Computer Architecture Technical Report

TUCZ / RA-TR-02-97

Nutzung von MPI für parallele FEM–Systeme

L. Grabowsky, Th. Ermer, J. Werner

University of Technology Chemnitz–Zwickau
Department of Computer Science
Computer Architecture
Prof. Dr. W.Rehm

Hinweis:
Dieser Beitrag ist erschienen in der Preprint-Reihe
des Chemnitzer SFB 393 unter der Bezeichnung SFB393/97-08.

Zusammenfassung

Der Standard des Message Passing Interfaces (MPI) stellt dem Entwickler paralleler Anwendungen ein mächtiges Werkzeug zur Verfügung, seine Software effizient und weitgehend unabhängig von Details des parallelen Systems zu entwerfen.

Im Rahmen einer Projektarbeit erfolgte die Umstellung der Kommunikationsbibliothek eines bestehenden FEM-Programmes auf den MPI-Mechanismus. Die Ergebnisse werden in der hier gegebenen Beschreibung der Cubecom–Implementierung zusammengefaßt.

In einem zweiten Teil dieser Arbeit wird untersucht, auf welchem Wege mit der in MPI verfügbaren Funktionalität auch die Koppelrandkommunikation mit einem einheitlichen und effizienten Verfahren durchgeführt werden kann.

Sowohl für die Basisimplementierung als auch die MPI–basierte Koppelrandkommunikation wird die Effizienz untersucht und ein Ausblick auf weitere Anwendungsmöglichkeiten gegeben.

Inhaltsverzeichnis

1 Einführung

Ausgangspunkt der hier vorgestellten Arbeiten war ein an der Fakultät für Mathematik der TU Chemnitz entwickeltes FEM–System für 2D–Potentialprobleme sowie das gegenwärtig in der Entwicklung befindliche entsprechende 3D–System[AMT95].

Die Kommunikation in diesen Systemen setzt auf eine spezielle Kommunikationsbibliothek, die sog. Cubecom–Bibliothek [HHMP95], auf, in der die Kommunikation auf der Hypercube–Topologie basiert.

In dieser Topologie sind kollektive Operationen sehr einfach formulierbar, fraglich ist jedoch, ob damit auch die größtmögliche Effizienz erreicht wird. Zudem bereiten spezielle Kommunikationsstrukturen, wie sie etwa bei der Koppelrandkommunikation auftreten, erhebliche Probleme und werden deshalb im Originalsystem gesondert behandelt.

Um diesen Problemen zu begegnen, wurde untersucht, inwieweit sich mit der weit umfangreicheren Funktionalität, die in MPI zur Verfügung steht, eine Vereinfachung und Systematisierung der anfallenden Kommunikationsanforderungen erreichen läßt.

2 Das Message Passing Interface (MPI)

Zunächst sollen einige grundlegende Konzepte von MPI, die für das Verständnis der hier dargestellten Überlegungen erforderlich sind, kurz beschrieben werden. Für eine ausführliche Darstellung sei auf [MPI95] verwiesen.

Der mit MPI vertraute Leser kann diesen Abschnitt überspringen.

Vorrangiges Ziel der Etablierung des Softwarestandards MPI war es, für möglichst viele Parallelrechnerarchitekturen eine Vereinheitlichung von Kommunikation und Synchronisation herbeizuführen. Alle bislang unternommenen Versuche, einen solchen Standard durchzusetzen, scheiterten an einer wirklich breiten Akzeptanz der Programmierer bzw. Anwender. Bei der Entwicklung von MPI integrierte man deshalb von vornherein alle betroffenen Parteien, von den Hardwareherstellern über die Industrie bis hin zu den potentiellen Anwendern. Als Resultat dieser Arbeit wurde im Februar 1993 der Message–Passing–Interface Standard MPI präsentiert.

2.1 Funktionsumfang

MPI ist eine Sammlung von Kommunikations– und Synchronisationsfunktionen, die dem Nutzer als C– bzw. Fortran–Bibliothek zur Verfügung steht.

MPI umfaßt insgesamt 129 Funktionen, wobei eine kleine Untermenge von ca. 20 Funktionen ausreichen dürfte, um die Anforderungen der meisten parallelen Anwendungen abzudecken. Die in MPI spezifizierten Funktionen decken die folgenden Bereiche ab:

- **Punkt zu Punkt Kommunikation**

  Basis–Kommunikationsroutinen zum paarweisen Austausch von Daten. Ausgehend von elementaren Send– und Receive–Funktionen existieren eine Reihe von Varianten zum Erzielen größerer Effizienz.

- **Operationen auf Gruppenniveau**

  Definition von Prozeßgruppen–Kommunikation, wie z.B. Barrier oder Broadcast

- **Prozeßgruppen**

  Bilden und Verwalten von Gruppen von Prozessen

- **Kommunikations–Kontexte**

  Separate und voneinander geschützte "Kommunikationsräume" ermöglichen die Einbindung selbstdefinierter paralleler Algorithmen–Bibliotheken

- **Prozeßtopologien**

  Abbildung eines Satzes von Prozessen auf Topologien, z.B. mehrdimensionale Gitter

- **Umgebungsverwaltung und –untersuchung**

  Funktionssatz zur Untersuchung und Verwaltung der vorliegenden Hardwareumgebung

  Die Anwendung dieser Routinen ermöglicht ein Maximum an Korrektheit, Robustheit und Portabilität von Message–Passing–Programmen.

- **Profiling Interface**

  Bereitstellung eines Mechanismus zur einfachen Integration von Performance–Untersuchungen durch den Nutzer

2.2 Gruppen und Kommunikatoren

Gruppen und Kommunikatoren sind wichtige Basiskonzepte von MPI. Innerhalb einer Gruppe besitzt jeder Prozeß eine eindeutige Identifikation, *rank* genannt, die im Bereich $[0, \ldots, Anzahl\_Prozesse - 1]$ liegt.

Kommunikatoren bieten als herausragende Eigenschaft die Möglichkeit, voneinander geschützte "Kommunikationswelten" zu errichten. Diese Eigenschaft garantiert, daß innerhalb eines Kommunikators die dort laufende Kommunikation immer auf denselben Kontext beschränkt bleibt und somit keine Auswirkung auf den Datenaustausch in anderen Kommunikatoren möglich ist. *MPI_COMM_WORLD* steht für den initialen Kommunikator, der automatisch mit dem Start eines MPI–Programmes aufgebaut wird. Dieser umfaßt zunächst alle Tasks.

Der Nutzer besitzt in MPI die Möglichkeit, weitere Kommunikatoren hierarchisch von bereits bestehenden abzuleiten.

Jeder Kommunikator korrespondiert in MPI mit einer *Prozeßgruppe*, die die gleichen Tasks umfaßt.

MPI verlangt bei der Spezifizierung eines Kommunikationspartners immer die Angabe des entsprechenden Kommunikator–Rank–Paares. Selbst wenn eine Task mehreren Kommunikatoren angehören sollte, ist durch dieses Adressierungsschema Eindeutigkeit gewährleistet.

