unix/mail, 6/94

Portable Parallelprogrammierung mit PVM

Volker Schnecke, Fachbereich Mathematik/Informatik, Universität Osnabrück

Zusammenfassung

Seit der weiteren Verbreitung der Parallelverarbeitung existiert der Wunsch nach einer einheitlichen Programmiersprache für parallele Systeme. Zur Zeit gibt es viele unterschiedliche, auf die spezielle Hardware zugeschnittene Programmiersprachen und -umgebungen, so daß eine Portierung eines Programms auf einen anderen Parallelrechner oft nur unter großem Aufwand möglich ist. Abhilfe scheint hier auf den ersten Blick das seit einigen Jahren verfügbare und weitverbreitete PVM -System zu bieten. Es wurde entwickelt, um parallele Programme auf einem vorhandenen heterogenen Rechnernetz zu implementieren, ist jedoch mittlerweile auch für nahezu alle Parallelrechner, auch massiv-parallele Systeme, zu haben.

PVM – Parallel Virtual Machine

Die *Parallel Virtual Machine* (PVM) ermöglicht es, ein heterogenes Netz von Unix-Workstations zu einem virtuellen Parallelrechner zusammenzufassen, und es besteht die Möglichkeit, echte Parallelrechner in diese virtuelle Maschine zu integrieren [1]. PVM wird seit 1990 am Oak Ridge National Laboratory und an der University of Tennessee in Knoxville entwickelt und ist zur Zeit in der Version 3.3.7 verfügbar [2].

PVM stellt eine Schnittstelle für Parallelprogrammierung unter C oder Fortran auf einem heterogenen Rechnernetz dar. Das System besteht aus Bibliotheken mit etwa 60 Funktionen, Dämon-Prozessen und einem einfachen Kommandoprozessor für die unterschiedlichen Architekturen. Der Anwender kann die Rechner eines heterogenen Netzes zu einer virtuellen Maschine zusammenstellen und auf den einzelnen Rechnern parallele Prozesse, die in PVM als *Tasks* bezeichnet werden, starten und synchronisieren, siehe Abbildung 1.

Tasks in PVM

Die Verwaltung aller Tasks unter PVM erfolgt über Dämon-Prozesse, die auf jedem Rechner existieren, der an der virtuellen Maschine beteiligt ist. Die Dämonen übernehmen die Nachrichtenverwaltung, die Verteilung der Tasks auf die einzelnen Rechner der virtuellen Maschine und die Ein-/Ausgabe der einzelnen Tasks. Der zuerst gestartete Dämon ist der Master-Dämon, der einige zusätzliche Aufgaben erfüllt, wie zum Beispiel das Rekonfigurieren der virtuellen Maschine durch das Starten bzw. Eliminieren anderer Dämon-Prozesse, was zur Laufzeit einer Applikation erfolgen kann.

Das Starten einzelner Tasks kann entweder aus einem speziellen Kommandoprozessor, der PVM -Console, heraus oder durch andere Tasks erfolgen. Dabei kann in dem entsprechenden Funktionsaufruf (`pvm_spawn()`) ein Rechner, der an der virtuellen Maschine beteiligt ist, oder eine

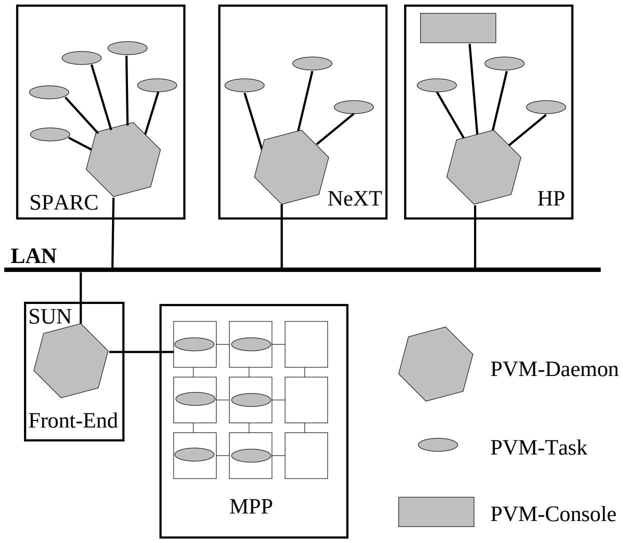

Abbildung 1: Eine parallele virtuelle Maschine

bestimmte Architektur in einem heterogenen Netz angegeben werden, auf der die Task innerhalb der virtuellen Maschine gestartet werden soll. Ohne diese Angaben werden die Tasks gleichmäßig auf die Hosts verteilt, wobei man deren relative Rechenleistung bei der Konfigurierung der virtuellen Maschine angeben kann.

