in die reale Knotennummer (`IHWO`) kann natürlich bei PARIX entfallen, da dieses von den Funktionen `Send` und `Recv` übernommen wird. Durch diese deutliche Übereinstimmung war es daher kaum ein Problem, die restlichen, für die Kommunikation nötigen, Funktionen auf der MPI-Basis zu implementieren.

4.2 Probleme bei der Portierung des Gesamtsystems

Zum Erzielen der besten Leistungsparameter und für ein optimales Ausnutzen der Eigenschaften des GC/PowerPlus wurden speziell für die PARIX-Version auf dem GC einige Besonderheiten, wie zum Beispiel zwei zusätzliche, auf die physischen Verbindungen der Knoten des Rechners abgestimmte Topologien, implementiert. Aus Zeitgründen konnten diese Besonderheiten der PARIX-Version nicht mit in die MPI-Variante übernommen werden. Da sich einige Algorithmen aber auf direktem Weg auf diese Topogien abstützen, mußte durch bedingte Compilierung eine Angleichung an den implementierten Leistungsumfang der MPI-Version erfolgen. Der dafür nötige Arbeitsaufwand für die Analyse der Funktion des FEM-Programmes und der jeweiligen Bibliotheken

PARIX

```
SUBROUTINE Send_Chan_0(N,X,NrLink)
...
lnr = NrLink-1
IF (N .GT. 0) THEN
  IER=Send(NrTop, lnr, X, 4*N)

ENDIF
RETURN
END

SUBROUTINE Recv_Chan_0(N,X,NrLink)
...

lnr = NrLink-1
IF (N .GT. 0) THEN
  IER=Recv(NrTop, lnr, X, 4*N)

ENDIF
RETURN
END
```

MPI

```
SUBROUTINE Send_Chan_0(N,X,NrLink)
...
IWHO = IEOR(ICH, ISHFT(1, NrLink - 1))
IF (N .GT. 0) THEN
  call MPI_SEND(X, N, MPI_INTEGER, IWHO,
     NrLink, MPI_COMM_WORLD, ierr)
ENDIF
RETURN
END

SUBROUTINE Recv_Chan_0(N,X,NrLink)
...
integer status(MPI_STATUS_SIZE)
IWHO = IEOR(ICH, ISHFT(1, NrLink - 1))
IF (N .GT. 0) THEN
  call MPI_RECV(X, N, MPI_INTEGER, IWHO,
     NrLink, MPI_COMM_WORLD, status, ierr)
ENDIF
RETURN
END
```

Abbildung 3: Quelltextvergleich zwischen PARIX und MPI

geht aber nicht auf Probleme mit dem Message-Passing-Interface zurück. Insbesondere kann in der Retrospektive gesagt werden, daß die aufgewandte Zeit für die Analyse und Anpassung der Bibliotheken in der gleichen Größenordnung, wie die Zeit für die Erweiterung der MPI-Version, also der Implementierung der speziellen Topologien, liegen dürfte.

5 Die Laufzeitanalyse der unterschiedlichen Versionen

5.1 Die Vorbereitung der Zeitanalyse

Das Ziel der Portierung war ein direkter Vergleich der Laufzeiteigenschaften der PARIX- und der MPI-Implementierung der architekturspezifischen Teile der Kommunikationsbibliothek. Da von vornherein mit einem Laufzeitnachteil der MPI-Version gerechnet wurde, war natürlich die Größe der für die zusätzlichen MPI-Rufe benötigten Zeit von besonderem Interesse.

Für einen aussagekräftigen Laufzeitvergleich waren aber noch ein paar Vorbedingungen zu schaffen. Durch den Verzicht auf die architekturoptimierten virtuellen Topologien bei der Implementierung der MPI-Version hat natürlich diese Variante ein deutliches Manko aufzuweisen. Um relevante Aussagen zu gewährleisten, wurde deshalb eine weitere Version erstellt, welche auf die PARIX-Umgebung des GC/PowerPlus zugreift, aber wie die MPI-Version nicht für dessen Architektur besonders optimiert wurde.

Es folgt eine Übersicht über die erstellten, bzw. verwendeten Implementationen der architekturabhängigen Kommunikationsfunktionen mit deren Leistungsmerkmalen:

ppc arbeitet mit der orginalen, für den GC/PowerPlus optimierten PARIX-Version der Bibliothek CUBECOM mit drei, zum Teil an die physische Linkstruktur des Rechners angepaßten, virtuellen Topologien (HyperCube, Ring und KettAkk)

ppcblank basiert auf einer abgerüsteten PARIX-Variante der Kommunikationsbibliothek CUBECOM ohne spezielle Optimierungen für den GC/PowerPlus und hat damit den gleichen Aufbau und die gleichen Voraussetzungen wie die MPI-Version **ppcmpi**

ppcmpi nutzt für die Kommunikation die grundlegenden, synchronen Funktionen der MPI-Bibliothek; die für die Arbeit des Programmes nötige HyperCube-Topologie wird durch Emulation bereitgestellt; spezielle Anpassungen an die Kommunikationsstruktur des Rechners werden nicht vorgenommen

Aus den nun möglichen Laufzeitvergleichen können damit zwei Aussagen gebildet werden:

- Der Geschwindigkeitsgewinn durch die Anpassung und Optimierung eines PARIX-Programmes an die spezielle Linkstruktur der zugrundeliegenden Architektur, als Differenz der Laufzeiten der Versionen **ppc** und **ppcblank**.
- Der Laufzeitoverhead, den die abstrahierende "Zwischenschicht" MPI mit sich bringt und mit dem die Vorteile der Architekturunabhängigkeit erkauft werden müssen. Ablesbar ist dies aus der Laufzeitdifferenz der Varianten **ppcmpi** und **ppcblank**.

5.2 Die Zeitanalyse

Gemessen wurden die folgenden Laufzeiten des FEM-Programmes mit der internen Zeitmessung dieser Applikation, die sowohl die Zeit für die Kommunikation, als auch für die Berechnung mit einbezieht. Ermittelt wurden der Zeitverbrauch der drei vorgestellten Versionen bei zunehmender Verfeinerung des Problems. Ein Maß für die steigende Komplexität und den wachsenden Aufwand ist der Parameter *Level*. Ausgehend von einer Grundauflösung des Problems bei Level 1, wird mit steigender Auflösung die

Kantenlänge der finiten Elemente halbiert. Damit vervierfacht sich bei jedem Level-Schritt die Anzahl der finiten Elemente. Entsprechend dazu steigt der Berechnungs-, Speicher- und Kommunikationsaufwand des Problems.

Gearbeitet wurde auf dem GC/PowerPlus-128 mit einem Cluster aus acht Prozesoren. Berechnet wurde hierbei das Problem "q". Die PARIX-Versionen wurden direkt auf den Knoten durch `px run ...` gestartet. Die MPI-Variante benötigt für den Start das Programm `mpirun`, welches den Code der Applikation läd und dessen Ausführung auf den einzelnen Knoten einleitet.

Hier folgt eine Aufstellung der ermittelten Zeiten (in Sekunden) in tabellarischer und graphischer Form (Abbildung 4) in Abhängigkeit von dem Auflösunglevel des Problems:

Level	**1**	**2**	**3**	**4**	**5**	**6**
ppc	0,27	0,30	0,44	0,79	2,19	7,31
ppcblank	0,24	0,29	0,40	1,90	3,22	8,60
ppcmpi	0,28	0,30	0,45	1,83	3,42	8,75

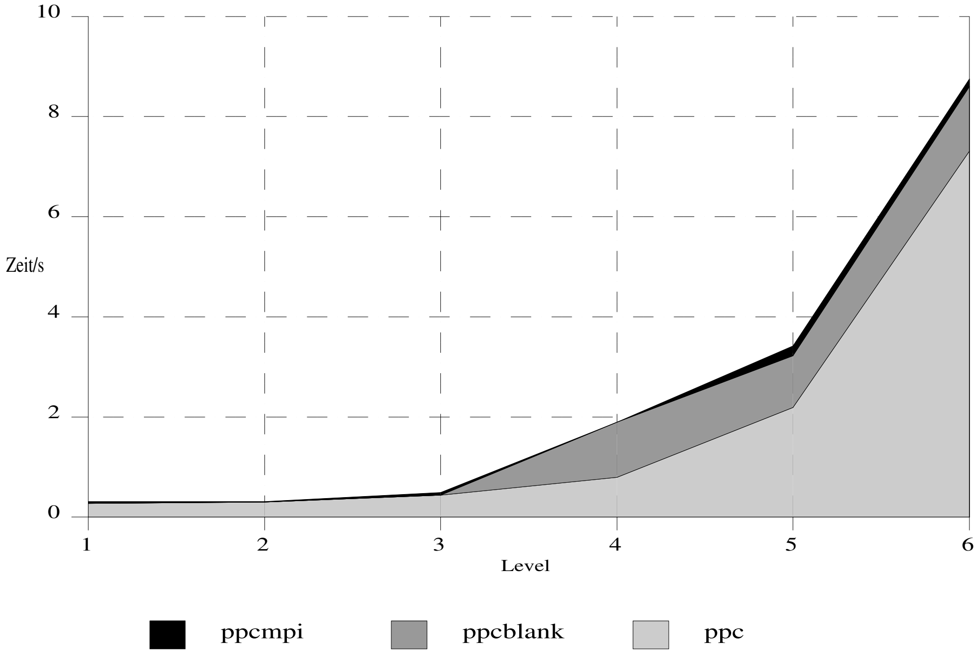


Abbildung 4: Die Laufzeit bei den einzelnen Versionen der Kommunikationsbibliothek

Die dargestellten Zeiten sind die Werte einer Meßreihe, die unter den beschriebenen Bedingungen aufgenommen wurde. Sie steht exemplarisch für alle durchgeführten Zeitmessungen, die auch mit anderen Clustergrößen und Problemen durchgeführt wurden. Größere Streuungen der Bearbeitungszeiten konnten bei gleichbleibenden Bedingungen (Clustergröße, Problem) nicht festgestellt werden. Auch bei veränderten Einstellungen behielten die Berechnungszeiten ihre Relationen bei, so daß die angegebene Meßreihe als representativ für das Zeitverhalten der Applikation angesehen werden kann.

5.3 Resultat der Zeitanalyse

Wie besonders aus der Grafik zu entnehmen ist, wird durch die Architekturoptimierung in der PARIX-Implementation (**ppc**) eine deutliche Laufzeitverbesserung gegenüber einer konventionellen PARIX-Applikation erreicht. Der Laufzeitoverhead der MPI-Bibliothek ist dagegen gering. Dies erklärt sich aus der Gleichheit der Funktionalität der benutzten PARIX- oder MPI-Routinen. Die Prozeduren der MPI-Bibliothek mußten lediglich die Kommunikationsrufe und deren Parameter an die entsprechenden PARIX-Funktionen weiterleiten.

6 Die möglichen Verbesserungen der MPI-Version

Wie aus dem eben genannten Resultaten hervorgeht, ist als entscheidendes Kriterium für die Laufzeit die Anpassung an die Architektur und Kommunikationsstruktur zu sehen. Die Performance der MPI-Implementation könnte also noch dadurch gesteigert werden, daß analog zu der orginalen PARIX-Variante, spezielle und angepaßte virtuelle Topologien implementiert werden. Diese würden die Struktur des GC/PowerPlus ausnutzen und damit die Kommunikationswege und -entfernungen optimieren und die Kommunikation gleichmäßiger im Netz der Links verteilen. Eine solche, optimierte Variante käme mit hoher Wahrscheinlichkeit an die Leistungsfähigkeit der PARIX-Variante heran, bringt aber auch gleichzeitig den Nachteil der Architekturabhängigkeit mit sich.

Die bisherige Implementation der MPI-Variante ist in Anlehnung an Abbildung 1 als eigenständiger Teil der Bibliothek CUBECOM zu sehen. Hierbei hat er den Vorteil, daß er nicht nur als Bindeglied der allgemeinen Funktionen zum GC/PowerPlus, sondern auch zu einer Vielzahl von anderen Parallelrechnern oder Workstation-Clustern fungieren kann. Trotzdem werden viele besondere Möglichkeiten des Message-Passing-Interface-Standards nicht verwendet. So werden sämtliche komplexeren Kommunikationsfunktionen, wie zum Beispiel Funktionen für globale Kommunikation oder auch für den simultanen Austausch von Werten zwischen zwei Knoten von der Bibliothek CUBECOM realisiert. All diese Aufgaben könnte allerdings auch die MPI-Bibliothek übernehmen und auf direkterem Wege auf die Hardware umsetzen. Die größere Leistungsfähigkeit von MPI würde durch eine Umstrukturierung der Kommunikationsbibliothek nach folgendem Schema (Abbildung 5) erschlossen werden.

Hierbei übernimmt MPI weite Teile der komplexen Kommunikationsfunktionen *und* die Anpassung an die jeweilige Parallelrechnerarchitektur. Vom Entwickler der Applikationen oder der Bibliotheken ist kein architekturspezifischer Code mehr zu schreiben, sondern die Bibliothek CUBECOM lediglich mit der entsprechenden MPI-Implementation zu linken.