2.3 Datentypen

Charakteristisch für MPI ist, daß der gesamte Datenaustausch auf MPI–spezifischen Datentypen basiert, die mit den herkömmlichen Datentypen in den üblichen Programmiersprachen korrespondieren. Diese Handhabung über die MPI–Datentypen ist notwendig, um in heterogenen Umgebungen die richtige lokale Datenrepräsentation zu garantieren. Darüberhinaus werden dem Nutzer Mechanismen angeboten, die den Aufbau nahezu beliebiger nutzerspezifischer MPI–Datentypen gestatten.

Diese allgemeine Datentypen bestehen aus:

- einer Folge von Basis–Datentypen
- einer Folge von integer/byte displacements

Ein derartiges Paar von Folgen (bzw. eine Folge von Paaren) wird in MPI *type map* genannt. Die Folge der Datentypen (ohne displacements) heißt *type signature*. Es sei

$$Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$$

eine derartige *type map*, wobei $type_i$ einen Basis–Datentyp und $disp_i$ die displacements bezeichnet. Es sei weiter

$$Typesig = \{type_0, \ldots, type_{n-1}\}$$

die korrespondierende *type signature*. Diese *type map* zusammen mit einer Basisadresse *buf* beschreibt einen Kommunikationspuffer, der aus n Einträgen besteht, wobei der i-te Eintrag an der Adresse $buf + disp_i$ steht und den Typ $type_i$ besitzt.

Ein handle auf einen solchen allgemeinen Datentyp kann als Argument einer Sende– oder Empfangsoperation benutzt werden.

Ein solcher i.a. nichtkontinuierlicher Datentyp kann mittels `MPI_Pack` zu einem kontinuierlichen gepackt bzw. mit `MPI_Unpack` wieder entpackt werden.

Für die Erstellung spezieller Formen von allgemeinen Datentypen stehen in MPI eine Reihe von Funktionen zur Verfügung, so z.B. `MPI_TYPE_CONTIGOUS`, das einen Datentyp erstellt, der aus aufeinanderfolgenden Kopien des Eingangstyps besteht oder auch `MPI_TYPE_VECTOR`, mit dem gleichgroße Blöcke einheitlichen Typs erstellt werden können.

Zur Realisierung der Koppelrandkommunikation wurde `MPI_TYPE_INDEXED` benutzt, mit dem aus Blöcken unterschiedlicher Länge und unterschiedlicher displacements, aber mit einheitlichem Typ ein Datentyp erstellt werden kann.

3 Die Implementierung der Cubecom–Bibliothek

3.1 Implementierung der Kommunikationsroutinen

3.1.1 Einleitung

Die Werkzeuge von MPI ermöglichten den Verzicht auf einige, in der ursprünglichen Version der Bibliothek Cubecom enthaltenen Routinen. Insbesondere konnte auf die Ringstruktur, die zur Übertragung von Daten zu einem Prozessor bei begrenzter Speicherkapazität verwendet wurde, verzichtet werden, da Synchronisation von Senden und Empfang sowie die Wahl der optimalen Linkverbindung MPI–intern gelöst werden.

Daraus resultiert aber die Notwendigkeit, daß das diese Routinen verwendende Programm nicht an die Empfangsreihenfolge des ursprünglich verwendeten Ringes gebunden ist[1].

Die enge Abhängigkeit des FEM–Paketes von der Cubecom–Struktur erforderte aber auch einige "unschöne"Programmierpraktiken, auf die an den entsprechenden Stellen verwiesen wird. Die originalen Bibliotheken findet man in [HHMP95], die Quelltexte der MPI–Version in [Erm96].

3.1.2 Implementierung der MPI–Umgebung

Die Verwendung von MPI–Routinen erforderte den Verzicht auf die ursprünglich genutzten und von der Bibliothek TCGMSG[2] bereitgestellten Mechanismen für parallele Applikationen. Sämtliche, auf obige Bibliothek beruhende Kommunikationsfunktionen mußten zunächst durch geeignete MPI–Rufe ersetzt werden. In Anlehnung an die in [Bey96] vorgeschlagene MPI–Version für Workstation–Cluster wurden die notwendigen

[1] In dem hier als Anwendung benutzten FEM–Programmsystem SPC-PMPo2 war dies der Fall. Die Implementierung einer "echten" Ringroutine ist aber jederzeit möglich, da MPI die Kommunikation auf Topologien ermöglicht. Gleiches gilt für die anderen Ringroutinen (z.B. `Ring_Forw, Ring_Back`)

[2] Entwickelt wurde unter LINUX, entsprechend ist TCGMSG die benutzte Kommunikationsbibliothek.
Für andere Architekturen werden andere Bibliotheken genutzt.

MPI–Rufe in den Sende– und Empfangsroutinen gesetzt und in der Datei trinit.f die Initialisierung der MPI–Umgebung vorgenommen.

In folgenden Dateien wurden Änderungen vorgenommen:

- **trinit.f**
  - *TRINIT(A):* Die Initialisierung der MPI–Umgebung erfolgt hier, die COMMON–Variablen ICH (eigene Prozessornummer), NCPUS und NPROC (maximale Anzahl Prozessoren) werden gesetzt.
  - *(*TRCLOSE:) Beenden der MPI–Umgebung.
- **sendrcv0.f**
  - *SEND_CHAN_0 (N,X,NrLink):*
    Die originale Senderoutine wurde durch MPI_Send ersetzt.
  - *RECV_CHAN_0 (N,X,NrLink):*
    Die originale Empfangsroutine wurde durch MPI_Recv ersetzt.
- **sndrcvn1.f**
  - *SEND_NODE_1 (N,X,NrLink):*
    Die originale Senderoutine wurde durch MPI_Send ersetzt.
  - *RECV_NODE_1(N,X,NrLink):*
    Die originale Empfangsroutine wurde durch MPI_Recv ersetzt.

3.1.3 Routinen mit Prozeß–Prozeß–Kommunikation

Folgende Routinen enthalten einfache MPI_SEND bzw. MPI_RECV Aufrufe und realisieren somit eine Kommunikation zwischen genau zwei Partnern.

RING_OUT (N,A,B,MaxB)

Funktion: Jeder Prozessor besitzt ein Datenpaket A der Länge N und sendet dieses zu Prozessor 0. Zunächst muß Prozessor 0 die Länge des zu sendenden Datums mitgeteilt werden, erst dann kann die eigentliche Übertragung erfolgen.

Parameter:

N	Länge des Datenpaketes in Worten
A	Adresse des Datenpaketes
B, MaxB	werden nicht verwendet

Bemerkung: Der Name RING_OUT wurde belassen, da die Routine unter diesem Namen aus anderen Programmteilen gerufen wird. Die Ringtopologie wird in der MPI-Version aber nicht mehr benutzt!

RING_RECEIVE0 (N,B,MaxB)

Funktion: Prozessor 0 empfängt die von den anderen Prozessoren mittels RING_OUT gesendeten Daten. Mit einem ersten Receive-Aufruf erhält Prozessor 0 die Länge des nachfolgenden Datenpaketes, der zweite Receive–Aufruf muß sicherstellen, daß auch die zugehörigen Daten empfangen werden. Die in `status(MPI_SOURCE)` enthaltene Information garantiert den Empfang vom korrespondierenden Sender.