Die Identifikation der einzelnen Tasks auf der virtuellen Maschine erfolgt über Task IDs. Diejenige Task, welche mittels `pvm_spawn()` weitere Tasks gestartet hat, erhält deren TID . Jede Task kann ihre eigene TID (`pvm_mytid()`) und die ihrer Vater-Tasks (`pvm_parent()`) erfragen und durch diese die TIDs der anderen Tasks auf der virtuellen Maschine erfahren. Der folgende Code zeigt die Initialisierung eines Masters im Master-Slave Prozeßmodell.

```
#include "pvm.h"

#define  NUM       4
#define  WORK      0
#define  RES      42

void main (void)
{
int info, i,
  slaves[NUM], work[NUM],
  res[NUM], total;

  info = pvm_spawn("slave",
     (char **) 0, PvmTaskArch,
     "SUN4", NUM, slaves);
```

```
  if (info != NUM)
  {  printf("%d tasks started",
         info);
     pvm_exit(); exit(-1);
  }

  work = DivideWork(NUM);

        /* Arbeit austeilen */
  for (i=0; i < NUM; i++)
  {  pvm_initsend
          (PvmDataDefault);
     pvm_pkint(&work[i], 1, 1);
     pvm_send(slaves[i], WORK);
  }

     /* Resultat aufsammeln */
  for (i=0; i < NUM; i++)
  {  pvm_recv(slaves[i], RES);
     pvm_upkint(&res[i], 1, 1);
  }
  total = ComposeResult(res);

  pvm_exit();
}
```

Die Master-Task startet, nachdem sie ihre eigene TID erfragt hat, vier Kopien des Programms `slave` auf Hosts der Architektur `SUN` innerhalb der virtuellen Maschine. In dem Vektor `slaves` erhält der Master die TIDs der gestarteten Tasks. Im folgenden werden einzelne Arbeitspakete an alle Tasks verschickt und danach die errechneten Teilergebnisse empfangen und zusammengefaßt.

Das Hauptprogramm eines Slaves kann so aussehen:

```
void main (void)
{
int master,
  work_part, res;

  master = pvm_parent();

  pvm_recv(master, WORK);
  pvm_upkint(&work_part, 1, 1);

  res = DoWork(work_part);

  pvm_initsend(PvmDataDefault);
```

```
  pvm_pkint(&res, 1, 1);
  pvm_send(master, RES);

  pvm_exit();
}
```

Kommunikation in PVM

Die Kommunikation innerhalb der virtuellen Maschine erfolgt durch den Austausch von Nachrichten (*Message Passing*) zwischen den Tasks bzw. den Dämonen. Zu diesem Zweck existieren in jeder Task entsprechende *Message Buffer*. Vor dem Versenden einer Nachricht muß dieser Platz allokiert werden, dabei ist die Größe des Puffers jedoch noch nicht festgelegt. Bei der Initialisierung dieses Puffers (`pvm_initsend()`) wird die Kodierung der Elemente dieser Nachricht festgelegt: Der Default ist XDR -Kodierung (`Extended Data Representation`), da zu diesem Zeitpunkt nicht feststeht, an welche Architektur die Nachricht innerhalb eines heterogenen Netzes verschickt werden soll. Es besteht auch die Möglichkeit, unkodierte Nachrichten zu versenden.

Nachdem der Puffer vom Dämon zur Verfügung gestellt wurde, können durch die Packroutinen beliebige elementare Datentypen zu einer Message zusammengestellt werden. Diese können dann mit speziellen Packroutinen (z.B. `pvm_pkint()`, `pvm_pkdouble()`, `pvm_pkstr()`) oder mit Hilfe eines Format-String (`pvm_packf()`) in den Sende-Puffer kopiert werden. Durch `pvm_send()` wird der Pufferinhalt an eine andere Task verschickt, wobei man der Message noch einen Typ mitgeben kann. `pvm_mcast()` verschickt die Nachricht an eine Reihe von Tasks, deren TIDs in einem Vektor übergeben werden.