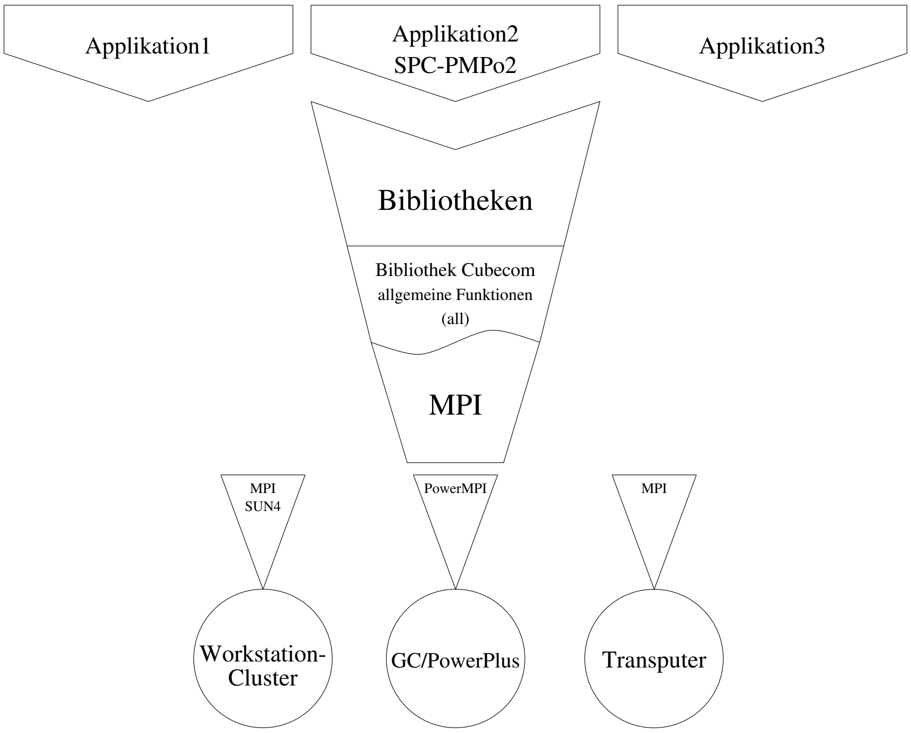


Abbildung 5: Die Applikation SPC-PMPo2 mit MPI als komplexen, funktionalen Bestandteil

Eine solche, drastische Umstrukturierung ist natürlich bei einer seit langem bestehenden Bibliothek, wie in diesem Fall der Bibliothek CUBECOM, kaum zu empfehlen. Vielmehr könnte bei einem erneuten Design von ähnlichen Applikationen oder Bibliotheken der umfassende Einsatz von MPI in der vorgeschlagenen Form in Betracht gezogen werden. Damit ließen sich dann architekturunabhängige und portable Applikationen und Bibliotheken erstellen, welche auf einer umfassenden Bibilothek für Prozeßkommunikation aufsetzen könnten. Als Nachteil wäre dann allerdings zu sehen, daß nur parallele Architekturen verwendet werden können, für die eine MPI-Implementation existiert.

Literatur

[1] G. Haase, T. Hommel, A. Meyer, M. Pester. Bibliotheken zur Entwicklung paralleler Algorithmen. Preprint SPC 95_20, TU-Chemnitz-Zwickau, Juni 1995

[2] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. May 5, 1994

[3] Klaus-Jürgen Bathe. Finite-Elemente-Methode. Springer-Verlag 1990

Application Oriented Monitoring

Matthias Ohlenroth

moh@informatik.tu-chemnitz.de
http://noah.informatik.tu-chemnitz.de/members/ohlenroth/ohlenroth.html

Abstract

Optimizing parallel applications is a complex task. Special tools are required to record and analyze their behaviour. This paper introduces facilities offered by monitoring environments based on source code instrumentation. Typical components of monitoring environments and their use are explained. Time based and event based trace generation are introduced. A sample monitoring session illustrates possibilities and limitations of available tools.

1 Introduction

Monitoring becomes more and more important, especially for parallel computing environments. Efficient parallel programming is a difficult task. The application programmer has to select the underlying services carefully for optimal speedup. This process is influenced by interactions and dependencies between all software and hardware layers. Potential performance problems must be detected while testing the application. Monitoring tools provide mechanisms to accelerate this task. Usually the programmer inserts probe functions into the source code. This requires several loops of the form:

add probe function – execute application – analyze data.

Portability becomes more and more important for parallel applications. But optimal performance requires precise adaption to the underlying parallel computer. Software abstractions are used to tone down this contrast. One sample is the message passing library MPI. The interface definition permits efficient implementation for different parallel computers. Interactions between application, message passing library and hardware have great impact on the performance of the application. One abstraction layer may offer different ways to solve a particular implementation problem. The choice made by the programmer influences the performace of the application.

Performance tuning of sequential applications requires analyzation of runtime data and interactions with lower layers of the architecture. Profiler help to find often called or long running functions. It is likely that optimizing these functions reduces the execution time considerably. Interactions between application and operating system may be analyzed. Special tools allow one to trace system calls.

Tuning parallel applications is more complex. A parallel program consists of communicating processes or threads. The efficiency depends on the algorithm, the implementation, the system software and the hardware. Performance bottlenecks result from insufficient parallel workload, communication and synchronization problems. Monitoring environments trace program behaviour, especially synchronization, communication, blocking times and application specific events. Inefficient program regions can be identified from application traces. Changes will be made to ensure high locality of computation, equal workload distribution and efficient communication.

2 The Components of a Monitoring Environment

A monitoring environment consists of an instrumentation tool[11], a trace library and a visualization tool. These components correspond to the three tasks of a monitoring session:

- instrument the application,

[11]Some systems do not offer an instrumentation tool. Other solutions integrate compiler and instrumentation tool.

- execute the application,
- analyze the trace data.

The instrumented application generates trace data during execution. These data can be examined on-line or off-line. Off-line tools are independent from the program execution. Stored trace data can be analyzed as time permits. On-line tools offer performance pictures of running applications. Special connections transfer trace data between application and presentation tool. Backward connections permit steering of the application. The user may influence the data partitioning of the application for instance. Figure 1 illustrates both approaches.

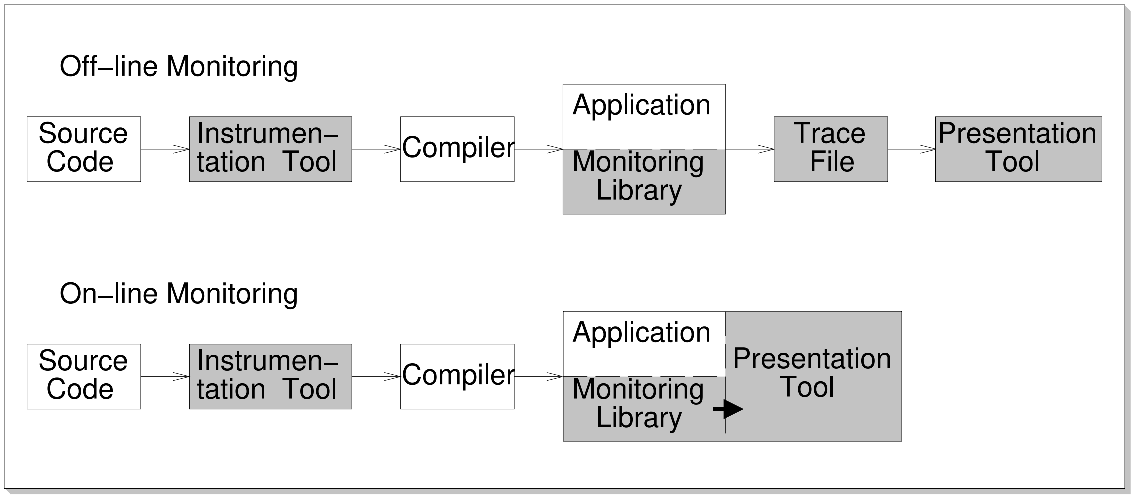


Figure 1: Monitoring Environments.

2.1 The Instrumentation

The instrumentation step adds calls to probe functions to the application code. This can be done automatically by the compiler. Automatic instrumentation is limited to program structures which are visible to the compiler. Other tools modify the program source code directly. After parsing the file instrumentation can be done interactively by the programmer. Tools permit the instrumentation of function calls and loops. The following example demonstrates the instrumentation of a procedure call. Probe functions are inserted before and after the function `test_func()`.

```
   ...
#define EventID 1000;

probe_function(EventID);
test_function();
probe_function(EventID);
```

...

The first probe function starts an interval timer inside the trace library. The second call stops this timer and generates a trace record containing the following members:

- node number,
- event identifier,
- value of the event local counter,
- local time,
- time duration.

The node number identifies the thread or process of the parallel application. Each event is represented by its unique event identifier. The number of occurrences of an event is recorded by the library. The local time and the duration of the event are stored.

Specific instrumentation may be added manually. To this class belong application dependent events and time duration events without relation to program structure. Timer might be started in one function and stopped inside a different function.

The instrumented source code will be saved to a new file. This file must be compiled and linked with the application instead of the original one. Direct instrumentation of source code does not require modifications of the compilation system.

2.2 The Trace Library

The trace library contains the probe functions and other code required to manage trace data streams. The library collects trace data, saves them to a trace file or forwards them to on-line visualization tools. The management of trace data requires resources of the application. These are processing time and memory. Event records will be buffered by the library. Some libraries use their own thread of execution.

Trace libraries require different levels of initialization. Some libraries are self-initializing. Default parameters like trace file name or socket number can be overwritten by the application. Special care is needed during application shutdown. The trace data buffer must be flushed before the main function of the application returns. This requires insertion of an explicit call to the trace library. Instrumentation tools generate appropriate wrapper functions for the main function. The wrapper function performs the required actions.

2.3 The Trace File

The trace file contains records for all events processed by the monitoring library. Depending on the monitoring environment one application trace consists of one or several different trace files. Every node may generate its own file. These files have to be merged into one file before analyzing the trace. Merging tools order records by their time stamp. Users may specify other rules.

Although some of the file format specifications are published most of the monitoring environments use their own file format. This prevents one from using a visualization tool with trace data generated by a different environment. Reasons for new file formats are limitations of other specifications. One known format is the self defining data format (SDDF) used by the Pablo [RAM+92] performance analysis environment. The file contains information about the used records followed by the data records. ASCII and binary representations of the same specification are defined. A set of tools perform transformations between both representations. Binary files with different byte ordering can be transformed to the native byte ordering of the host computer. Trace data generated on a parallel machine may be analyzed on a host system with different architecture.

2.4 The Visualization Tool

The presentation tool provides a framework for trace data processing and visualization. Some tools can be configured by the user. Programs for on-line and off-line visualization are available.

A visualization session consists of module instances and connections between them. Configurable tools offer a set of modules. The user selects modules from the groups file I/O, filtering, data processing and visualization. Modules and connections form an acyclic graph. Connections transfer data records between modules. Different records may be passed along one connection. Modules have to be configured. This includes input ports, output ports and internal parameters. Sessions may be saved and reused later.

Figuring out performance problems is a difficult process. Several modification of the session configuration are required to clarify the problem. Hence the following procedure has to be performed several times:

$$\begin{array}{c}\text{create or modify session configuration}\\ \downarrow\\ \text{process trace data and generate graphical presentation}\\ \downarrow\\ \text{interpret data}\end{array}$$

The user has to analyze visualized data and figure out the performance problem.

Some tools offer different levels of support. New users may benefit from predefined session configurations. More experienced users want to create their own sessions. Expert users may reach limits of the modules provided by the program. They wish to create new modules and extend the visualization tool. The Pablo [RAM+92] tool offers a class library. The library separates module implementation, connection management and low-level data processing. New modules can be build on top of the existing class library.

On-line visualization tools contain components for on-line program steering. Adjustments made by the user are propagated back to the monitoring library. Special variables transfer information back to the application. The application defines these variables and registers them with the monitoring library.

3 Trace Data Generation

This section introduces different approaches to generate trace data. Each peace of a trace file, called record, provides information about one particular event of the application. This information is limited to one thread or processing node and does not reflect global states. A record represents an application event. All records together provide information about the behaviour of the application.

Architecture support for monitoring differs. Some architectures provide hardware support for trace data generation. Depending on the level of hardware support hardware monitoring or hybrid monitoring may be used. Hybrid monitoring combines hardware and software facilities to produce accurate trace information. Software monitoring is used on systems without hardware support. Two common monitoring techniques are

- time-driven monitoring and
- event-driven monitoring.

Both approaches provide different levels of details, program perturbation and implementation costs.

3.1 Time-driven Monitoring

Time-driven monitoring requires operating system support. The program counter is sampled periodically by the operating system. Sampling is local to each thread or processing node. This technique exposes the most expensive parts of the application. It is likely that most samples belong to often called or long-running functions. The processing of program counter samples requires compiler support. Samples must be assigned to source code lines or functions. The sampling frequency used by the operating system has great impact on the accuracy.

3.2 Event-driven Monitoring

A chain of events represents the program behaviour. The level of precision is adjustable. Event-driven monitoring requires higher implementation cost. The programmer is concerned with the instrumentation process directly. Special instrumentation requires coding by hand. This method offers more control to the programmer. He choses relevant events and controls directly the level of precision. Events are generated by calls to the monitoring library. Each call to a probe function requires processing time and delays program execution. The library buffers event records and writes them to the trace file. One major disadvantage is the program perturbation. Libraries like Pablo provide mechanisms to enable or disable probe functions and ajust the event type on-line.

The following event types are distinguished:

- Trace event,
- Count event,
- Time event.

Trace events represent single program events, i.e. function calls. Application specific events belong to this class. Each event record contains at least the local time, the event identifier and the node number. The application may append further data. A sample code fragment is given in figure 2. All calls to the function `doit()` are traced. Count events register the number of occurences of particular events. The monitoring library appends the event number automatically. The use of count events is demonstrated by function `start()`. Time events are used to measure the time duration between two application-specific instrumentation points. The monitoring library manages one interval timer for each time event. The timer is started and stopped by two consecutive events. Time events are used to measure the time between calls to the functions `start()` and `stop()` of example 2. The same event would be generated if both functions are called in reverse order.

```
start () {
    count_event ( COUNT_EVENT_1 );
    time_event ( TIME_EVENT_1 );
    ...
}

stop () {
    time_event ( TIME_EVENT_1 );
    ...
}
```

```
doit (int n) {
    int i;

    trace_event ( TRACE_EVENT_1 );

    for ( i=0; i<n; i++ ) {
        start ();
        stop ();
    }
}
```

Figure 2: Using probe functions.