Parameter:

N	enthält die Anzahl zu empfangener Worte
B	Zeiger auf den Empfangspuffer
MaxB	wird nicht verwendet

Bemerkung: Die Routine muß von Prozessor 0 genau (NPROC-1) mal gerufen werden, wobei NPROC die Anzahl der an der Kommunikation beteiligten Prozessoren bezeichnet. Die Reihenfolge des Empfangs ist nicht festgelegt.

3.1.4 Routinen mit globaler Kommunikation

Diese Routinen behandeln den Datenaustausch zwischen allen Prozessen.

TREE_DOWN (N,WORDS)

Funktion: Prozessor 0 besitzt ein Datenpaket der Länge N, das an jeden Prozessor zu senden ist.

Parameter:

N	Länge der zu sendenden Daten in Worten
WORDS	Zeiger auf den Datenpuffer

Bemerkung: Es sind zwei Broadcasts notwendig, da die Kommunikationspartner die Länge des Datenpaketes vorab nicht kennen.

TREE_DOWN_0 (N,WORDS)

Funktionalität und Parameter entsprechen denen bei TREE_DOWN, es entfällt aber der erste Broadcast, da die Länge der Nachricht vorab allen Kommunikationspartnern bekannt ist.

TREE_UP (N,A,Nout,B,MaxB)

Funktion: Prozessor 0 erwartet von jedem Prozessor ein Datenpaket A mit i.A. unterschiedlicher Länge N, das auf dem Resultatsfeld B mit Gesamtlänge Nout abgelegt wird.

Parameter:

N	Länge des zu sendenden Datums in Worten
A	Zeiger auf den Sendepuffer
B	Zeiger auf den Empfangspuffer
Nout	Gesamtlänge des Resultatsfeldes
MaxB	wird nicht verwendet

Bemerkung: TREE_UP ruft die korrespondierende C-Funktion auf (Vergleiche 3.1.7).

TREE_UP_0 (N,A,Nout,B,MaxB)

Funktion: Prozessor 0 erwartet von jedem Prozessor ein Datenpaket A mit einheitlicher Länge N, das auf dem Resultatsfeld B abgelegt wird. Verwendet wird die MPI_Gather-Funktion.

Parameter:

N	Länge des zu sendenden Datums in Worten
A	Zeiger auf den Sendepuffer
B	Zeiger auf den Empfangspuffer
Nout	Länge des Empfangspuffers in Worten
MaxB	wird nicht verwendet

Bemerkung: Die MPI_Gather–Subroutine liefert als Länge des Empfangspuffers nur die Länge des vom jeweiligen Sender empfangenen Datums zurück. Nout ergibt sich damit aus NPROC*N.

CUBE_CAT(N,Xin,Nout,Hout)

Funktion: Jeder Prozessor besitzt ein Datenpaket der Länge N, welches zu jedem Prozessor zu senden ist. N ist i.A. auf jedem Prozessor verschieden.

Parameter:

N	Länge des eigenen Datenpaketes
Xin	Zeiger auf den Sendepuffer
Nout	Länge des Empfangsfeldes nach Zusammenfassung
Hout	Zeiger auf den Empfangspuffer

Bemerkung: Die von der Originalroutine verwendete Terminologie bezüglich der Reihenfolge der Speicherung der einzelnen Daten im Resultatsfeld mußte teilweise beibehalten werden, da diese im weiteren Verlauf der FEM–Berechnung erwartet wird. Die Berechnung von Displacements im C–Teil der Routine sorgt für die korrekte Reihenfolge (Jeder Prozessor erwartet seine eigenen Daten am Anfang des Puffers, die Daten der anderen Prozessoren werden in aufsteigender Prozessorordnung hintereinander gespeichert).

3.1.5 Routinen mit globalen Reduktionsoperationen

Allgemeines Vor dem Aufruf der jeweiligen Reduce–Funktion mußte eine Vektorkopieroperation eingefügt werden, weil Eingangs– und Resultatsvektor oft Aliase für denselben Speicherbereich darstellten[3], dies aber von MPI_Reduce nicht akzeptiert wird. Im FORTRAN–Teil der Routinen werden die Vektoroperationen gerufen, die eigentliche Reduktionsoperation erfolgt im C–Teil, da nur in C der Vergleich von Funktionszeigern zu realisieren war.
Funktionell erfüllen die Routinen TREE_DOx und CUBE_DOx (x = D für Doppelworte bzw. x = I für Worte) die gleiche Aufgabe, der einzige Unterschied besteht darin, daß TREE_DOx das Resultat auf Prozessor 0 berechnet und erst im Anschluß allen anderen Prozessoren mitteilt. In CUBE_DOx hingegen errechnet jeder Prozessor für sich selbst den Resultatsvektor.

TREE_DOx (N,X,Y,H,VDop)

Funktion: Eine Reduktionsoperation VDop wird über alle Prozessoren ausgeführt, das Ergebnis entsteht auf Prozessor 0 und wird von diesem mittels Broadcast allen anderen Prozessoren mitgeteilt.
Die Vektoren sind vom Typ DOUBLE PRECISION für x = D und INTEGER für x = I.

Parameter:

[3]Da die Reduce-Operationen ohnehin in C implementiert sind, wäre an dieser Stelle ein Vergleich der Zeiger möglich, um diese Kopieroperation nur auszuführen, wenn sie erforderlich ist.

N	Länge der Operandenfelder
X	Resultatsvektor
Y	Operand des aktuellen Prozessors
H	Hilfsfeld
VDop	Zeiger auf eine Vektoroperation

Bemerkung: Die Routinen rufen die Reduce-Routinen TREEUP_DOx auf und verteilen das Ergebnis mittels Broadcast. CUBE_DOx erfüllen die gleiche Aufgabe mittels Allreduce, hier wurde streng der Terminologie der originalen Routinen gefolgt. Zeitmessungen lassen keine Vorteile der einen oder anderen Routine erkennen.

TREEUP_DOx

Funktion: Eine Reduktionsoperation VDop wird über allen Prozessoren ausgeführt das Ergebnis entsteht nur auf Prozessor 0.
Die Vektoren sind vom Typ DOUBLE PRECISION für x = D und INTEGER für x = I.

Parameter:

N	Länge der Operandenfelder
X	Resultatsvektor
Y	Operand des aktuellen Prozessors
H	Hilfsfeld
VDop	Zeiger auf eine Vektoroperation

CUBE_DOx

Funktion: Eine Reduktionsoperation VDop wird über alle Prozessoren ausgeführt, das Ergebnis entsteht auf jedem Prozessor.
Die Vektoren sind vom Typ DOUBLE PRECISION für x = D und INTEGER für x = I.

Parameter:

N	Länge der Operandenfelder
X	Resultatsvektor
Y	Operand des aktuellen Prozessors
H	Hilfsfeld
VDop	Zeiger auf eine Vektoroperation

Bemerkung: TREEUP_DOx erfüllt die gleiche Aufgabe (Vergleiche 3.1.5).