Im allgemeinen Fall werden die Nachrichten zwischen zwei Tasks über die PVM -Dämonen auf den entsprechenden Hosts geroutet. Für wiederholte Kommunikation besteht die Option, daß die Nachrichten direkt zwischen zwei Tasks verschickt werden, dieses ist jedoch nur für eine begrenzte Zahl von Kommunikationskanälen möglich.

Die Nachrichten werden durch den Dämon auf der Empfängerseite gepuffert und können von einer Task mittels `pvm_recv()` abgefragt werden. Dabei kann gezielt auf eine Nachricht eines bestimmten Typs, eines bestimmten Absenders oder auf eine beliebige Nachricht für diese Task reagiert werden. Die einzelnen Komponenten der Nachricht müssen in derselben Reihenfolge, in der sie vor dem Verschicken in den Puffer kopiert wurden, durch die entsprechenden Routinen aus dem Empfänger-Puffer wieder ausgelesen werden. Neben dem blockierenden Empfangen gibt es noch nicht-blockierendes Empfangen (`pvm_nrecv()`) und Empfang mit Time-out (`pvm_trecv()`).

Das Verschicken einer Nachricht unter PVM ist immer asynchron, da die Nachrichten nach Aufruf einer Sende-Routine an den entsprechenden Dämon übergeben werden. Die Sende-Task fährt in der Programmausführung fort, ohne auf eine Bestätigung durch den Empfänger zu warten. Eine Synchronisation von Tasks wird in PVM durch Gruppenfunktionen realisiert. Dabei können einzelne Tasks dynamisch zu Gruppen zusammengefaßt werden. Die Verwaltung dieser Gruppen erfolgt durch einen Gruppen-Server, der automatisch bei dem ersten Aufruf einer Gruppenfunktion gestartet wird. Neben der Synchronisation (`pvm_barrier()`) können Nach-

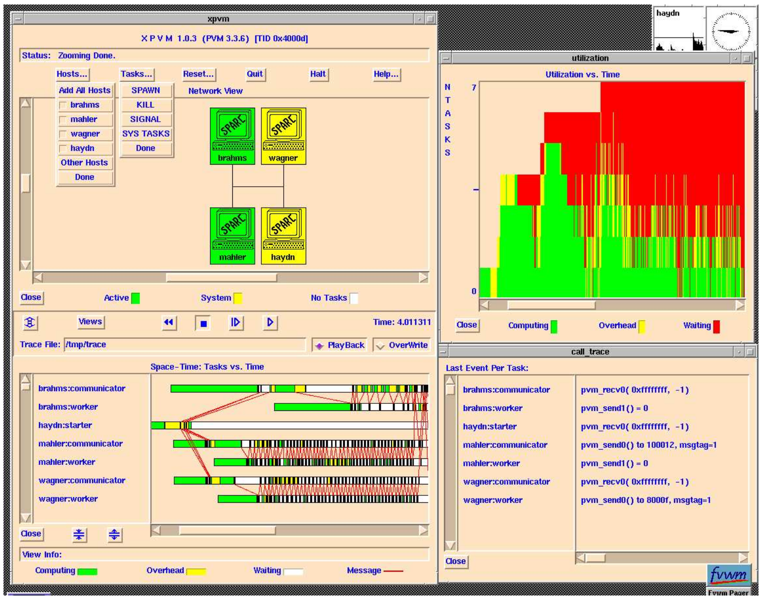

Abbildung 2: Ein Screendump von XPVM

richten an alle Tasks einer Gruppe verschickt (`pvm_bcast()`) oder es kann eine globale Funktion (Maximum, Minimum, Summe, Produkt) von diesen Tasks ermittelt werden (`pvm_reduce()`).

Insgesamt kann man sagen, PVM ist ein System, welches einen sehr einfachen Einstieg in die Welt der Parallelverarbeitung ermöglicht und alle zur Parallelprogrammierung notwendigen Funktionalitäten zur Verfügung stellt. Bei der Programmierung kann der Anwender sowohl auf herkömmliche Unix-Tools (z.B. Debugger) zurückgreifen (es ist sogar unter PVM möglich, zu jeder Task einen Debugger-Prozeß zu starten) als auch spezielle Tools zum Monitoring und zur Performance-Analyse der Parallelisierung (*XPVM, ParaGraph*) nutzen.