4 A Sample Problem

This section describes one particular monitoring session. Opportunities and limitations offered by monitoring tools are illustrated. A FEM application written by the departement of mathematical science was tested on a four processor Compaq ProLiant 4000 system. The program was written for distributed memory multiprocessor systems. It was ported to Linux using the message passing library TCGMSG. The algorithm ensures equal load distribution over all processors. Hence execution times of identical procedures running on different processors should be nearly identical. Blocking times during global synchronization are expected to be small. This behaviour was demonstrated on different parallel computers. Tests have shown linear speedup on these systems for sufficient large problem size.

Experiments on the Compaq system pointed out performance problems. The speedup was considerably lower than expected on configurations with more than two active processors. Other experiments on this architecture demonstrated hardware limitations. The memory bus bandwith is sufficient for two processors. Memory access attempts by additional processors are delayed. Routines with low processing time for each memory reference suffer from this problem. Performance problems belonging to this class are expected to occur inside the module *prloes.f*. The function `prloes()` contains local computations and global synchronization steps. The bandwith problem leads to unpredictable increase of the execution time for local computations. Faster processors are blocked during synchronization steps.

This supposition should be confirmed using the Pablo performance analysis environment. At first the Pablo monitoring library was ported to Linux. The Fortran sourcecode *prloes.f* was converted to c using the program `f2c`. The instrumentation code was inserted manually[12]. A first step should prove our supposition. Instrumentation was limited to communication steps and code blocks between them. Figure 3 shows the function `prloes()` after instrumenting the first block.

[12]This was done for demonstration purpose only. Automatically inserted code would be more difficult to read.

```
int prloes_(integer *nku, integer *ncu, integer *n,
    doublereal *a, integer *la, doublereal *c, integer *lc, doublereal *
    cc, integer *lcc, integer *kette, integer *iglob, doublereal *w,
    doublereal *r, doublereal *v)
{

...

startTimeEvent (0);

    vd0mul_(n, &w[1], &c__1, &r[1], &c__1, &c[1], &c__1);
    hstmul_(n, &w[1], &lc[1]);
    vdmult_(n, &w[1], &c__1, &w[1], &c__1, &c[1], &c__1);
    vdcopy_(&problem_1.ncrossg, &v[1], &c__1, &c_b9, &c__0);
    i__1 = problem_1.ncrossl;
    for (i = 1; i <= i__1; ++i) {
        ito = iglob[i];
        v[ito] = w[i];
    }

endTimeEvent (0);

startTimeEvent (1);

    treeup_dod__(&problem_1.ncrossg, &v[1], &v[1],
        &v[problem_1.ncrossg + 1],
        (U_fp)vdplus_);

endTimeEvent (1);

....

}
```

Figure 3: The instrumented function `prloes()`.

Trace events are generated by the probe functions `startTimeEvent()` and `endTimeEvent()`. The function `treeup_dod__()` performs a global reduction operation on a hypercube communication topology. Figure 4 presents execution time charts obtained by the Pablo visualization tool. The visualization was done on a workstation running SunOS. Four processors were used for this test. The picture does not distinguish between events generated by different processors. The upper graph draws the execution times from the computation block. Communication times are presented by the second graph. The execution time differes considerably on both graphs. Further investigation is needed to clarify the problem.

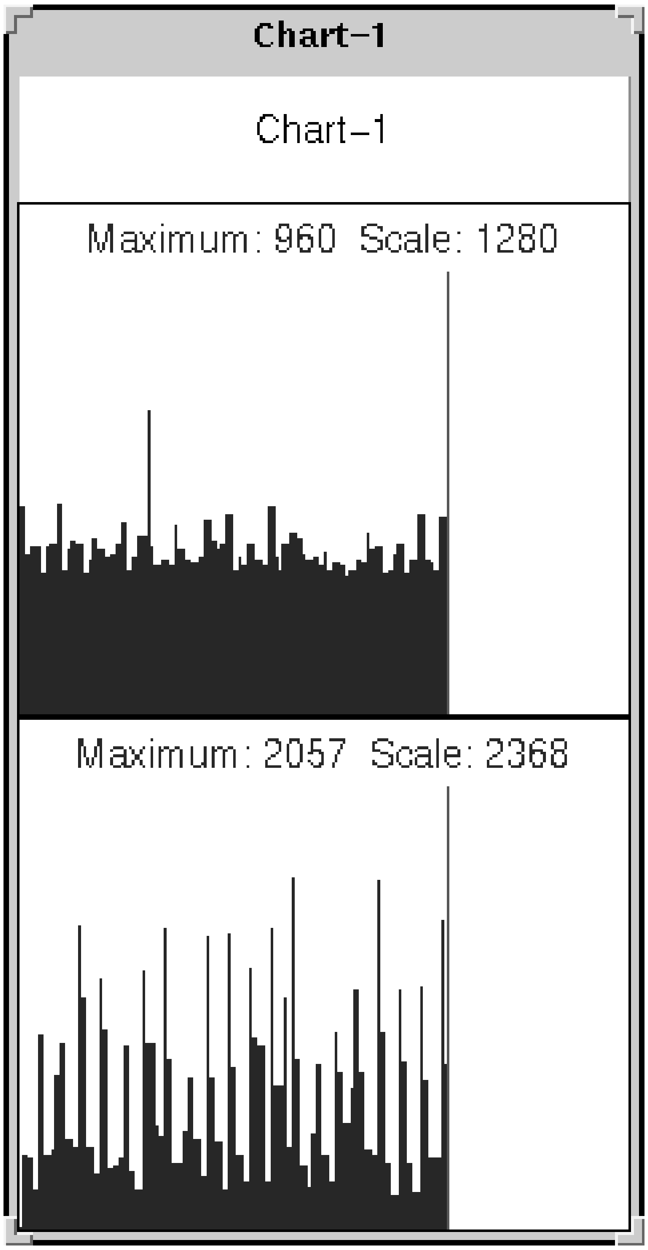


Figure 4: Execution and communication time.

The traced code block contains four function calls. During a second step these functions are instrumented separately. Figure 5 lists the instrumented code. Procedure entry and exit events are generated by the function `PabloTraceProc()`. A unique identifier is assigned to each event. Figure 6 presents the obtained trace data. Each graph shows the execution time of one function called by `prloes()`. The first graph belongs to function `vd0mul_()`. Other functions are ordered as they occure in the source code. The execution time of the functions `vd0mul_()` and `treeup_dod__()` differes considerably. Both functions contribute to the performance problem. We evaluate the function `vd0mul_()` in detail. Figure 7 lists the source code. Memory operations are executed by all instances of this function concurrently. Hence concurrency is limited by the memory access bandwith.

```
int prloes_(integer *nku, integer *ncu, integer *n,
    doublereal *a, integer *la, doublereal *c, integer *lc, doublereal *
    cc, integer *lcc, integer *kette, integer *iglob, doublereal *w,
    doublereal *r, doublereal *v)
{
...

PabloTraceProc (0,0,0);
    vd0mul_(n, &w[1], &c__1, &r[1], &c__1, &c[1], &c__1);
PabloTraceProc (1,0,0);

PabloTraceProc (2,0,0);
    hstmul_(n, &w[1], &lc[1]);
PabloTraceProc (3,0,0);

PabloTraceProc (4,0,0);
    vdmult_(n, &w[1], &c__1, &w[1], &c__1, &c[1], &c__1);
PabloTraceProc (5,0,0);

PabloTraceProc (6,0,0);
    vdcopy_(&problem_1.ncrossg, &v[1], &c__1, &c_b9, &c__0);
PabloTraceProc (7,0,0);

    i__1 = problem_1.ncrossl;
    for (i = 1; i <= i__1; ++i) {
        ito = iglob[i];
        v[ito] = w[i];
    }

PabloTraceProc (8,0,0);
    treeup_dod__(&problem_1.ncrossg, &v[1], &v[1],
        &v[problem_1.ncrossg + 1],
        (U_fp)vdplus_);
PabloTraceProc (9,0,0);

....
}
```

Figure 5: The instrumented function `prloes()`.

```
    SUBROUTINE VD0mul(N,X,ix,Y,iy,Z,iz)
C   IMPLICIT AUTOMATIC (A-Z)
```

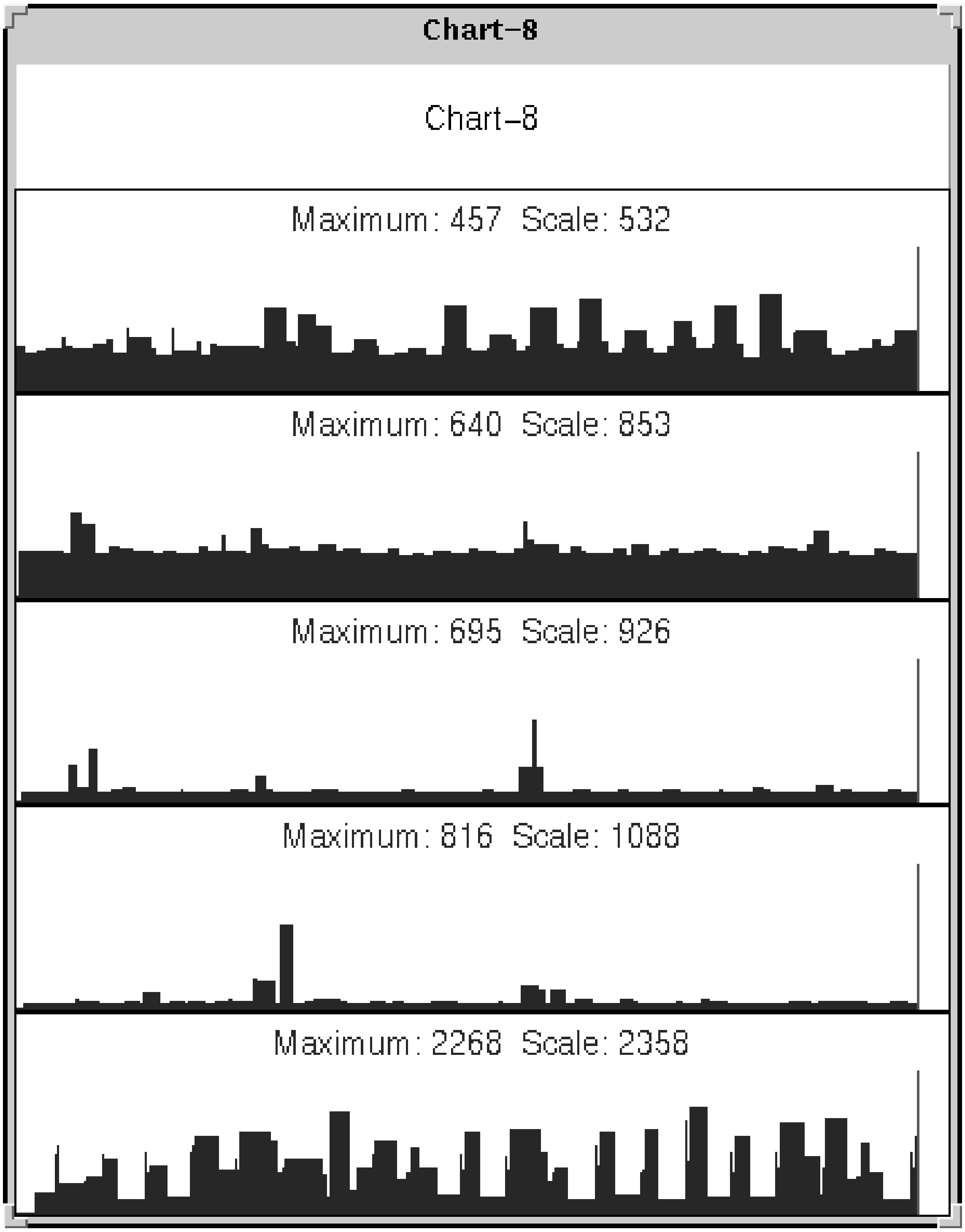


Figure 6: Visualization with the Pablo tool.

```
Double Precision X,Y,Z,S
DIMENSION X(1),Y(1),Z(1)
jx=1
jy=1
```

```
    jz=1
    DO 1 I=1,N
    S = Y(jy)
    IF (Z(jz) .EQ. 0D0) S=0D0
    X(jx) = S
    jx = jx + ix
    jy = jy + iy
1   jz = jz + iz
    RETURN
    END
```

Figure 7: The function `VD0mul_()`.

Althrough monitoring helps to identify the performanc problem it does not provide a solution. The presented problem cannot be solved due to hardware limitations. A second dependency was not mentioned until now. The past tests were done using small FEM problems. Each processor has to perform computations on small memory regions. Only some cache lines must be loaded by each processor. Larger problems require a lot of load operations. In this case the average load delay would be identical on all processors. Hence the problem would be invisible to monitoring atempts.

The sample described above illustrates some issues which may limit the success of monitoring sessions:

- The visability of problems may be limited.
- The visability of problems depends on input data.
- Execution time variations are hidden after several local iterations.

5 Conclusions

Monitoring environments provide tools for performance analysis of parallel applications. A source code based method which does not require compiler support was introduced. User controlled instrumentation offers high flexibility. Monitoring libraries generate and save event records. Instrumentation should be done carefully. Monitoring activities require application resources and perturb the application. Hence application behaviour may change.

Capabilities and limitation of monitoring tools were demonstrated using a sample problem. A small set of probe functions is sufficient for most monitoring requirements. Expressive and configurable graphical tools ease interpretation of trace data. This requires flexible filtering and processing capabilities.

Future work adresses the topics visualization, event processing and operating system monitoring. Visualization and processing facilities offered by different tools will be analyzed in detail. Knowledge how current tools satisfies the need of programmers is needed to prepare better tools. While preparing the demonstration some experiments have shown transient effects. These effects are attributed to operating system activities. Identifying them requires monitoring facilities inside the operating system. This includes interrupt processing, scheduling and other process state switches. We want to integrate monitoring facilities into Linux/SMP.

References

[GEKS94] W. Gu, G. Eisenhauer, E. Kraemer, and K. Schwan, *Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs*, Tech. report, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 1994, URL: ftp://ftp.cc.gatech.edu/pub/tech-reports/1994/GIT-CC-94-21.ps.Z.