3.1.6 Routinen für die Zeitmessung

Allgemeines Die Originalbibliothek bietet die Möglichkeit, die Zeiten für die Sende– und Empfangsroutinen zu bestimmen. Vom FEM-Paket werden die Zeiten für die Simulation gemessen, die Zeiten für Senden und Empfang der einzelnen Prozessoren werden aufgeschlüsselt, sowie die reine Rechenzeit (CPU–Zeit), die sich aus der Differenz der Gesamtzeit und den Zeiten für die Kommunikation errechnet. In MPI existiert mit MPI_WTIME eine Routine, die die vergangene Zeit (in Sekunden) zwischen zwei Aufrufen ermittelt, jedoch keine Möglichkeit, strikt nach Sende– und Empfangszeiten zu unterscheiden. So ist z.B. bei einem Broadcast zwar die Dauer der Abarbeitung der Routine für jeden Prozessor meßbar, eine differenziertere Aussage jedoch nicht zu treffen. Ebensowenig läßt sich bei den Reduce-Routinen zwischen Kommunikations– und Rechenzeit (die eigentliche Reduce–Operation) unterscheiden.

Es wird nach folgenden Kommunikationszeiten unterschieden:

TBCast:	Zeiten für Broadcast
TGather:	Zeiten für Gathering
TScatter:	Zeiten für Scattering
TSndRcv:	Zeiten für einfache Send– und Receive–Aufrufe
TReduce:	Zeiten für Reduktionsoperationen

Obige Variablen sind in times.inc definiert.
Zur Verwendung im FEM–Paket werden die einzelnen Zeiten auf die beiden Variablen Tinput bzw. Toutput der Originalbibliothek aufaddiert, wobei nun Tinput die reinen Kommunikationszeiten repräsentiert und Toutput die Zeiten für die Reduktionsoperationen angibt.

INIT_TIME

Funktion: Die einzelnen Variablen für die Zeitmessungen werden zu 0 initialisiert.

GET_TIME

Funktion: Die globalen Variablen für die Zeitmessung werden auf Tinput (TBCast, TSndRcv, TGather) bzw. auf Toutput (TReduce) aufaddiert und für die Übergabe an write_times (zuständig für Ausgabe) in ein Feld gespeichert. Der prozentuale Anteil der Zeiten an der Gesamtzeit wird berechnet.

Bemerkung: Die Felder zur Bestimmung der Kommunikationslast (z.B. N_out) wurden beibehalten, auch wenn in MPI eine direkte Bestimmung der gesendeten und empfangenen Datenworte nicht möglich ist. Diese werden nicht verwendet.

3.1.7 Die C-Routinen

Bemerkung: Die einzige Motivation, die Reduce–Operationen in einem separaten C–Teil vorzunehmen, ist die Tatsache, daß die Abfrage von Funktionszeigern in FORTRAN nicht möglich ist, die Kenntnis dieser aber Voraussetzung für den korrekten Aufruf der korrespondierenden MPI–Funktion ist.
tree_up_c bzw. cube_cat_c erfordern die Berechnung der Speicheradresse für den Empfangspuffer jedes einzelnen Prozesses, sowie die dynamische Allokierung von Speicher.

convert

Rufzeile: mpi_op = convert(VDop);

Funktion: convert übernimmt einen Zeiger auf eine Vektoroperation, vergleicht diesen mit Zeigern auf VDplus, VDmult, VDmin bzw. VDmax und gibt den Zeiger auf die entsprechende MPI–Funktion zurück.

Parameter:

mpi_op	Zeiger vom Typ MPI_Op
VDop	Funktionszeiger der zu testenden Vektoroperation

treeup_doX_c(local, result, number, op)

Funktion: Anwendung der zur Vektoroperation "op" korrespondierenden MPI–Operation auf die lokalen Daten "local" und Verknüpfung mit den Daten "result". Das Ergebnis entsteht auf dem Feld "result" nur auf Prozessor 0.
Verwendet wird MPI_Reduce.
X = i local und result enthalten integer–Werte
X = d local und result enthalten double–Werte

Parameter:

local	Zeiger auf das Datenfeld des eigenen Prozessors
result	Zeiger auf Datenfeld des Resultatvektors (nur auf Prozessor 0 richtig belegt)
number	Länge des local–Vektorfeldes
op	Funktionszeiger

cube_doX_c(local,result,number,op)

Funktion: Anwendung der zur Vektoroperation "op" korrespondierenden MPI–Operation auf die lokalen Daten "local" und Verknüpfung mit den Daten "result". Das Ergebnis

entsteht auf dem Feld "result" auf allen Prozessoren.
Verwendet wird MPI_Allreduce.
X = i local und result enthalten integer–Werte
X = d local und result enthalten double–Werte

Parameter:

local	Zeiger auf das Datenfeld des eigenen Prozessors
result	Zeiger auf Datenfeld des Resultatvektors
number	Länge des local–Vektorfeldes
op	Funktionszeiger

tree_up_c(N,start,Ngesamt,result)

Funktion: Prozessor 0 erwartet von jedem Prozessor ein Datenpaket mit i.A. unterschiedlicher Länge N, die in Reihenfolge der Prozessornummer hintereinander auf das Feld result abgelegt werden.

Parameter:

N	Anzahl zu sendender Worte
start	Zeiger auf den Sendepuffer
Ngesamt	Länge des Empfangspuffers
result	Zeiger auf den Empfangspuffer

Bemerkung: Die unterschiedlichen Längen der einzelnen Datenpakete erfordert zunächst das Einsammeln der Längeninformation. Prozessor 0 errechnet daraus die Displacements für den Empfangspuffer und ermittelt die Gesamtlänge Ngesamt als Summe der Länge der einzelnen Pakete.

cube_cat_c(N,start,Ngesamt,result)

Funktion: Jeder Prozessor besitzt ein Datenpaket der Länge N (N verschieden), welches zu jedem anderen Prozessor zu senden ist.

Parameter:

N	Anzahl zu sendender Worte
start	Zeiger auf den Sendepuffer
Ngesamt	Länge des Empfangspuffers
result	Zeiger auf den Empfangspuffer

Bemerkung: Entsprechend der Terminologie des FEM–Paketes erwartet jeder Prozessor sein individuelles Datenpaket am Anfang des Empfangspuffers, was eine entsprechende Berechnung der Speicheradressen verlangte.

3.1.8 Routine kettakk.F

Bemerkung: ../libs/src/DDCMcom/kettakk.F ist die einzige Datei außerhalb der Cubecom–Bibliothek, in der Änderungen vorgenommen wurden. Als einzige Routine des FEM–Paketes greift KettAkk direkt auf die im Verzeichnis **../libs/src/Cubecom/pvm3**[4] definierten Sende– bzw. Empfangsroutinen zu, also ohne die Funktionalität der in **../Cubecom/all** definierten Funktionen zu nutzen. Um gänzlich auf die architekturabhängigen Kommunikationsroutinen (siehe 3.1.2) zu verzichten, wurden die entsprechenden Rufe durch MPI_SendRecv[5]ersetzt.