XPVM und ParaGraph

Bei der Entwicklung von parallelen Programmen ist es immer sinnvoll, durch ein Tool das Laufzeitverhalten der parallelen Prozesse zu betrachten, um Programmierfehler zu erkennen und die Effizienz der Parallelisierung zu steigern. Zu beobachten ist dazu die Auslastung der einzelnen Prozessoren bzw. Hosts und der Overhead der Parallelisierung. Dieser wird in erster Linie durch die Kommunikation bestimmt, d.h. durch die Zahl und Größe der Nachrichten und die Wartezeit der einzelnen Tasks auf entsprechende Nachrichten. Auch können solche Tools beim Auffinden von Kommunikationsdeadlocks sehr nützlich sein.

Ein solches Tool ist XPVM [2]. Es handelt sich dabei um eine in *Tcl/Tk* entwickelte X-Version der PVM-Console. XPVM ermöglicht anhand von automatisch erzeugten *Trace-Files* eine Visualisierung des Programmablaufs. Insbesondere wird hier der zeitliche Verlauf der einzelnen Tasks und die Kommunikation zwischen ihnen grafisch dargestellt, siehe Abbildung 2. In einem weiteren Fenster wird die von jeder Task zuletzt aufgerufene PVM -Routine angezeigt, so daß

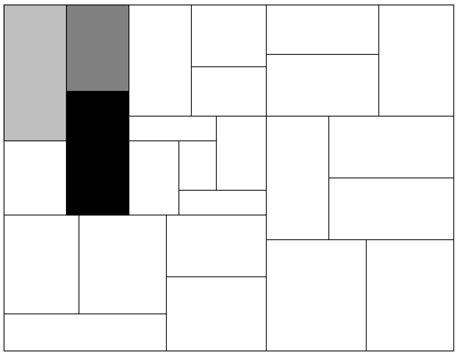

Abbildung 3: Ein Floorplan für einen Chip mit 25 Modulen

— zumindest in Ansätzen — deutlich wird, in welchen Bereichen des Programmtexts sich die einzelnen Tasks befinden.

Ebenfalls auf Log-Files arbeitet das Tool *ParaGraph* [3]. Im Gegensatz zu XPVM kann hier eine Analyse jedoch nur im Anschluß an den Programmablauf erfolgen. Dieses Analyse-Tool ist deutlich umfangreicher und allgemeiner. Es ist nicht speziell für PVM entwickelt worden, weshalb der Einsatz von *ParaGraph* in Zusammenarbeit mit PVM sehr umständlich ist. Es bietet jedoch die Möglichkeit, anhand der im Programmlauf erfolgten Kommunikation eine optimale Prozeßtopologie zu ermitteln. Dieses ist insbesondere auf echten Parallelrechnern wichtig, da hier die einzelnen Prozessorelemente nicht über einen gemeinsamen Bus kommunizieren, sondern immer nur mit aufgrund der vorgegebenen Hardware-Topologie im System benachbarten Prozessoren.

PVM auf MPPs

Mittlerweile wurde PVM auch auf viele Parallelrechner-Architekturen portiert und ist selbst für massiv-parallele Systeme (*MPPs*) mit mehreren hundert oder tausend Prozessoren verfügbar. Es ist somit möglich, Programme, die auf Workstation-Clustern entwickelt und getestet wurden, direkt auf Parallelrechner zu übertragen. Jedoch müssen dabei Effizienzverluste in Kauf genommen werden, wie an dem folgenden Beispiel deutlich wird.