[GVS94] W. Gu, J. Vetter, and K. Schwan, *An Annotated Bibliography of Interactive Program Steering*, Tech. report, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 1994, URL: ftp://ftp.cc.gatech.edu/pub/tech-reports/1994/GIT-CC-94-15.ps.Z.

[Noe94] R. J. Noe, *Pablo Instrumentation Environment Reference Manual*, Tech. report, Department of Computer Science, University of Illinois, Illinois, 1994.

[RAM+92] D. A. Reed, R. A. Aydt, T. M. Madhyastha, R. J. Noe, K. A. Shields, and B. W. Schwartz, *An Overview of the Pablo Performance Analysis Environment*, Tech. report, Department of Computer Science, University of Illinois, Illinois, 1992.

An Implementation of MPI – The Shared Memory Device

Oliver Langer

ola@informatik.tu-chemnitz.de
http://noah.informatik.tu-chemnitz.de/members/langer/langer.html

1 Introduction

This paper discusses the structure und functionality of the shared memory device under the MPICH implementation of MPI. The description is done with regard to other projects, concerning modifications of the shared memory device.
The author's intention is neither to describe all details, nor simply mention the basic behaviour of the device. He wants to give an explanation of important structures and mechanisms, which are frequently used in the device, so that programmers get familiar with the source code. By pointing out structure- and function-names, considering filenames and mentioning other details related to the source code, the author also wants to give a little reference with this paper[13].
Another goal is to figure out those characteristics of the device, which for a multithreaded version[14] may be of interest.

Before continuing, several document conventions:

- Names, taken from source code, are `emphasized` .
- Sometimes, the location (concerning the file) of a piece of source code is given. The underlying file structure is based on MPICH's version 1.0.12.
- Several identifiers are followed by other identifiers, which are placed within brackets. Those in the brackets are to be found in the source code and simply refer to the identifiers, mentioned not in brackets[15]

[13] As a result, several sections (like "MPI_Send") grew a little up...
[14] realized through a modification of the current shared memory device
[15] e.g. macros are such candidates

2 An Implementation of MPI

MPICH is an implementation of MPI, developed at Argonne National Laboratory. It is freely available und supports a number of systems including shared memory systems. To support a number of different systems (distributed memory systems, shared memory systems, workstation cluster, etc.), the MPICH implementation is divided into a higher-level and a lower-level part (see also figure 1[16]). The higher-level part defines the

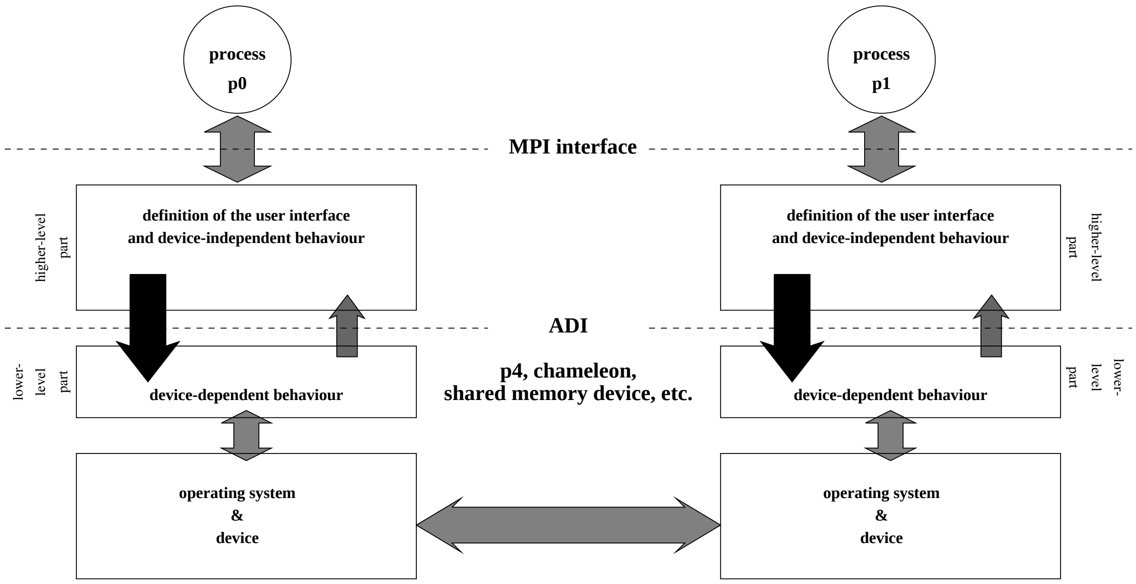


Figure 1: Structure of MPI

user-level functions, like MPI_Send and is responsible for the general control flow of the MPI-functions. It does never refer to any hardware- or system-specific functions.
The core message passing routines, it has to use, are mapped through the *Abstract Device Interface (ADI)* (see glossary) , which is device- / system-dependent. Its general task is to send data to/receive data from another process by using TCP, shared memory or other communication packages, like P4.
As a result of this structure, implementors have to follow the definition of the ADI, if they want to create a new lower-level part or want to modify an existing one. So the following sections mainly refer to the ADI and its implementation in the shared memory device.
For information about the general behaviour of a MPICH device, based on the ADI, please refer to [3] and to [2] for the belonging reference.

[16]Note: From this figure you may conclude, that the MPI-function are to be found outside the process' adress space. This is **not** the case!

3 The Init Process

The initial process of an MPI program is responsible for evaluating commandline parameters, allocating and organizing process-local memory buffers and shared memory regions, building **MPI_COMM_WORLD** (see glossary) and **MPI_COMM_SELF** (see glossary) , creating datatype information, installing errorhandling routines and the predefined reduction operations.
Each MPI application has to release the initial process by the `MPI_Init()` call, to be found in `MPI_Init()` [def. in *src/env/init*] . First, `MPI_Init()` initializes the lower-level part and after that, the higher-level part of MPI.

3.1 Initializing the Lower-level Part

At the beginning of each MPI application the programm is started directly by its name or indirectly via `mpirun` . Thus only one process (`root process` [17]) is active. Its task is to read the commandline parameters (in order to find the number of desired processes), allocate the shared memory region, start the other processes and give each of it its unique identificator (`id`). This is done in `MPID_SHMEM_init()` [def. in *mpid/ch_shmem/shmempriv.c*] .
The root process gathers the number of desired processes from the "-np" option, given by the commandline parameter and extracts this information from it, because it is necessary only to the root process.
The shared memory device does have its own routines (allocation/deallocation) for managing shared memory regions during runtime. That means that in the initial procedure, a shared memory region is allocated via system calls and subsequent allocation/deallocation of shared memory refers to functions of the device [def. in *mpid/ch_shmem/p2p.c*]. The memory managing functions are derived from ANSI K&R and use a linked list to organize this region.
In order to get the shared memory region und initialize the memory management routines `p2p_init()` [def. in *mpid/ch_shmem/p2p.c*] is called. An allocation of "nr. of processes * 128 + sizeof(*MPID_SHMEM_globmem*) [18]" Bytes is done specific to the underlying system. The current implementation supports allocation through `mmap()` function, `system V IPC` and through IRIX specific functions.
After calling p2p_init, memory for a *MPID_SHMEM_globmem* structure is assigned to *MPID_shmem* using `p2p_shmalloc()` call. Its function is described in section "Usage of Shared Memory" starting on page 102.
Next, all processes are started via `p2p_setpgrp()` [def. in *mpid/ch_shmem/p2p.c*] , which sets up the process group id, followed by `p2p_create_procs()` [def. in *mpid/ch_shmem/p2p.c*] , which starts the other processes using `fork` .
Now each process gets its id by reading and increasing *MPID_shmem->globid++* and

[17] in this case, root means first. After creating the other processes, their all treated alike
[18] structure is descriped next

initializing of the lower-level part has finished[19].

3.2 Initializing the Higher-level Part

Initialization of the higher-level part is to be found in `MPIR_Init()` [def. in *src/env/initutil.c*] . Its task is to setup the MPI-enviroment, including topologies, **MPI_COMM_WORLD** and
MPI_COMM_SELF, creating datatype-information[20], installing the predefined reduce operations and errorhandling routines, creating standard Attributes and reading and evaluating parameters from the commandline. The description of the related details will be left over, because mainly, they are of interest for understanding the higher-level part.

4 Usage of Shared Memory

The allocated piece of shared memory is structured through the following type `MPID_SHMEM_globmem()` [def. in *mpid/ch_shmem/channel.h*] :

```
typedef struct {
    p2p_lock_t availlock[MPID_MAX_PROCS];
    p2p_lock_t incominglock[MPID_MAX_PROCS];
    p2p_lock_t globlock;
    MPID_SHMEM_Queue    incoming[MPID_MAX_PROCS];
    MPID_SHMEM_Stack    avail[MPID_MAX_PROCS];
    VOLATILE MPID_PKT_T pool[MPID_SHMEM_MAX_PKTS];
    VOLATILE int        globid;
    MPID_SHMEM_Barrier_t barrier;
    } MPID_SHMEM_globmem;
```

The device regards the shared memory region as a pool, where processes exchange messages. The size of this pool does not grow up during runtime and as a result, large messages may be split into several pieces and then explicitly send. This (put-/get-) mechanism will be mentioned later.
Messages are delivered through so called *packets* (see glossary) , which contain the sending mode[21], a context identifier, the local rank of the sending process, the pointer to the buffer, holding the message data, the length of the message data and other information, depending on the sending/receiving mode.
So to send data, a process does the following:

1. create and setup a packet in shared memory

[19]The following calls `MPID_SHMEM_Init_recv_code()` [def. in *mpid/ch_shmem/shmemrecv.c*] and `MPID_SHMEM_Init_send_code()` [def. in *mpid/ch_shmem/shmemsend.c*] don't have any effect
[20]for the predefined MPI datatypes
[21]for example "short" or "long"

2. copy the message data "into the packet"

3. insert this packet into the `incoming queue` of the receiving process

With regard to the shared memory, it has to store the packets and the incoming queue. This is realized by the components *incoming[i]*, pointing to the first element of the incoming queue of process i and by *pool*, keeping space for packets.
In order to avoid loss of efficiency caused by frequently allocating/freeing packets, each process is assigned a predefined number of packets. These available packets are organized through a stack and process' i top pointer to its stack is stored in the component *avail[i]*. But by delivering data through inserting one of its packets into receiver's incoming queue, the sending process loses one of its available packets. As a result, allocation of new packets would be necessary causing loss of efficiency. To solve the problem, each receiving process hands over the received packet to its owner. This is done by `MPID_SHMEM_FreeSetup()` [def. in *mpid/ch_shmem/shmempriv.c*] (MPID_PKT_RECV_FREE), when a receiving(!) process frees the packet!
`MPID_SHMEM_FreeSetup()` simply inserts the packet into the "*avail* stack" of the corresponding process. To guarantee atomic modifications upon stack i of process i, lock mechanisms[22] are used refering to the locking variable stored in component *availlock[i]*. *incominglock[i]* is defined in analogy form, used to manipulate *incoming[i]*.
The other components not mentioned up to now, are used for global locking (*globlock*), for realizing barriers (*barrier*) and *globid* is used for determining the process' identifier in the initial process (see section "The Init Process" on page 101) and for finishing all processes by `MPID_SHMEM_finalize()` [def. in *mpid/ch_shmem/shmempriv.c*] .
To summarize the things noted here, figure 2 shows a possible MPI session, where process $P1$ sends a packet to $P4$ and receives a packet from $P3$ and $P4$, and process $P3$ receives a packet from $P3$.

5 Send and Receive Operations

In the following sections we will discuss the way, in which MPI sends and receives data. This will be done by regarding the two operations `MPI_Send()` and `MPI_Recv.()` Before doing so, let us consider some notes necessary to understand the send and receive mechanism.
In MPI, each packet consists of a *control message* (see glossary) and the message data. If the length of the message data is small enough, it is sent within the control message so that send and receive operations will work more efficiently[23].
The control message includes information about the communication mode (used to transfer the data), the context id, the rank of the sending process, the tag (given by `MPI_Send`), the length of the sent data, a pointer to the data and a pointer to the next

[22] their implementations depend on the underlying system (semaphores, mutexes, spin locks etc)
[23] in the current implementation a size of less than or equal to 1024 Bytes is required to deliver it within the control message. Default limit of 1024 Bytes can be change by the `pkt_size` commandline parameter or by changing `MPID_PKT_MAX_DATA_SIZE()` [def. in *mpid/ch_shmem/packets.h*]

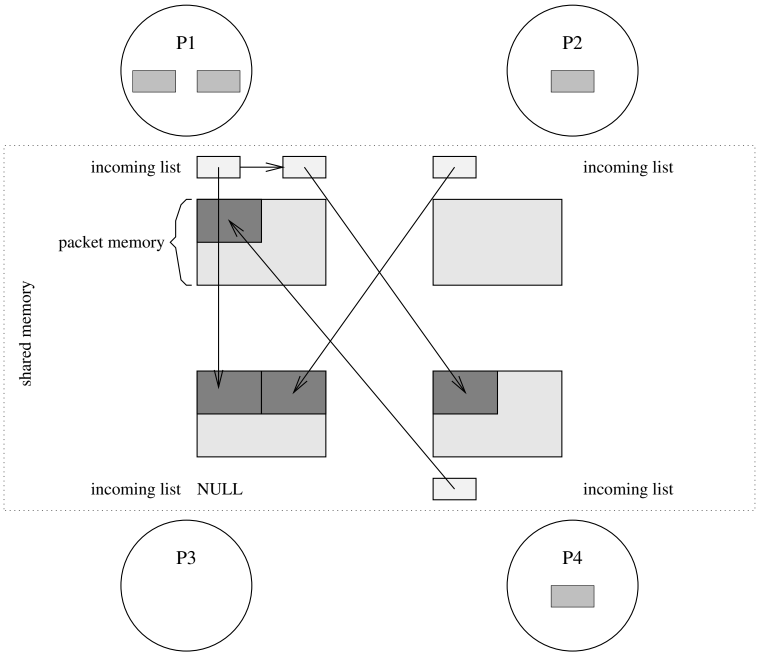


Figure 2: shared memory layout

packet, used to link packets in a list.
The communication mode fixes the type of the message (wether message data is delivered by the control message [type= `MPID_PKT_SHORT`] or not [type= `MPID_PKT_LONG`]) and defines synchronous (type= `MPID_PKT_SHORT_SYNC`) or asynchronous (type= `MPID_PKT_LONG_SYNC`) mode. It may also include commands, like `MPID_PKT_REQUEST_SEND` or `MPID_PKT_DO_GET` or `MPID_PKT_OK_TO_SEND` , which coordinate the communication processes. These commands are discussed in the following chapters, that cover send and receive operations.
The context id is set by the higher-level part of MPI and identifies the communicator context, the communication takes place in (e.g. WORLD or SELF) and the communication type (e.g. point-to-point or collective).
Rank and tag are given by `MPI_Send()` and used on part of `MPI_Recv()` to pick out the desired message.
The length field specifies the size of the whole message data, to be sent.