3.2 Effizienz der Implementierung

3.2.1 Allgemeines

Anhand verschiedener Modellberechnungen konnte die vollständige Funktionalität der neuen MPI–Routinen überprüft werden, insbesondere wurde die tatsächliche Architekturunabhängigkeit der MPI–Implementierung untersucht. Sowohl auf unterschiedlicher Hardwarebasis (LINUX, SUN, GC/PowerPlus, GCel) als auch bei Verwendung unterschiedlicher Devices (shmem, ch_p4) waren keine funktionalen Schwächen erkennbar.
Von besonderem Interesse war natürlich das Laufzeitverhalten, speziell der Gewinn an Rechengeschwindigkeit gegenüber der ursprünglichen MPI–Variante. Insbesondere die erzielten Werte auf dem GC/PowerPlus sprechen eindeutig für den verstärkten Einsatz von MPI.
Mittels der neuen Version ist nun der Programmlauf mit Prozessoranzahlen möglich, die nicht zur eigentlichen Hypercubetopologie gehören, also auch auf Prozessoranzahlen m mit $m \neq 2^x$ mit $x = 0, 1, 2, \ldots$

3.2.2 Speicherbedarf

Der erhöhte Speicherbedarf ist der vielleicht einzig relevante Nachteil gegenüber der originalen Programmversion. Dies wirkt sich besonders in Mehrbenutzerumgebungen negativ auf das Laufzeitverhalten aus. In extremen Fällen kam es sogar zum Absturz während der Initialisierung des Programmes[6].

[4] ../pvm3 bezieht sich wieder auf LINUX

[5] Um die Unabhängigkeit des Paketes von der Kommunikationsbibliothek zu erhalten, sollte stattdessen eine exchange-Routine in **../Cubecom/all** genutzt werden

[6] Durch den erhöhten Speicherbedarf kam es auf SUN teilweise zu der Situation, daß eine gewisse Anzahl von Prozessen mit TCGMSG gerade noch erzeugbar war, mit MPI aber nicht mehr. In diesem Fall bricht MPI dann natürlich den Programmstart ab. Auf dem GCel und GC/PowerPlus war nur ein geringfügig erhöhter Speicherbedarf zu verzeichnen.

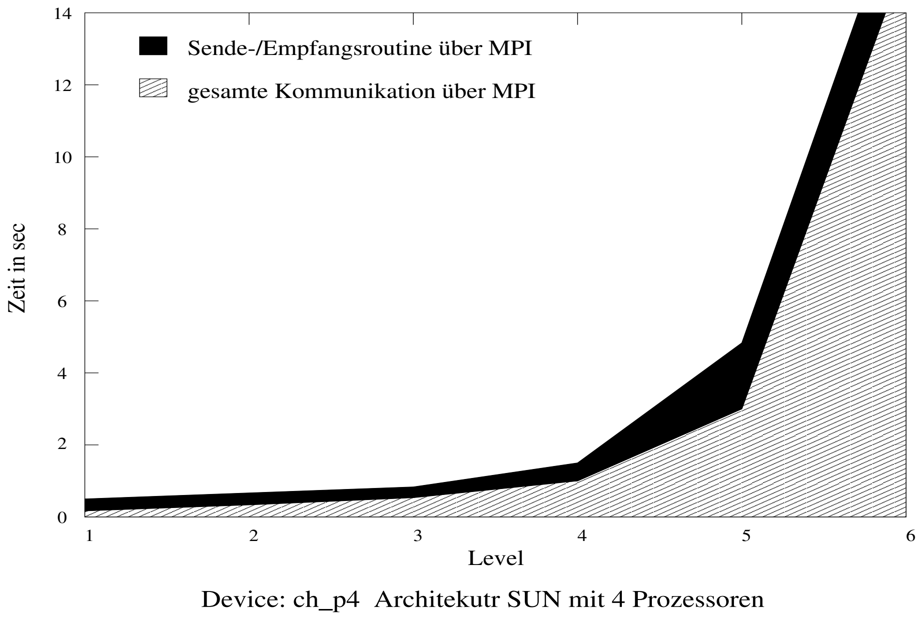

Abbildung 1: *Abhängigkeit der Gesamtzeit vom Level (distributed memory)*

3.2.3 Laufzeitverhalten

Betrachtet werden nur die Zeiten für die Berechnung der Simulation. Die Dauer der Initialisierung, der Generierung der Steifigkeitsmatrix oder für die Grafikausgabe wurden nicht berücksichtigt. Wie zu erwarten, liegt die MPI– Version in der Programm–Ladephase weit hinter der TCGMSG–Variante zurück, was mit dem gestiegenen Verwaltungsaufwand (z.B. zum Anlegen der Prozeßgruppenfiles) zu begründen ist.
Als Referenznetz für die folgenden Messungen wurde die Dreiecksvernetzung **keule16** gewählt. Gerechnet wurde auf SUN mit vier Prozessoren bzw. auf dem GC/PowerPlus mit einem Cluster aus vier Knoten (also acht Prozessoren).

SUN mit Device ch_p4 Abbildung 1 zeigt den Gewinn der vollständigen MPI–Variante gegenüber der ursprünglichen Version. Vor allem die MPI–internen Optimierungen bzgl. der Topologie zeichnen für den Geschwindigkeitszuwachs verantwortlich, ein Vorteil der MPI–Operationen gegenüber den Vektoroperationen ist ebenso zu vermuten. Keine Aussage kann über das Laufzeitverhalten im Vergleich zur Version mittels TCGMSG getroffen werden, da TCGMSG durch den verwendeten shared memory stark bevorteilt ist (Vergleiche 3.2.4).

SUN mit Device shmem Abbildung 2 zeigt einen Vergleich zwischen allen drei Versionen der Cubecom, wobei der Gewinn durch die vollständige MPI–Version gegenüber TCGMSG für den Einsatz ersterer spricht. Der abweichende Kurvenverlauf bei höherem Level hat seine Ursache teilweise sicher im erhöhten Speicherbedarf der MPI–Variante (siehe 3.2.2). Inwieweit auch andere Faktoren zu diesem Verhalten führen, muß noch genauer untersucht werden.