Eine einfache Anwendung

Als Anwendungsbeispiel dient ein Optimierungsproblem, welches durch einen parallelen *Branch & Bound*–Algorithmus gelöst wird. Derartige Baumsuchverfahren finden in der künstlichen Intelligenz oder in der kombinatorischen Optimierung zahlreiche Anwendungen und beinhalten eine natürliche Parallelität. Die Floorplan-Optimierung (*floorplan sizing*) ist ein Problem aus dem Bereich des VLSI -Designs. Hier sind die Bausteine eines Chips auf einem Layout mit minimaler Fläche anzuordnen, wobei die relativen Positionen der einzelnen Module in der vorangehenden Plazierungs-Phase schon festgelegt wurden. Für die Module existieren mehrere alternative Ausprägungen mit unterschiedlichen Ausmaßen, gesucht ist nun eine Kombination dieser verschiedenen Alternativen, so daß sich ein rechteckiges Layout mit minimaler Fläche

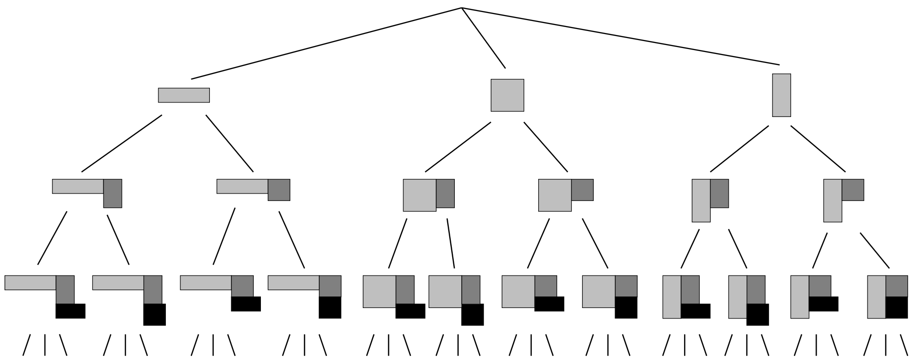

Abbildung 4: Die oberen Ebenen eines Suchbaums für das Floorplan-Optimierungsproblem

ergibt. Abbildung 3 zeigt einen Floorplan, der jedem der 25 Module eines Chips einen Raum zuweist und somit vorschreibt, welche Blöcke in dem Layout benachbart sind. Existieren für jedes Modul im Schnitt vier unterschiedliche Implementierungen, so gibt es insgesamt 4^{25} verschiedene Layouts. Diese können in einem Baum sukzessive erzeugt werden. Abbildung 4 zeigt die oberen Ebenen eines möglichen Suchbaums für den Floorplan aus Abbildung 3. Jeder innere Knoten beschreibt ein Teil-Layout, die Blätter des vollständigen Suchbaums beschreiben alle möglichen Layouts für den gesamten Chip. Ein *Branch & Bound*–Algorithmus durchsucht nun systematisch, ausgehend von der Wurzel, diesen Baum. Dabei wird die Suche an einer Stelle dynamisch abgebrochen und an einer anderen Stelle fortgesetzt, wenn anhand einer Heuristik abzusehen ist, daß aus dem bislang erzeugten Teil-Layout keine gute Gesamtlösung entstehen kann.

Die Parallelisierung

Bei jeder Parallelisierung müssen im wesentlichen zwei Aufgaben gelöst werden:

- Das Problem muß in eine ausreichende Anzahl von Teilproblemen zerlegt werden, welche dann von den einzelnen Tasks parallel bearbeitet werden (*work partitioning*).
- Die einzelnen Teilprobleme müssen auf die Tasks verteilt werden (*load balancing*).

Die Zerlegung des Problems in mehrere Teilprobleme ist im Fall der Baumsuche sehr einfach und nahezu beliebig fein möglich: Jeder innere Knoten im Suchbaum ist die Wurzel eines Teilbaums, der als Teilproblem von einer Task bearbeitet werden kann. Da im allgemeinen jedoch die Größe der bei der Aufteilung entstandenen Teilbäume im voraus nicht bekannt ist, muß zur Laufzeit eine dynamische Lastverteilung durchgeführt werden, um ein paralleles System effizient zu nutzen.