5.1 MPI_Send

Through this section we regard the mechanism of sending data from one process to another. In this respect we take a look at a blocking send, initiated by the `MPI_Send()` call. If you are not familiar with the memory oranisation mentioned here, then refer to section 3, starting on page 102.
To send data in blocking mode, MPI provides the `MPI_Send()` [def. in *src/pt2pt/send.c*]

call. When called, `MPI_Send()` first creates an *shandle*. The information about the destination rank, the tag, datatype of the buffer, *communicator* (see glossary) , pointer to the buffer, length of buffer, the rank of the sender and several other information not mentioned here are stored in this *shandle*. After that, `MPI_Send()` creates and setups a *dhandle* by setting the pointer of the buffer and the number of bytes to be copied[24]. Therefore we have two handles: *shandle* and *dhandle*. The *dhandle* is the device's version of the *shandle* and is placed within the structure of the *shandle*. The lower-level part mainly uses this *dhandle* and also modifies it! It is of type **MPID_SHANDLE** and defined in *mpid/ch_shmem/dmshmem.h*. *shandle* is mainly used by the higher-level part and it is defined in *include/mpir.h*.
After initializing *shandle* and *dhandle*, `MPID_SHMEM_post_send()` [def. in *mpid/ch_shmem/shmemsend.c* and then `MPID_SHMEM_complete_send()` [def. in *mpid/ch_shmem/shmemsend.c*] are called(provided that the flag **MPID_LIMITED_BUFFERS** isn't set when compiling MPI. If it is set, then `MPID_SHMEM_Blocking_send()` [def. in *mpid/ch_shmem/shmemsend.c*] is called, which marks the send process as nonblocking and then also calls `MPID_SHMEM_post_send()` and `MPID_SHMEM_complete_send()`). Both functions get the *dhandle* of the requested send (`MPID_SHMEM_post_send()` also gets the *shandle*) and thus have all information, which are necessary to handle the send.
The task of `MPID_SHMEM_post_send()` is to start (not to finish) the send. Depending on the size of the message data, `MPID_SHMEM_post_send()` either begins a short send[25] (via `MPID_SHMEM_post_send_short()` [def. in *mpid/ch_shmem/shmemsend.c*]) or a long send (via `MPID_SHMEM_post_send_long_get()` [def. in *mpid/ch_shmem/shmemsend.c*]). At first both functions allocate a packet in the shared memory region using `MPID_SHMEM_GetSendPkt()` [def. in *mpid/ch_shmem/shmempriv.c*] (the short version creates an packet of type **MPID_PKT_SHORT_T** [def. in *mpid/ch_shmem/packets.h*], the long version of type **MPID_PKT_GET_T**[def. in *mpid/ch_shmem/packets.h*], which carries additional information).
`MPID_SHMEM_GetSendPkt()` simply tries to get a free packet from the *avail* list[26]. If it fails to get one, `MPID_SHMEM_GetSendPkt()` waits until a process gives a packet back to the related sender.
After allocation, mode, context id, local rang, tag, pointer to and the length of the message data are stored in the packet. Additionally `MPID_SHMEM_post_send_long_get()` sets up a `send id` , `receive id` , the length of the delivered partial message data and the pointer to that part of the message data, not sent by now. Their meanings will be discussed next.
In case of a short send, the message data is copied into a buffer placed in the packet. Then, `MPID_SHMEM_post_send_short()` calls `MPID_SHMEM_SendControl()` [def. in *mpid/ch_shmem/shmempriv.c*] (**MPID_SENDCONTROL**). The `MPID_SHMEM_SendControl()` function just inserts the packet into the *incoming* list of the receiver. After that, the send is marked as completed (in the shandle and therefore visible from the higher-level

[24]the number of bytes is calculated with regard to the given datatype

[25]that means that the message data may fit in the control message. See also section 3, starting on page 103

[26]see section "Usage of Shared Memory"

part of MPI[27]) and the short send is done in the lower-level part of MPI and with it `MPI_Send()` finished (before returning to MPI_Send, the lower-level part checks the incoming list for arrived messages `MPID_SHMEM_check_incoming()` [def. in *mpid/ch_shmem/shmemrecv.c*]).
In case of a long send, a piece of shared memory equal to the size of the message data is requested by calling `MPID_SetupGetAddress()` [def. in *mpid/ch_shmem/shmempriv.c*]. `MPID_SetupGetAddress()` tries to get the greatest possible and necessary space in the shared memory by using `p2p_shmalloc()` [def. in *mpid/ch_shmem/p2p.c*] and copies a part of or, if possible, the whole message data into the buffer. After that, the packet is sent via `MPID_SHMEM_SendControl()`.
Due to the limited size of available shared memory, it may be possible, that the whole message data does not fit into the allocated shared memory space and therefore has to be divided into smaller parts, each of it requested [28] by and then sent to the receiver. This is the default behaviour of a long sent, startet by `MPID_SHMEM_post_send_long_get()`. After sending the first packet, the sender expects a packet with mode set to `MPID_PKT_DONE_GET()` from the receiver. If gotten, the sender checks, wether all data has been sent. If not, it sends a packet with mode set to MPID_PKT_CONT_GET and filled with the next corresponding piece of message data[29]. This is done until all message data has been sent to the receiver.

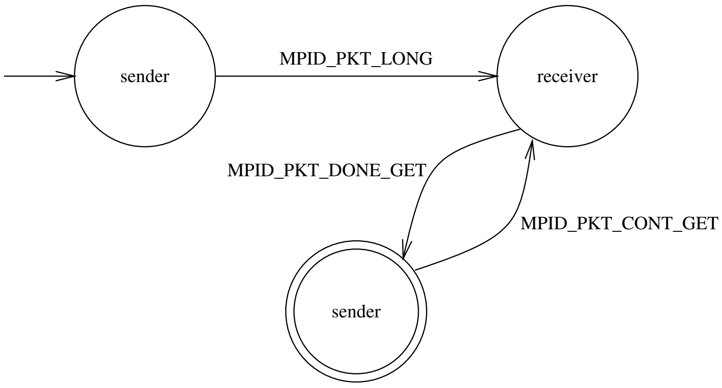


Figure 3: General behaviour of a long send

In order to manage this handshake, sender and receiver make use of the components *send_id*, *recv_id*, *len_avail* and *cur_offset*, to be found in a packet of type **MPID_PKT_GET_T**. *send_id* is used by the sender and identifies the *shandle* related to the received request. While waiting for a packet of type **MPID_PKT_DONE_GET** it is possible, that the sender receives another request(previously initiated by an asynchronous send) from a different process, so that it has to select a different *shandle*. *send_id* simply points to the correspondig *shandle* and makes the right selection possible (the receiver takes over

[27]the component `completer` in the shandle is set to 0 indicating a completed receive
[28]except for the first packte
[29]in the following, this mechanism will be named by "put-/get-mechanism"

the *send_id* from the received packet and stores it in the **MPID_PKT_DONE_GET**-packet).
recv_id gives the matching receive handle (*rhandle*) on part of the receiver. Its initial value 0 tells the receiver, that it is the first packet of a new send.
The size of the partial message data, delivered by a packet, is stored in *len_avail* and *cur_offset* points to the data, that has to be sent through the next send.
As already mentioned, `MPID_SHMEM_post_send_short()` sets up these components and calls
`MPID_SHMEM_SendControl()` to insert this packet in the *incoming* list of the receiver. After that, it assigns **MPID_CMPL_SEND_GET** to a flag in the *shandle* (the component *completer*), which indicates, that there is a send, not yet finished. Then, returning to `MPID_SHMEM_post_send()`, `MPID_SHMEM_check_incoming()` is called to receive the expected **MPID_PKT_DONE_GET** packet as response by the receiver. If arrived, `MPID_SHMEM_check_incoming()` uses `MPID_SHMEM_Done_get()` [def. in *mpid/ch_shmem/shmemget.c*] , to either deliver the next part of the message data or to mark the sending process as completed through setting component *completer* of the *shandle* equal to 0. It should be mentioned here, that `MPID_SHMEM_Done_get()` allocates a new packet via `MPID_SHMEM_GetSendPkt()` [def. in *mpid/ch_shmem/shmempriv.c*] when sending the next piece of message data instead of reusing the received packet.

5.2 MPI_Recv

In this section we do not want to describe a blocking receive operation in great detail, because its function can be deduced from the behaviour of a blocking send (through `MPI_Send()`. See previous section). Especially we take a look at the treatment of multiple packets, not necessary relating to each other[30], by the `MPI_Recv()` call.
A blocking receive can be done by the `MPI_Recv()` [def. in *src/pt2pt/recv.c*] call. Like `MPI_Send()`, it creates a handle, called *rhandle*, in which it stores the tag, context identificator, communicator, the datatype of the message data and a pointer to the buffer, to be filled with the message data and the length of the message data. This information is used to pick out the expected message from "pool" of arrived messages. In analogy to `MPI_Send()`, it also creates a handle for the lower-level part - the *dev_rhandle* - and setups its fields. Then, `MPID_SHMEM_blocking_recv()` [def. in *mpid/ch_shmem/shmemrecv.c*] (**MPID_Blocking_recv**) is called, with the corresponding *rhandle* passed to it.
Before continuing, let us imagine a situation, in which multiple messages of several processes have been arrived. How are these messages handled, when `MPI_Recv()` is started? In this respect remember, that the shared memory region is of limited space and that each sender expects the receiver to give back the packet after its evaluation[31]. The given question will be regarded in the following.
When called, `MPID_SHMEM_blocking_recv()` first uses `MPIR_search_unexpected_queue()`

[30] that means, caused by different processes for example
[31] see section "MPI_Send" for more information about allocating and freeing space for packets

[def. in *src/util/mpirutil.c*] (**DMPI_search_unexpected_queue**) to search the so called `unexpected queue` in order to find a *rhandle* containing a message, that corresponds to the one, specified by the MPI_Recv call. If found, the *rhandle* will be removed from the `unexpected queue` and after that, `MPID_SHMEM_blocking_recv()` calls `MPID_SHMEM_Process_unexpected_get()` [def. in *mpid/ch_shmem/shmemget.c*] to complete the receiving procedure and to copy the message data into the receiver's buffer and to store information about the message (source, tag, length) in the *rhandle*, specified through the `MPI_Recv()` call (the *rhandle* stored in the unexpected queue, does not necessary have to be one, that is given by `MPI_Recv()`, because when a process receives a packet, for which it does not have a *rhandle*[32], it creates a *rhandle*, fills it with the information about the received packet and inserts this *rhandle* into the `unexpected queue`).
Remember that, due to the length of the message data, it may be necessary to split the message data into several pieces and send each piece explicitly [33]. Therefore, `MPID_SHMEM_Process_unexpected_get()` uses the function `MPID_SHMEM_complete_recv()` [def. in *mpid/ch_shmem/shmemrecv.c*] , which handles subsequent put-/get-transfers through `MPID_SHMEM_check_incoming()` in order to get the missing message data from the sender. The put-/get-mechanism is descriped in section "MPI_Send".
If the message was not in the `unexpected queue` , `MPID_SHMEM_blocking_recv()` reads the next available packet through `MPID_SHMEM_ReadControl()` [def. in *mpid/ch_shmem/shmempriv.c*] (MPID_PKT_POST_AND_WAIT). First, `MPID_SHMEM_ReadControl()` checks the `local` queue[34] for available packets and if none existing, it checks the *incoming* queue. If there is also no packet, `MPID_SHMEM_ReadControl()` waits for an incoming packet.
After obtaining a packet, `MPID_SHMEM_blocking_recv()` looks up, wether this packet is the one, the user expects to get. If so, the message is read in its whole by `MPID_SHMEM_Copy_body()` followed by `MPID_SHMEM_complete_recv()`.
In case of an unexpected packet, the packet is inserted in the `unexpected queue` via `DMPI_msg_arrived()` [def. in *src/dmpi/dmpi.c*] . For that reason, when `MPID_SHMEM_blocking_recv()` is called, the `unexpected queue` is searched at first (refer to the begining of this section).
It is also possible, that `MPID_SHMEM_ReadControl()` returns a packet of type **MPID_PKT_CONT_GET** (which results from another receive) or of type **MPID_PKT_DONE_GET** (by which another process signals a completed receive). In case of **MPID_PKT_CONT_GET**[35] the message data of the corresponding packet will be copied into the buffer, that is given by the related *rhandle*. Otherwise (**MPID_PKT_DONE_GET**[36]), the next piece of message data [37]will be sent to the receiver.