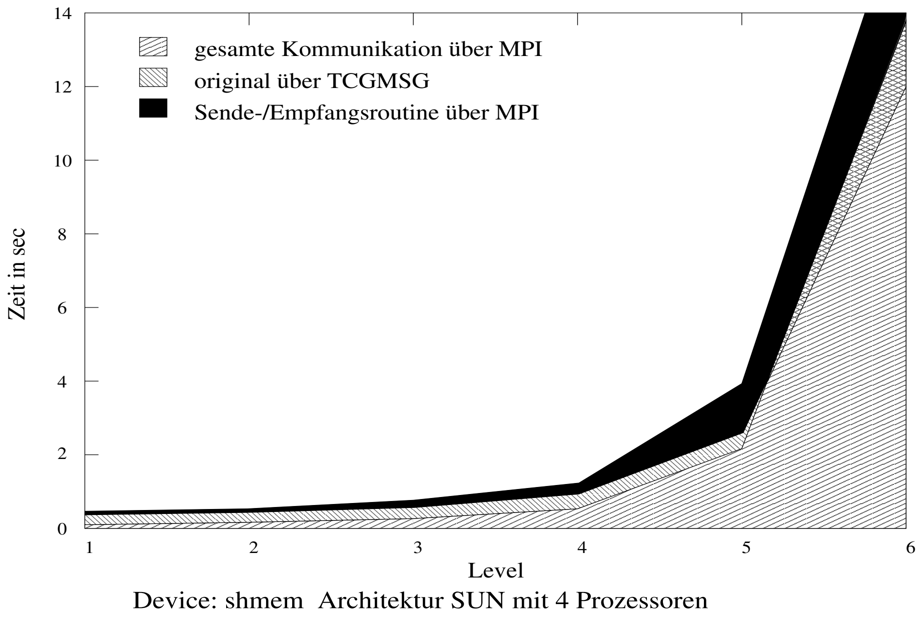

Abbildung 2: *Abhängigkeit der Gesamtzeit vom Level (shared memory)*

GC/PowerPlus Abbildung 3 zeigt einen Vergleich zwischen den verschiedenen, für den GC/PowerPlus verfügbaren Versionen der Bibliothek Cubecom. Deutlich erkennbar ist die Überlegenheit der neuen MPI–Variante gegenüber der MPI-Version, die nur die Kommunikationsfunktionen auf MPI abbildet, aber selbst die für die Architektur optimierte ppc–Variante liegt erkennbar hinter den Simulationszeiten zurück, was die Vermutung nahe legt, daß PowerMPI selbst Optimierungen für den GC/PowerPlus enthält.

3.2.4 Stabilität

Außer auf SUN konnte auf allen betrachteten Architekturen eine stabile Funktion festgestellt werden.

Das Programm beendete sich auf SUN bei Verwendung einer Levelzahl, die einen größeren Speicherbedarf erzeugt, als verfügbarer Speicher im Programm spezifiziert ist, mit einer Fehlermeldung, die als Ursache eine falsche Länge des gesendeten oder empfangenen Datenpaketes auswies. Dies wird aber normalerweise vom Programm selbst erkannt und darauf unter Ausgabe einer entsprechenden Warnung reagiert.

Da dieser Fehler bei den niedrigeren Leveln nie auftrat, ebensowenig auf GCel (MPICH) und dem GC/PowerPlus mit dem zugrunde liegenden PowerMPI, kommen als Fehlerursache Konfigurationsfehler des Programms für SUN oder aber Fehler der auf SUN verwendeten MPI–Implementierung in Frage. Dies ist noch genauer zu klären.

Das Device ch_p4 ist laut User–Guide in der Lage, selbsttätig die zugrunde liegende Speicherarchitektur zu erkennen, müßte demnach auf entsprechender Hardware automatisch auf shared memory zugreifen. Allerdings konnte diese wünschenswerte Eigen-

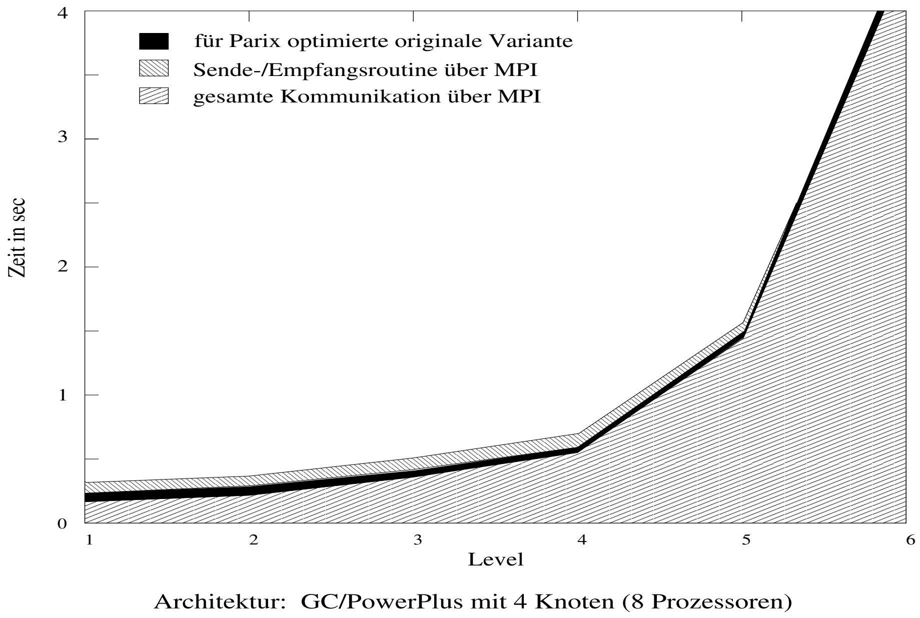

Abbildung 3: *Abhängigkeit der Gesamtzeit vom Level (auf GC/PowerPlus)*

schaft nicht immer bestätigt werden, ch_p4 verwendete auf SUN bzw.LINUX distributed memory.

4 Eine MPI–basierte Realisierung der Koppelrandkommunikation

4.1 Anforderungsspezifikation

Zum Verständnis der nachfolgenden Ausführungen werden einige Bezeichnungen benötigt, die für das betrachtete Originalsystem [HLM91, AMT95] gewählt wurden. In diesem entsteht die endgültige Vernetzung dadurch, daß ein gegebenes Grobnetz mehrfach verfeinert wird. Die Knoten dieses Grobnetzes werden als *Crosspoints* bezeichnet. Die durch die Verfeinerung zwischen den Crosspoints entstehenden Knoten werden zu *Ketten* zusammengefaßt. Diese sind jeweils durch Start-Crosspoint, End-Crosspoint und ihre Länge gekennzeichnet. Abbildung 4 zeigt die Situation für ein auf 3 Prozessoren aufgeteiltes Gebiet mit den zwischen den Prozessoren auszutauschenden Crosspoints und Ketten.

Zum Vergleich soll an dieser Stelle kurz auf die Realisierung im Ausgangssystem eingegangen werden.

Dabei wird im 2D–Fall die Tatsache ausgenutzt, daß an einer Kette nicht mehr als zwei Prozessoren beteiligt sein können. Daher lassen sich Prozessorpaare bilden, die so geordnet werden, daß eine möglichst große Anzahl von Austauschoperationen parallel abgearbeitet werden kann. Alle Ketten einer Prozessorgrenze werden dann in geeigneter

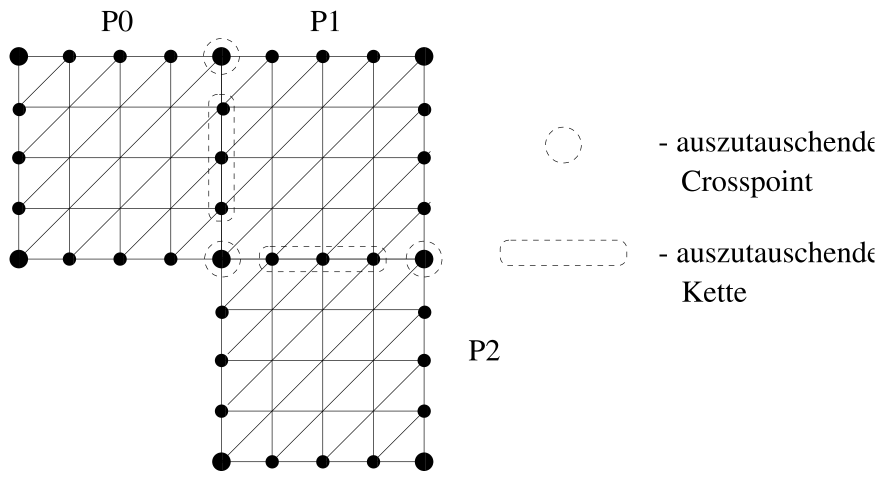

Abbildung 4: Kommunikationsstruktur für ein Beispielgebiet auf 3 Prozessoren

Weise in ein Hilfsfeld kopiert und dieses zwischen den beiden Prozessoren des jeweiligen Prozessorpaares ausgetauscht. Anschließend erfolgt die Summation korrespondierender Ketten.

Die Crosspoints werden gesondert behandelt. Jeder Prozessor kopiert dabei seine lokalen Crosspoints in einen Vektor, der die Länge der Anzahl der globalen Crosspoints besitzt, an die Stelle, die der globalen Crosspoint-Nummer entspricht. Dieser Vektor wird dann global summiert. Anschließend kopiert jeder Prozessor die seinen lokalen Crosspoints entsprechenden Komponenten zurück.

Das Verfahren im 3D–Fall ist erheblich komplizierter und soll hier nicht dargestellt werden.

Allgemeiner lassen sich die Kommunikationsanforderungen wie folgt beschreiben: Von einem Vektor sind jeweils korrespondierende Anteile gewisser Länge zwischen den beteiligten Prozessoren zu summieren. (Der Crosspoint wird dabei als Sonderfall eines Vektors mit Länge 1 verstanden). In dieser Form bleibt die Anforderungscharakteristik auch im 3D–Fall gültig.

4.2 Definition der Benutzerschnittstelle

Die im Abschnitt 4.1 recht unkonkrete Formulierung der "korrespondierenden Anteile" soll hier genauer beschrieben werden. Hierfür wird für jeden (möglicherweise) auszutauschenden Teilvektor eine **global** eindeutige Identifikationsnummer festgelegt. Zusätzlich werden der Startindex des Teilvektors (*displacement*) und seine Länge benötigt.

Damit kann man feststellen, welcher Prozessor welche der global eindeutigen Identifikationsnummern lokal besitzt. Aus dieser Information sind dann die für jeden der Teilvektoren beteiligten Prozessoren ermittelbar.

Auf der Basis von MPI bietet sich damit eine naheliegende Realisierungsvariante an.

Aus allen Teilvektoren, die zwischen genau den gleichen Prozessoren auszutauschen sind, wird über die Information der Längen und displacements mittels `MPI_TYPE_INDEXED` ein MPI–Datentyp erzeugt und aus den beteiligten Prozessoren eine Prozessorgruppe im Sinne von MPI gebildet, aus der dann ein Kommunikator erzeugt werden kann.

Damit kann mittels `MPI_Pack` ein Vektor erzeugt werden, der genau die in dieser Prozessorgruppe zu summierenden Teilvektoren enthält, dieser durch `MPI_Allreduce` bezüglich des entprechenden Kommunikators summiert und mit `MPI_Unpack` wieder zurückgeschrieben werden. Damit wird die Koppelrandkommunikation vollständig formalisierbar.

Zur Erzeugung der Datentypen und Kommunikatoren wurde eine Fortran–Routine[7] implementiert, die die folgenden Parameter besitzt:

```
      subroutine getcomm(nvec, nvecl, size, disp, ids, idsl,
     +                   ncomm, comm, types, h, ierror)

      integer nvec, nvecl, size(*), disp(*), ids(*), idsl(*),
     +        ncomm, comm(*), types(*), h(*)
```

nvec	(i):	Anzahl der (global) auszutauschenden Teilvektoren
nvecl	(i):	Anzahl der auf dem jeweiligen Prozessor vorhandenen Teilvektoren
size	(i):	Array der Länge nvecl, enthält die Längen der Teilvektoren
dips	(i):	Array der Länge nvecl, enthält die displacements der Teilvektoren
ids	(i):	Array der Länge nvec, enthält die global eindeutigen Identifikationsnummern der Teilvektoren
idsl	(i):	Array der Länge nvecl, enthält die Identifikationsnummern der Teilvektoren, die auf dem jeweiligen Prozessor vorhanden sind
ncomm	(o):	Anzahl der benötigten Kommunikatoren
comm	(o):	Array der Länge ncomm, enthält die handles auf die Kommunikatoren
types	(o):	Array der Länge ncomm, enthält die handles auf die Datentypen
h:		Hilfsfeld
ierror	(o):	Fehlervariable

Dabei sind mit (i) Eingangs– und mit (o) Ausgangsarameter bezeichnet. Die Kommunikatoren werden dabei so angeordnet, daß bei Ausführung in der durch den Feldindex in `comm` gegebenen Reihenfolge möglichst viele Prozessoren parallel arbeiten können. Werden Teilvektoren angegeben, die nur auf einem Prozessor vorhanden sind, so wird dies natürlich erkannt und diese nicht aufgenommen. Wird in der lokalen Liste (`idsl`) eine in der globalen Liste (`ids`) nicht vorkommende Nummer angegeben, so kehrt `getcomm` mit der auf diesen Wert gesetzten Fehlervariablen (`ierror`) zurück.

[7] Die vorgesehene Beispielanwendung ist in Fortran implementiert. Daher wurde zunächst diese Realisierung erstellt. Eine Nutzung in C ist jedoch möglich. Auch eine C++-Variante auf der Basis von OOMPI [LSM] dürfte mit vertretbarem Aufwand erstellbar sein.

4.3 Eine Beispielanwendung

Als Anwendung wurde das an der Fakultät für Mathematik der TU Chemnitz entwickelte FEM–Programm SPC–PMPo2 benutzt. Dazu wurde in dieses Programm die Erzeugung der für `getcomm` (s. 4.2) notwendigen Daten eingefügt. Die ursprüngliche Ausführung der Koppelrandkommunikation wurde dann durch die Abarbeitung der erzeugten Kommunikatoren in folgender Weise ersetzt:

```
        do 10 i=1,ncomm
                call mpi_pack_size(1, types(i), comm(i),
     +                  isize, ierror)
                ipos=0
                call mpi_pack(w, 1, types(i), v, isize,
     +                        ipos, comm(i), ierror)
                isize=ipos
                ih = ipos / sizeofdouble
                call mpi_allreduce(v, v(ih+1),ih,
     +                  mpi_double_precision, mpi_sum,
     +                  comm(i), ierror)
                ipos=0
                call mpi_unpack(v(ih+1), isize, ipos, w, 1,
     +                  types(i), comm(i), ierror)

10      continue
```

Dabei ist `w` der zu summierende Vektor, `v` ein Hilfsvektor. Die Konstante `sizeofdouble` ist die Größe einer Variablen vom Typ `double precision` in Byte.

Zu erkennen ist die vollständige Formalisierung des Ablaufs. Es werden außer den Datentypen `types` und den Kommunikatoren `comm` keine weiteren Informationen benötigt.