[32]because there hasn't been a matching receive since

[33]refer to section "MPI_Send" starting on page 104

[34]the local queue is process-local. As soon as possible, packets are removed from the incoming queue and placed in the local queue. The local queue is not visible to other processes, so that atomic operations are not necessary to modify it

[35]given by a sender, who sent the next piece of message data

[36]given by a receiver to acknowledge the receive of a piece of message data

[37]if existing

6 Nonblocking Send And Receive

In MPI, an nonblocking send is initiated using MPI_Isend()[38] and finished by MPI_Wait() on part of the sender. The receiver calls MPI_Irecv() and then MPI_Wait() to receive a message in nonblocking mode.
MPI_Isend() [def. in *src/pt2pt/isend.c*] creates a *shandle* and a *dhandle* and then initializes the *shandle*[39]. This is done by calling MPI_Send_init() [def. in *src/pt2pt/create_send.c*] . After that, by calling MPI_Start() [def. in *src/pt2pt/start.c*] , MPI_Isend() sets up the components of the *dhandle* and starts a put-/get-mechanism through using the function MPID_SHMEM_post_send() [def. in *mpid/ch_shmem/shmemsend.c*] [40]. Then, MPI_Isend() returns back, returning *shandle* to the caller.
To finish a started nonblocking send, the user has to call MPI_Wait(), that, with respect to MPI_Isend(), has to finish a put-/get-process. Therefore MPI_Waitall() [def. in *src/pt2pt/waitall.c*] (simply called by MPI_Wait) refers to MPID_SHMEM_complete_send() [def. in *mpid/ch_shmem/shmemsend.c*] , that on its part uses MPID_SHMEM_check_incoming() [def. in *mpid/ch_shmem/shmemrecv.c*] to respond to each incoming packet of type **MPID_PKT_DONE_GET** and as a result, completes all open sends by sending all missing pieces of message data.
On the other side, the receiver uses MPI_Irecv() to start a nonblocking receive. Like MPI_Isend(), it creates a *rhandle* and a *dhandle*, initializes the *rhandle* and then calls MPI_Start(). This time MPI_Start() calls MPID_SHMEM_post_recv() [def. in *mpid/ch_shmem/shmemrecv.c*]
(**MPID_Post_recv**) [41]. MPID_SHMEM_post_recv() starts with searching the unexpected queue for the desired message. If a matching packet was found, MPID_SHMEM_Process_unexpected_get() is called, which completes the whole receive.
If it is not in the unexpected queue , MPID_SHMEM_post_recv() simply adds the *rhandle* to the so called posted receive queue and the receiving process is done.
In order to finish a nonblocking receive, the user also utilizes MPI_Waitall() (via MPI_Wait()), which ends the receiving process by calling MPID_SHMEM_complete_recv() [def. in *mpid/ch_shmem/shmemrecv.c*] (**MPID_Complete_recv**).
MPID_SHMEM_complete_recv() frequently refers to MPID_SHMEM_check_incoming() until the given receive is completed. Regarding MPID_SHMEM_check_incoming(), there might be a problem: To which *rhandle* does it has to assign the first packet of a new receive ? Remember, that in contrast to a MPI_Recv(), we do not explicitly wait for a specific packet and so do not have any *rhandle* ready. The *rhandles* of nonblocking receives are to be found in the posted receive queue . So if the first packet of a new receive arrives (e.g. a packet of type **MPID_PKT_DO_GET**), MPID_SHMEM_complete_recv() searches the posted receive queue to get the corre-

[38]also possible: MPI_Issend() (synchronous, nonblocking send), MPI_Ibsend() (buffered, nonblocking), MPI_Irsend() (ready, nonblocking)
[39]for detailed information, please refer to section "MPI_Send", starting on page 104
[40]this function and the put-/get-mechanism are mentioned in section "MPI_Send"
[41]the different behaviour of MPI_Start() depends on the type of the given handle (send-/receive-handle)

sponding *rhandle* and after that, it calls `MPID_SHMEM_Do_get()` [def. in *mpid/ch_shmem/shmemget.c*] to copy the message data, stored in the packet, into the receivers buffer, determined by the *rhandle*. Next, it sets *recv_id* of the packet equal to the pointer to the *rhandle* and sends it back with mode set to **MPID_PKT_DONE_GET**. If the sender expects to send the next piece of message data, then it delivers a packet of type **MPID_PKT_CONT_GET**, which holds the recv_id of the previous gotten **MPID_PKT_DONE_GET** packet. Through this mechanism, when getting another piece of data (where the packet is of type **MPID_PKT_CONT_GET**), the receiver just has to read the *recv_id* to get the corresponding *rhandle* and as a result it has to search the `posted receive queue` only when receiving the first packet of a new receive (e.g. of type **MPID_PKT_DO_GET**).

7 Broadcast and Reduce Operations

Although broadcast and reduce operations are not supported by the general shared memory device and therefore realized by the higher-level part of MPI, we regard these operations, because there is a special implementation for convex machines[42], making use of shared memory.
In MPI, reduce operations may start by the `MPI_Reduce()` [def. in *src/coll/reduce.c*] function, which calls the corresponding reduce function through an array of function-pointers, pointed by *communicator->collops*. Assuming, that the reduce operation takes place within one communicator. In this case, *communicator->collops* points to the array *intra_collops* [def. in *src/coll/intra_fns.c*] (otherwise to *inter_collops* [def. in *src/coll/inter_fns.c*]), so that `intra_Reduce()` [def. in *src/coll/intra_fns.c*] is called in this context.
This function uses a tree-structured communication, in which each node uses blocking send and receive operations(via `MPI_Send()` and `MPI_Receive()`).
Similar to `MPI_Reduce()`, `MPI_Broadcast()` refers to `intra_Bcast()` [def. in *src/coll/intra_fns.c*] via *communicator->collops* and also uses a tree-structured communication and blocking send and receive operations.
`MPI_Allreduce()` [def. in *src/coll/allreduce.c*] does its work by calling `MPI_Reduce()` (with root=proc_0) and `MPI_Bcast()` (with root=proc) following.
If given explicitly[43], `MPI_Reduce()` and/or `MPI_Bcast()` may refer to functions of the lower-level part. This is done in case of using the shared memory device on a convex machine[44], where `MPID_SHMEM_Reduce()` [def. in *mpid/ch_shmem/shmemcoll.c*] and/or `MPID_SHMEM_Bcast()` [def. in *mpid/ch_shmem/shmemcoll.c*] are executed.
The algorithm, `MPID_SHMEM_Reduce()` uses, depends on the size of the given send-

[42]based on the shared memory device
[43]that means, if, during compile-time, the macros **MPID_FN_Bcast** and **MPID_FN_Reduce** are set to the corresponding functions in the lower-level part
[44]so that the macro **MPI_cspp** is set

buffer and therefore on `count` [45]. If count is less than the number of the participating processes, then `MPID_SHMEM_Reduce()` calls `MPID_SHMEM_Small_reduce()`. Otherwise the data will be parted in regions of equal size and each process operates over one by reading the related regions from all processes and storing the results into the root's[46] buffer. So each process has to copy its sendbuffer into the shared memory and make it visible to the other processes. To realize the visibility of the copied sendbuffer, each process saves the related pointer into its barrier-flag, which can be read by all processes.
`MPID_SHMEM_Small_reduce()` [def. in *mpid/ch_shmem/shmemcoll.c*] uses a tree-structured communication. It also refers to barriers in order to realize the communication and coordination of the calculating procedure. The barrier flag is used to indicate, wether the concerning process has its result ready or not and the pointer to the barrier flag (which is also visible by all processes) is set to the pointer of the region, holding the result. Therefore each node(=process) waits for its subnodes to get ready, by regarding their barrier flags[47] and then calculates over the given operands and sets its barrier flag itself equal to null[48]. When this is done up to the root process, the root process[49] starts to cancel the barrier and finishes `MPI_Reduce()` with the desired result.
In the devices version of `MPI_Reduceall()`, each process calculates over a partial region and put the result in its buffer (this algorithm is similar to the one, mentioned above). After that, all processes gather the partial results from each other.

[45] see MPI_Reduce-call
[46] determined by `MPI_Reduce-call()`
[47] set to null, if ready
[48] which causes the process to block
[49] and with it, all other processes

8 Conclusion

Through the last sections, we pointed out several features of the shared memory device, including important data structures, like *MPID_SHMEM_globmem*, basic functionalities like the put-/get-mechanism, the interaction between the lower- and the higher-level part, e.g. explained in section "Nonblocking Send and Receive" and the usage of the shared memory, especially shown in sections "Usage of Shared Memory" and "Broadcast and Reduce Operations". The corresponding descriptions should give an orientation through the implementation and by mentioning necessary names, locations, etc., it may also serve as a reference, while working on the implementation.
At the end, the author wants to note several characteristics, which are described by the sections and which may be of interest for future developement, concerning a mulithreaded MPI:

- Due to its process-oriented design, each communication results in two memory transfers: sender to shared memory and receiver from shared memory. For a multithreaded MPI, where all communicators take place in the same adress space, only one copy operation is necessary. This can be realized by just "sending" the pointer to the send-buffer.
- The underlying shared memory is allocated only at the initial process from the system and then organized by the device itself. This organization can be left out in a multithreaded enviroment.
- The size of the shared memory is limited (see section "Usage of Shared Memory", starting on page 102) and as a result, large message data is split into several packets and delivered by a done-/get- mechanism. The splitting and done-/get-mechanism can be simplified, by sending the pointer to the sendbuffer[50] and on part of the receiver, copying the buffer by only one operation in its whole.
- The whole treatment of the packets may be simplified, regarding their allocation and deallocation
- The collective operations, like `MPI_Broadcast()` and `MPI_Reduce()`, which currently are only optimized for convex machines, should also be implemented in a multithreaded version.
- A multithreaded MPI may also include put- and get- operations, which are not part of the current MPI specification.

[50]as mentioned before

References

[1] NFS ARPA and other. *Message Passing Interface*, 1994. available via ftp from info.mcs.anl.gov/pub/mpi.

[2] William Gropp. *MPICH ADI Implementation Reference Manual*, 1995. available via ftp from ftp.anl.gov/pub/mpi file adiman.ps.

[3] Ewing Lusk?? *Creating a new MPICH device*, 1995. available via ftp from ftp.anl.gov/pub/mpi/workingnote file newadi.ps.

[4] Peter S. Pacheco. *A User's Guide to MPI*, 1995.

[5] Anthony Skjellum William Gropp, Ewing Lusk. *Portable Parallel Programming with the Message-Passing Interface*, 1994. available via ftp from http://www.mcs.anl.gov/mpi/usingmpi/index.html.

9 Glossary

ADI The Abstract Device Interface. It defines the core message passing routines and other system-related features and therefore lets the higher-level part of MPI abstract from features, specific to the system.

MPI_COMM_WORLD In MPI, so called `communicators` define the scope in which a communication operator may have its effect. This scope is determined by a group of processes (`process group`) and by a context (which contains local information of a communicator). Together, group and context build a communicator. `MPI_COMM_WORLD` is such a communicator, predefined by the initial process (see p. 101). It stores references to all processes start up at initial sequence. (for detailed description see [1, p. 133ff])

MPI_COMM_SELF Predefined communicator (like MPI_COMM_WORLD), only consisting of the process itself.

control message In MPI, each message data is bound to a `control message` which describes the underlying protocol, used when sending the message data, the length of the data etc. See also p. 103.

packet A packet is used by the low-level part of MPI and enables send/receive requests by exchanging packets through the shared memory region. It contains the sending mode[51], a context identifier, the local rank of the sending process, the pointer to the buffer, holding the message data, the length of the message data and other information, depending on the sending/receiving mode. For detailed description see file *mpid/ch_shmem/packets.h*, esp. look at type `_MPID_PKT_T` , where you can see all possible types of packets.

[51] for example "short" or "long"

Principles of Parallel Computers and some Impacts on their Programming Models

Thomas Radke
tomsoft@informatik.tu-chemnitz.de
http://noah.informatik.tu-chemnitz.de/members/radke/radke.html

Wolfgang Rehm
rehm@informatik.tu-chemnitz.de
http://noah.informatik.tu-chemnitz.de/members/rehm/rehm.html

Abstract

In this paper we briefly outline some principles of parallel architectures and discuss several impacts on their programming models. At first, parallel computers are generally classified. A description of the most important classes – Multiprocessors and Massively Parallel Systems – follows, with some details about chosen machines. The corresponding programming models for shared memory and distributed memory architectures are introduced. The special relationship between machine architecture and efficient parallel programming is emphasized here. The paper concludes with some hints for the software developer where to use which parallel programming model.

1 Introduction

The last ten years have seen the employment of parallel computers for the solution of complex scientific, mathematical, and technical problems developing into key technology. The paradigm shift towards parallelism has led to changes on all levels, from machine hardware to application programs. A broad spectrum of parallel architectures has been developed.

In general, a parallel algorithm can only be efficiently implemented if it is designed for the specific needs of the architecture. Thus the knowledge of primary computer design principles is of course relevant for software developers as well as numerical analysts in the field of computational physics. This fact is often underestimated by software developers.

For this reason in the following we present a brief introduction into basic architectures of parallel computers.

2 Overview on Architecture Principles

Before the development of "vector computers" in the 1970s, so-called "mainframes" were also used for scientific computing although they have typically been the workhorses of data processing departments.

The first supercomputer architectures involved the use of one – or, at most, a few – of the fastest processors that could be obtained by increasing the packing density, minimizing switching time, heavily pipelining the system, and employing vector processing techniques, which apply a small set of program instructions repeatedly to multiple data elements.

Vector processing has proven to be highly effective for certain numerically intensive applications, but much less so for more commercial uses such as Online Transaction Processing or databases.

In fact the sheer computational speed was achieved at substantial costs, namely by sophisticated highly specialized architectural hardware design and the renunciation of such techniques as Virtual Memory (to facilitate the programmability). In particular, the last fact has led to the development of a considerable body of specialized program code.

Table 1: (Vector) supercomputer performance

Computer type	MFLOPS/Processor	Clock Rate [μs]
Cray Y–MP	300	6.0
Cray C90	952	4.2
NEC SX 3/14R	6400	2.5
Fujitsu VP 2600/10	5000	3.2
Hitachi S–3800/180	8000	2.0

Another way that respects conventional programmability has led to the design of so-called multiprocessor systems (MPS). Only small changes to previous uniprocessor systems had to be made by adding a number of processor elements (PEs) of the same type to multiply the performance of a single processor machine. Although there were effects on the programming model, at least the essential fact of an unified global memory could be maintained.

Further developments discarded the demands on a unified global memory because of the impossibility of its physical realization where hundreds and thousands of processors are used. The total memory is distributed over the total number of processors; each one having a fraction in the form of a local memory.

In the 1980s the first massively parallel processors (MPP) began to appear, with the single goal of achieving far greater computational power than vector computers at greatly improved price/performance ratios by using low cost standard processors.

A still essentially unsolved problem for the use of such systems is the development of appropriate programming models. No standard programming model which satisfies the needs of all applications has yet been found although a variety of competing models have been developed, including message passing, data-parallel programming, and the virtual shared memory concept. However, the efficient use of parallel computers with distributed memory requires the exploitation of data locality, which can indeed be found in most important numerical applications.

Because it is easier to bring activities onto established architectures than to do so on parallel machines, high-performance workstations are often still prefered for program implementations. If the performance needs increase then a cluster of interconnected workstations (WSC) can also be considered as a parallel machine. But typically the interconnection network of such clusters is characterized by relatively small bandwidths (some MBytes/s for 1 KByte messages) and high latency[52] (in the range of milliseconds for 1 KByte messages). Thus suitable applications are of a competitive rather than cooperative type (with naturally high communication requirements).

Nowadays we realize that all mentioned types – MPS, MPP, and WSC – as well as advanced types of vector computers (multivector computers, see Table 1, [5]) are integrated in a network environment and can be combined to form a heterogeneous supercomputer. The recent development of message passing interface (MPI) is a landmark achievement in making such systems programmable.

Summarizing we note that each architecture has its strong and weak points and it will take continuous improvement to overcome its drawbacks. Currently, parallel computer development is heavily influenced by the technological capabilities. As a consequence we notice a trend to massive parallel arrangements of symmetric multiprocessor systems, what we call MPP/SMP.

3 General Classification

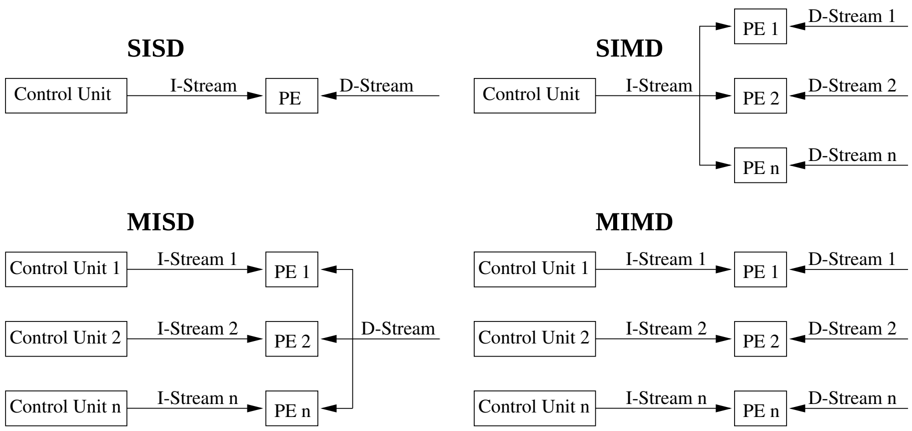


Figure 1: Flynn's classification of computer architectures

Michael Flynn [6] introduced a classification of various computer architectures based on notions of instructions (I–streams) and data streams (D–streams) (Fig. 1). Conventional sequential machines with one processing element (PE) are called Single In-

[52]Latency is the total amount of time it takes for the sender to pack the message and send it to the receiver, and for the receiver to receive the message and copy (unpack) it into its own buffer.

struction Single Data (SISD) computers. Multiple Instruction Multiple Data (MIMD) machines cover the most popular models of parallel computers.

There are two major classes of parallel computers, namely shared-memory multiprocessors and message-passing multicomputers (Fig. 2).

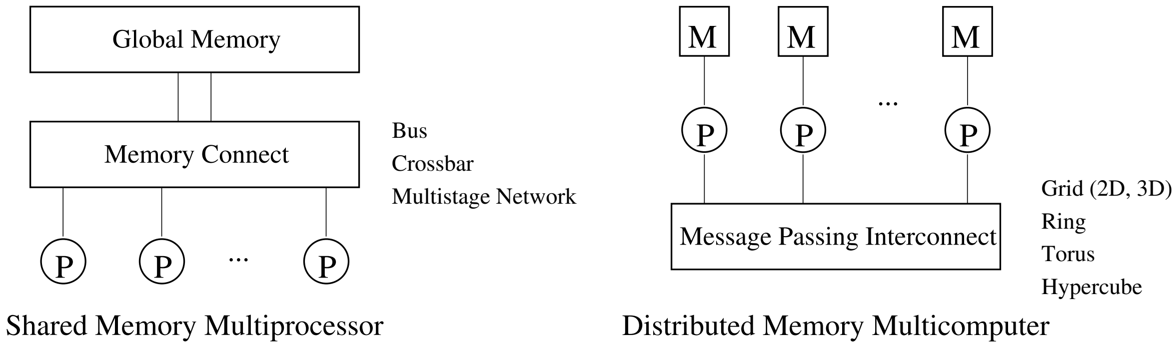


Figure 2: Parallel computer classes

The processors in a multiprocessor system communicate with each other through shared variables in a common memory, whereas each computer node in a multicomputer system has a local memory, unshared with other nodes. Interprocessor communication is done here through message passing.

4 Multiprocessor Systems

A Multiprocessor System (MPS) is typically a RISC-based shared-memory multiprocessor machine designed to provide a moderate amount of parallelism (up to 30 processors) to achieve more power than high-end workstations offer (for RISC-processors see Table 2). Most computer manufacturers have multiprocessor (MP) extensions to their uniprocessor product line (Table 3).

All additional processors are attached to the same global bus. Dedicated bus lines are reserved for coordinating the arbitration process between several requestors. The scalability of such systems is restricted to some dozens of processors due to the limited bandwidth of the common bus, which must be shared by all processors. The processors have equal access time to all memory nodes, which is why it is called a uniform memory-access (UMA) multiprocessor model.

On the contrary in nonuniform memory-access (NUMA) models the access time varies with the location of the memory word. This is because the memory is actually distributed but there are hardware means that the collection of all local memories forms a global address space accessible by all processors. A processor's local memory can be accessed faster than a remote one. Such a logically shared memory based on physical distributed memory is called a Virtual Shared Memory (VSM), especially if

Table 2: Performance of some RISC CPUs

CPU Type	Clock Rate [MHz]	Perf. [MIPS]	CPU Type	Clock Rate [MHz]	Perf. [MIPS]
Alpha 21164	300	1200	MPC601	80	240
MPC604	100	400	MPC620	133	532
SuperSparc	60	180	UltraSparc	167	668
PA7200	140	280	R4400SC	150	150
R10000	200	800	MC68060	50	100
Pentium 100	100	200	Pentium Pro	133	399

Table 3: Multiprocessor systems

Company	Model	Scalability [processors]	I/O Type	Bus Bandwidth
Sequent	Symmetry 5000	1...30	symm.	240 MByte/s
Silicon Graphics	PowerChallenge	1...30	symm.	1.2 GByte/s
Sun	SPARCstat. 20 HS14	1...4	symm.	–
Compaq	ProLiant 4000	1...4	symm.	267 MByte/s

there is essential hardware support to realize this (Fig. 3). One special version of a VSM architecture is a cache-only memory architecture (COMA) such as in the KSR-1 machine (Fig. 4 [2]). Caches copy data from other caches if necessary. There is a continuous process of data migration. A cache attracts the needed data, and in the ideal case the user is completely freed from predefining the data layout.

Drawbacks of such wonderful architectures lie in the synchronization costs for maintaining the cache coherency as well as the global synchronization (via semaphores). For further modifications of COMA models see [8].

Another distinction can be made between asymmetric and symmetric multiprocessor systems (Fig. 5). When all processors have equal access to all peripheral devices the system is called a symmetric multiprocessor (SMP). All processors are equally capable of running the executive programs, such as the operating system kernel and I/O service routines. In asymmetric systems only a master processor can execute the OS and handle I/O. Thus I/O becomes a bottleneck. Such systems we find today often in form of two-processor stations whereas symmetric solutions signify 4-processor board-based workstations or servers (Table 3).

To overcome the drawbacks of limited speed of a unified common global bus, connection schemes with crossbar technology have recently been developed [3]. The advantage

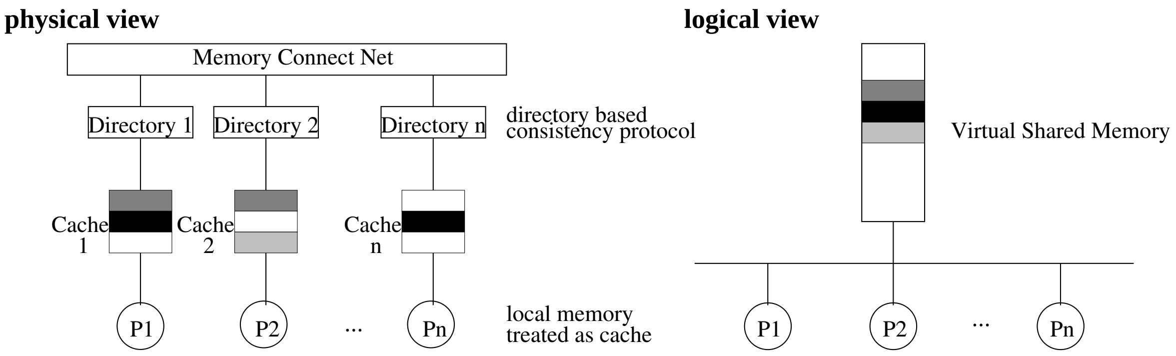


Figure 3: Virtual shared memory

is that more than one connection can be active (dark points in Fig. 6) at the same time. The achievable transfer rates can be about 600 MByte/s per CPU. The global bus is still in use but only as a broadcast medium for the snooping-bus cache-coherence mechanism [8].

5 Massively Parallel Processor Systems

Massively Parallel Processor systems (MPP) usually consist of from hundreds to several thousands of identical processors, each of which has its own memory (distributed memory). The processors communicate with each other by message passing. There is no common global memory, although there are some approaches supporting a virtual shared memory by combinations of hardware and software. In this sense the KSR Virtual-Shared-Memory computer can be classified as an MPP system.

Distributed memory multicomputers are most useful for problems that can be broken down into many relatively independent parts, each of which requires extensive computation. The interactions should be small because the overhead of interprocessor communication can degrade the system performance. The main limiting factors are bandwidth and latency. Modern communication system techniques use special latency reduction protocols such as wormhole routing. Moreover, different latency hiding methods in software may be applicable.

A fully connected network (clique) is applicable only for small numbers of nodes. To provide high-speed connections among individual processing nodes most parallel machines employ 2D or 3D crossbar switches, e.g., the Cray T3D and Hitachi SR2201 (Fig. 7). Table 4 shows the characteristics for prominent networks.

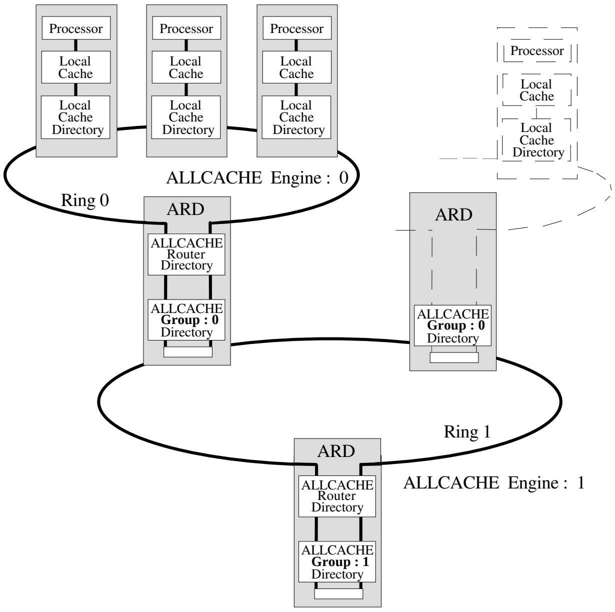


Figure 4: KSR ALLCACHE architecture

6 Multiple Shared-Memory Multiprocessors

One approach – especially a technology-driven one – for building a massively parallel system involves multiple shared-memory multiprocessors connected by a very high bandwidth interconnect, such as HiPPI, in an optimized topology.
One such interconnection of high-performance shared-memory multiprocessors (of MPP/SMP type), the PowerChallenge array from SGI Corp., has been demonstrated to solve so-called "Grand Challenge" problems.

A node in a message-passing interconnect is represented by a full SMP. A great advantage of such arrangements is that the computation-to-communication ratio (a measure for the proportion of maximum computational power and communication peak performance) can be very high. That is, the amount of message passing is low compared with the amount of work to be done in each SMP node for each message sent.