4.3.1 Effizienz der Implementierung

Die Implementierung wurde auf Parallelrechnern vom Typ GCel und GC/PowerPlus (beide von Parsytec) getestet und mit der ursprünglichen Implementierung[8] hinsichtlich der Effizienz verglichen. Bei allen Messungen wurde dabei als Gebiet ein Einheitsquadrat mit einer Grobvernetzung mit 128 Elementen benutzt. Abbildung 5 zeigt einen Laufzeitvergleich zwischen der vorhergehenden Implementierung (GCel) und der unter Verwendung des in 4.3 beschriebenen Vorgehens (MPI.GCel) auf dem GCel.

Mit Level wird dabei die Anzahl der Verfeinerungsschritte des Grobnetzes bezeichnet. Die Abbildung 6 zeigt den Vergleich für 8 und 32 Prozessoren auf dem GC/PowerPlus.

[8] Zum Vergleich wurde die in 3 beschriebene Programmversion benutzt, die im Rahmen von [Erm96] entstand. Diese wies auf fast allen betrachteten Parallelrechnern eine höhere Effizienz als die jeweilige Ausgangsversion auf.

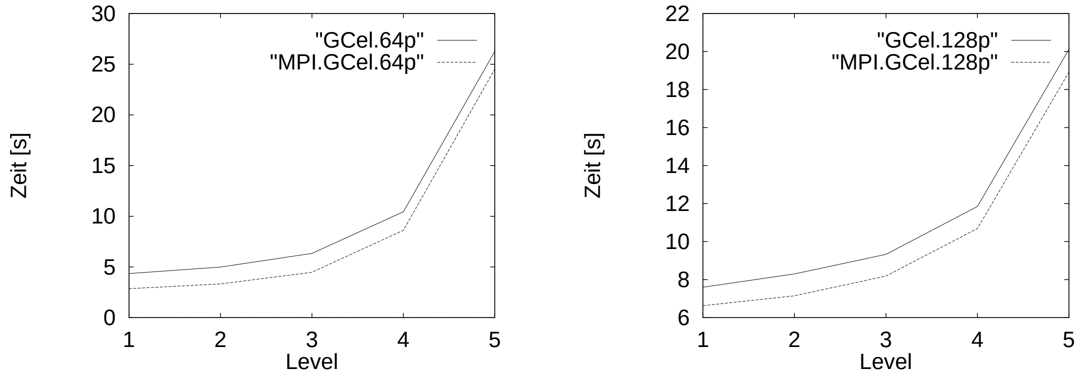

Abbildung 5: Laufzeit für 64 und 128 Prozessoren auf dem GCel

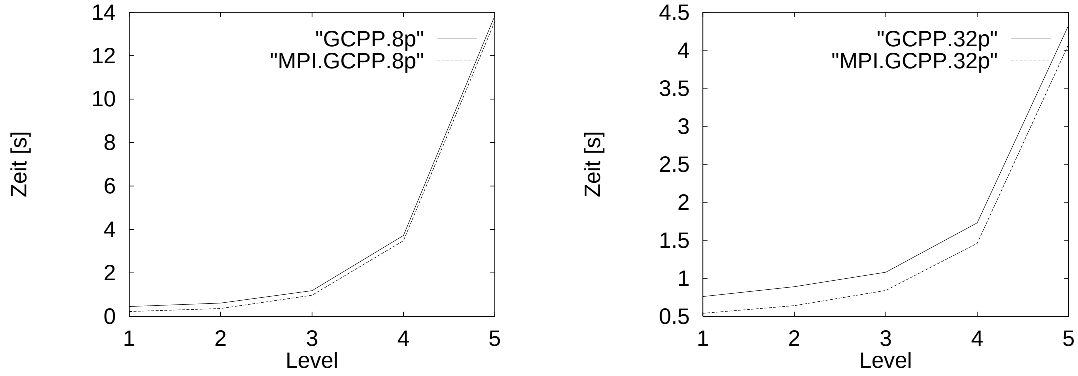

Abbildung 6: Laufzeit für 8 und 32 Prozessoren auf dem GC/PowerPlus

Die Resulate zeigen zum einen die gesteigerte Effizienz. Die Verbesserung nimmt natürlich mit wachsender Problemgröße ab, da der Gesamtkommunikationsanteil an der Rechenzeit mit der Problemgröße sinkt. Zum anderen ist zu erkennen, daß der Vorteil mit Anwachsen der Prozessorzahl größer wird. Auch dies ist leicht erklärbar. Der im Originalsystem durchgeführte globale Austausch der Crosspoints wird im Verhältnis zu der hier vorgestellten Implementierung, die mit größtmöglicher Lokalität arbeitet, immer aufwendiger.

5 Zusammenfassung

Die Ergebnisse zeigen, daß mit MPI ein wirkungsvolles Mittel zur Verfügung steht, für verschiedenste Hardwareplattformen sowohl Standardoperationen als auch komplexere Kommunikationsstrukturen systematisch und mit hoher Effizienz zu beherrschen. Die gezielte Anwendung komplexer Datentypen, speziell auf die Anwendungserfordernisse abgestimmt, kann zu einer erheblichen Vereinfachung der Programmierung führen.

Nach der hier vorgestellten Anwendung im 2D–Fall ist als nächster Schritt die Einbindung in das 3D–System geplant. Hierzu sind innerhalb der Cubecom–Implementierung keine Änderungen nötig. Lediglich einige wenige Funktionen sind noch zu implementieren, so z.B. `Ring_Forw`, das in der Grafik–Bibliothek benutzt wird.

Für `getcomm` ist eine Modifikation der Verwaltung der Identifikationsnummern erforderlich, die aber lediglich deren Repräsentation betrifft.

Gegenwärtig laufen im Rahmen dieser Gruppe Arbeiten zur Implementierung eines SCI–Devices sowie eines Multithreaded Devices, mit denen nach ersten mit Testversionen durchgeführten Messungen eine deutlich verbesserte Effizienz auf SUN zu erwarten ist.

Literatur

[AMT95] Th. Apel, F. Milde, and M. Theß *SPC-PM Po 3D Programmers Manual*, Preprint-Reihe der Chemnitzer DFG–Forschergruppe "Scientific Parallel Computing" 95_34, TU Chemnitz, 1995.

[Bey96] U. Beyer, *Portierung eines FEM-Programmes auf Workstation-Cluster mit dem Message Passing Interface (MPI)*, Studienarbeit, TU Chemnitz, Fakultät für Informatik, 1996.

[Erm96] Th. Ermer, *Weiterentwicklung einer MPI–Portierung eines FEM–Programms*, Studienarbeit, TU Chemnitz, Fakultät für Informatik, 1996.

[HHMP95] G. Haase, T. Hommel, A. Meyer, and M. Pester, *Bibliotheken zur Entwicklung paralleler Algorithmen*, Preprint-Reihe der Chemnitzer DFG–Forschergruppe "Scientific Parallel Computing" 95_20, TU Chemnitz, 1995.

[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.

[LSM] A. Lumsdaine, J. M. Squyres, and B. C. McCandless, *Object Oriented MPI (OOMPI): A C++ Class Library for MPI Version 1.0*, available from http://www.cse.nd.gov/~lsc/research/oompi/.

[MPI95] *MPI: A Message–Passing Interface Standard*, June 1995, available from http://www.mcs.anl.gov/Projects/mpi/index.html.