7 Multithreading Programming Model

With the evolution of MPS originating from conventional uniprocessor machines, the programming of such systems was historically formed by features of UNIX. This classical operating system allowed the quasi-concurrent execution of several tasks (multitasking) and provided some mechanisms for inter-process-communication (e.g. pipes, sockets, shared memory segments). These kernel services were quite expensive in their implementation (they are based on underlying standard network protocols) and caused

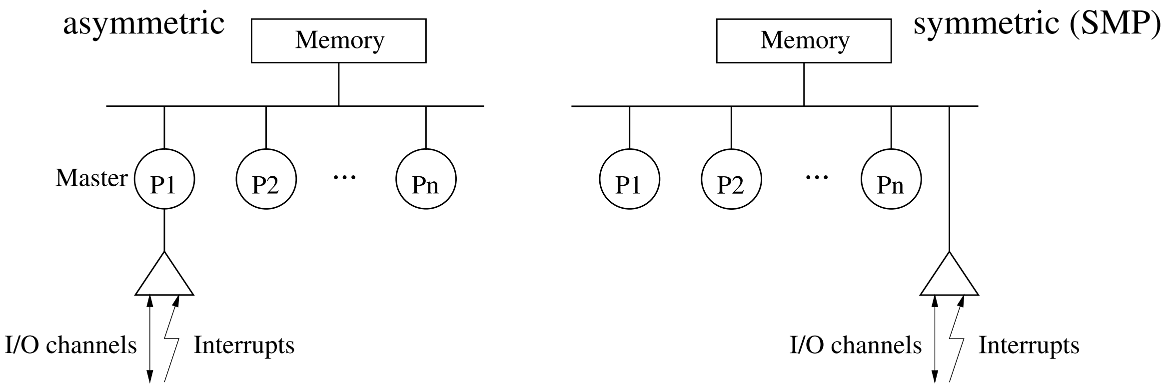


Figure 5: I/O types of multiprocessor systems

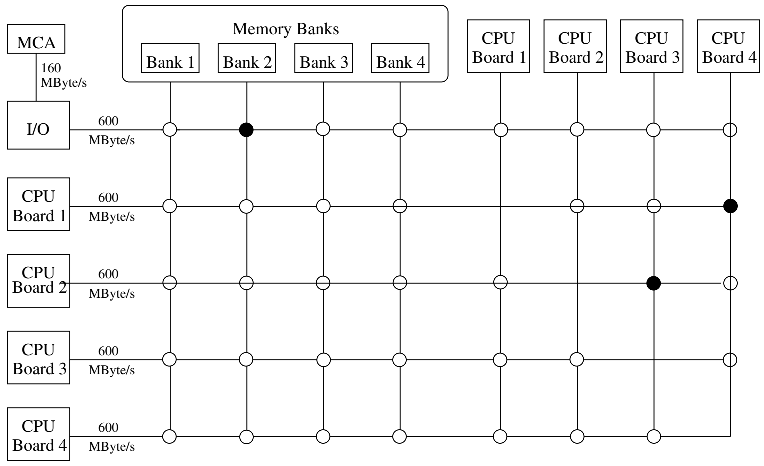
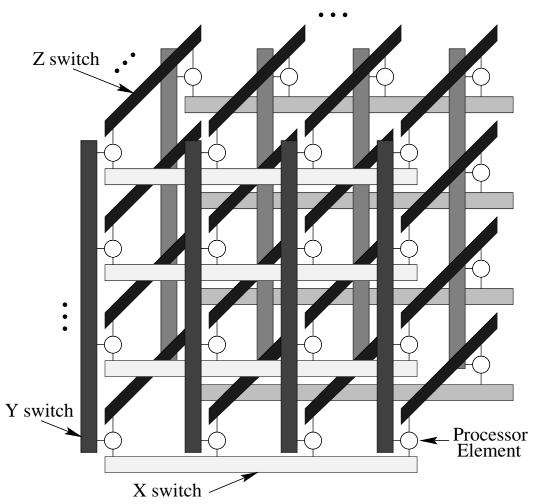


Figure 6: Crossbar switch

high overhead. So they seemed to be unsuitable for efficient parallel programming.

For this reason the traditional task concept of UNIX was extended in a manner so that a process can have more than one single execution flow and may be divided into several threads of control which are independent of each other and thus can be executed in parallel. In this programming model a thread can be thought of as a light-weight process with much less state information than a normal UNIX task – it just owns a stack, a register set, and a program counter. All threads see the same address space. Communication between threads is performed through shared memory variables. Access to these variables is managed by synchronization primitives (e.g. mutexes, semaphores, monitors, ...).

In general, the application programmer does not have to worry at all about the mapping of his threads onto the processor set of the MPS. This functionality can be fully implemented in the operating system kernel (pure kernel-level threads) or is part



- message transmission network for linking PEs
- A crossbar switch consists of 3 crossbars, one for each axis, to create 2D and 3D structures.
- At each level, a crossbar switch is capable of switching up to 8*8 connections.
- Data transfer rate : 300 MByte/s in each direction of the bidirectional ports

Figure 7: 3-dimensional crossbar network in the Hitachi SR2201 [1]

of a thread library with kernel support (mixed user/kernel level threads).

The exclusive use of kernel level threads is reasonable if the number of threads does not exceed the number of processors in the system. Each thread is fixed bound to its own processor and can run fully parallel to others. Synchronization can be implemented as busy waiting (the associated processor is not released but spins on a condition to become true).

If there are more threads than available processors (and this is the most frequent case) busy waiting between threads is no longer applicable because of possible deadlocks. In this case synchronization can cause a thread switch on a processor. The switching of kernel-level threads can only be done in the kernel, i.e., special system calls are needed. The resulting overhead (kernel thread context switch plus entering/leaving the kernel) may drastically decrease the efficiency of a pure kernel-level thread management. That is why a mixed thread management is more favourable in this case: the programmer uses user-level threads which are managed by a thread library; these user-level threads are internally mapped onto some kernel-level threads with their number corresponding to the number of available processors (see Fig. 8). Thread context switches can now completely be done in user mode, and time-expensive system calls are unnecessary.

It is true that the problem of optimal load balancing in MPS is not as difficult as in MPP. Because of the global shared memory every thread can principially be scheduled on any processor without explicite migration. But in NUMA architectures, thread locality must be taken into consideration for achieving efficient multithreading. For instance, if there are still some thread data in a processor's cache this thread should be scheduled with precedence on that processor again. This technique, called memory conscious scheduling, is especially used in systems with multi-level memory hierarchies [4].

Table 4: Properties of interconnection networks

Type	Degree	Connections	Diameter	Bisectional Width	Symm.
Clique	$N-1$	$N(N-1)/2$	1	$(N/2)^2$	yes
Linear chain	2	$N-1$	$N-1$	1	no
Ring	2	N	$[N/2]$	2	yes
Binary tree	3	$N-1$	$2((\log_2 N)-1)$	1	no
2D grid	4	$2N-2\sqrt{N}$	$2(\sqrt{N}-1)$	$2\sqrt{N}$	no
2D torus	4	$2N$	$2[\sqrt{N}/2]$	$2\sqrt{N}$	yes
Hypercube	$\log_2 N$	$N\log_2 N$	$\log_2 N$	$N/2$	yes

The efficiency of I/O intensive multithreaded applications strongly depends on the I/O architecture. In asymmetric systems every I/O operation forces a thread switch onto the master processor (which is capable of serving the request, see Fig. 5). So the master may become a bottleneck. Only SMP systems guarantee a scalable I/O performance because each I/O request can be served on the processor where the thread resides. This circumstance is less decisive for multithreaded programs with a high ratio of computation to communication.

At present, there exist a number of modern commercial and noncommercial standard operating systems that support multithreading and symmetric multiprocessing: Solaris 2.x from SUN, Mach, Linux-SMP as public domain software, Windows-NT from Microsoft). Research is aimed at the development of a unique multithreading programming interface (currently proposed as "POSIX 1003.4a Threads Extension Draft"). With this, the application programmer should be able to easily port his programs on any MPS architecture.

8 Message Passing Programming Model

Message passing is the natural programming model for distributed memory architectures. It is based on Hoare's CSP concept (communicating sequential processes [7]) where an application consists of several sequential tasks that communicate with each other by exchanging data over communication channels. These tasks are distributed among the nodes of a MPP and thus are executed in parallel. The communication channels are mapped onto the communication network. The communication hardware in modern MPP systems is capable of operating independently of its assigned compute node so that communication and computation can be done concurrently.

The efficiency of the parallel application is essentially determined by the quality of mapping the process graph with its communication edges onto the underlying dis-

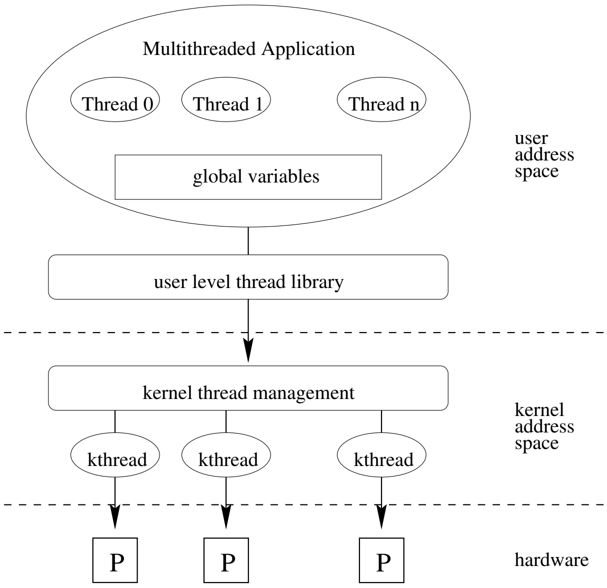


Figure 8: Multithreading with user- and kernel-level threads

tributed memory architecture (see Fig. 9). In the ideal case each task gets its own processor, and every communication channel corresponds with a direct physical link between both communication nodes.

This can be realized in most cases from the view of available processors in massively parallel systems. However, scalability requires a relatively simple communication network (2D, 3D grid, ring, torus) so at this point compromises are unavoidable. For instance, a logical communication channel is routed when it passes one or more grid points. This transfer of data takes time especially if there is no hardware support and the routing must be done by software emulation.

On the one hand, communication paths with different delays arise by nonoptimal mapping of communication channels onto the network, on the other hand several logical channels are multiplexed on one physical link. From the application programmer's point, the usable communication bandwidth is decreased.

Since the beginning of the development of MPP, algorithms for various application classes with static problem sizes emerged that find the optimal mapping scheme for a given machine topology and thus allow best exploitation of hardware performance. The identical transformation on other topologies is often combined with a loss of efficiency. That is why porting a parallel application requires at least some basic knowledge from the programmer about the target architecture.

In recent years research activities were extended to the field of adaptive parallel algorithms development, i.e., of those application classes for which the process graph is adapted to the problem size dynamically. The decision of how to inbed the actual process graph into the processor graph cannot be made statically at compile time but

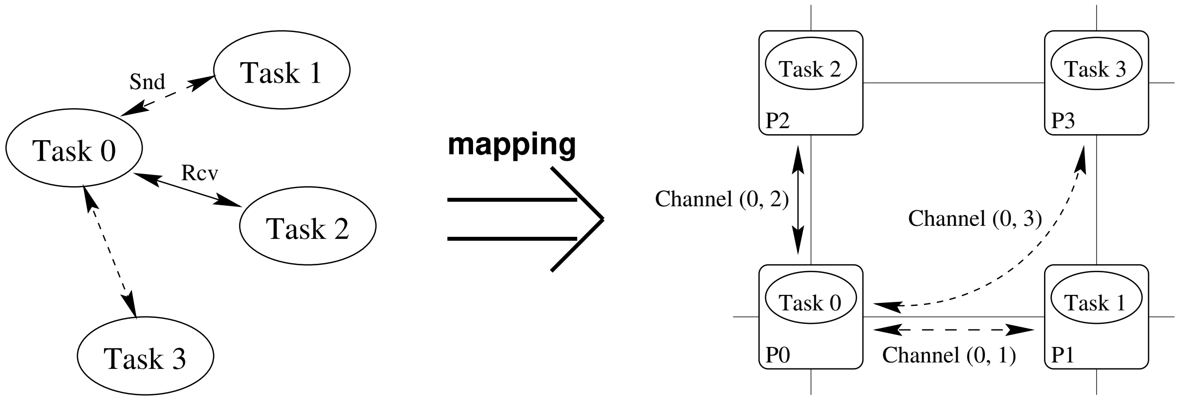


Figure 9: Mapping of the process graph onto the MPP

only at runtime. Newly created tasks should be placed on processors with less workload to ensure a load balance. In addition, the communication paths to other tasks should be kept as short as possible and not be overloaded by existing channels.

Those highly complex decisions cannot be made by the application programmer alone anymore. At this point the operating and runtime system of the MPP has to provide suitable process management functionalities (e.g. for obtaining status information on the current system workload, task placement, and migration facilities) to support the programmer in his difficult job.

9 Summary

Parallel machines can be classified as multiprocessors and massively parallel systems. These classes differ in the scale of parallelism and the memory architecture. While multiprocessors have equal access to a global shared memory and thus are limited in processor number, the latter are scalable to hundreds up to thousands of processors with each node having its own local memory.

The hardware architecture determines the way in which the parallel computer is programmed. For multiprocessors the multithreading programming model is prefered. Parallelity in application programs is expressed here as cooperation of several threads of control which share some global data and synchronize with each other. Message passing is used in distributed memory systems. Processes are executed in parallel on different processor nodes and communicate over channels.

The ratio of communication to computation in a parallel program is decisive for its efficiency. Massively parallel computers provide a high computational power but typically have a lower communication bandwidth so that I/O intensive applications probably achieve poor performance. For this class of applications Multiprocessors with their extremely low communication costs would be better suited. The application programmer has to keep these facts in mind when implementing his algorithms on a target architecture.

References

[1] Hitachi Product Sheet SR2201, Hitachi Corp.

[2] KSR *Parallel Programming – Manual*, (Kendall Square Research Corporation, Massachusetts, 1993)

[3] *Cross-Bar und Snooping-Bus*, iX no. 7, (1995)

[4] F. Bellosa, *Memory Concious Scheduling and Processor Allocation on NUMA Architectures*, TR–I4–5–95, Computer Science Dep., Univ. Erlangen, (1995)

[5] J. Dongarra, *Performance of Various Computers using Standard Linear Equation Software*, Tech. Report January 7, 1995, Computer Science Department, Univ. of Tennessee, Knoxville, (1995)

[6] M. Flynn, *Some Computer Organizations and their Effectiveness*, IEEE Trans. Computers, no. 21, 948 (1972)

[7] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, (1989)

[8] K. Hwang, *Advanced Computer Architecture*, Mc Graw-Hill, (1993)

Redaktion:
Fakultät für Informatik
Prof. Dr. W. Rehm
Tel.: 0371/531-1420, Fax: 0371/531-1628, e-mail: rehm@informatik.tu-chemnitz.de

Assistenz: Dipl. Math. L. Grabowsky, e-mail: grabowsk@informatik.tu-chemnitz.de