In einem einfachen Parallelisierungsansatz, der nach einem Master-Slave Prozeßmodell arbeitet, erzeugt ein Master-Prozeß die ersten Ebenen dieses Suchbaums und speichert die Suchfront, d.h. die Knoten, an denen er die Suche abgebrochen hat. Diese Knoten werden dann der Reihe nach an die einzelnen Slaves abgegeben. Sie beschreiben Teilbäume, welche die Slaves parallel durchsuchen. Ist die Suche in einem Teilbaum erfolglos abgebrochen worden, so teilt ein Slave dieses dem Master-Prozeß mit und erhält ein neues Teilproblem in Form eines einzelnen Knotens aus der Suchfront. Wird eine neue Lösung gefunden, so muß die sich daraus eventuell ergebende bessere Schranke allen Tasks bekanntgegeben werden. Dieses Verfahren setzt sich fort, bis alle Knoten der Suchfront abgearbeitet wurden und somit das optimale Layout ermittelt wurde.

Die Portierbarkeit dieses Ansatzes

Besteht die virtuelle Maschine aus einem heterogenen Netz von Workstations, so verfügen die einzelnen Hosts zum einen über unterschiedliche Rechenleistungen, zum anderen sind sie je nach der Zahl der übrigen Prozesse und Benutzer unterschiedlich stark ausgelastet. Dieses wird bei der oben beschriebenen Parallelisierung berücksichtigt, da die einzelnen Tasks kleine Teilprobleme erhalten und nach deren Abarbeitung selbständig beim Master neue Arbeit erfragen. Tasks auf weniger starken oder mehr ausgelasteten Hosts stellen somit weniger Anfragen, da sie für die Suche in ihren Teilbäumen eine längere Zeit benötigen als diejenigen Tasks, welche auf leistungsfähigeren Rechnern laufen. In einem Workstation-Cluster kommunizieren alle Maschinen über einen gemeinsamen Bus, d.h., die Kommunikation zwischen allen Knoten ist gleich teuer, wobei die Kommunikationsbandbreite jedoch konstant und unabhängig von der Zahl der Maschinen ist. Aus diesem Grund ist ein Master-Slave Ansatz bei der Parallelisierung durchaus legitim.

In einem MPP sind die Prozessoren zwar alle gleich stark und stehen einem einzelnen Benutzer im Normalfall exklusiv zur Verfügung, jedoch ist die Zahl der Prozessoren hier viel höher. Auf Parallelrechnern ist die Kommunikation wesentlich schneller, und die Bandbreite steigt in der Regel mit der Größe des Systems. Da hier aber die Prozessoren in einer bestimmten Topologie untereinander verbunden sind, ist die Kommunikationsgeschwindigkeit abhängig von dem Abstand der kommunizierenden Prozessoren. Steht für die Kommunikation kein spezielles Routing-Netzwerk zur Verfügung, so werden die Nachrichten durch die Prozessoren hindurchgereicht, so daß jede Kommunikation die Rechenleistung der daran beteiligten Prozessoren hemmt. Für den oben beschriebenen Master-Slave Ansatz bedeutet dies, daß zum einen die Kosten für jede Kommunikation zwischen einer Slave-Task und dem Master unterschiedlich teuer sind, zum anderen verdichtet sich das Nachrichtenaufkommen in der Nähe desjenigen Prozessors, der die Master-Task hält (*Bottleneck*). Auf einem massiv-parallelen System sollte ein Lastverteilungsverfahren gewählt werden, welches die Last nur zwischen direkt benachbarten Prozessoren austauscht, um den Kommunikationsoverhead möglichst gering zu halten. PVM stellt hier jedoch keine weiteren Routinen zur Programmierung von MPPs zur Verfügung, etwa zur Bestimmung der Position der Tasks im parallelen System oder zur Ermittlung der TIDs von Tasks auf benachbarten Prozessoren, um für die Kommunikation kurze Wege nutzen zu können.

Die Architektur der auf dem Markt erhältlichen MPPs variiert stark, und jeder Hersteller bietet für sein System Programmierumgebungen an, welche direkt auf die Hardware zugeschnitten sind und somit die höchstmögliche Leistung erreichen. Eine PVM -Implementation für einen Parallelrechner setzt oft auf diese Hardware-spezifischen Programmierumgebungen auf und muß einen gemeinsamen Bus simulieren, was die Kommunikationsleistung auf dem MPP im Vergleich zur systemeigenen Programmierumgebung deutlich herabsetzt. Auch kann man, wenn man portabel programmieren will, viele systemspezifische Eigenschaften nicht ausnutzen. Dieses ist aber wichtig, wenn ein paralleles Programm skalierbar sein soll, d.h., wenn es auch große Systeme effizient nutzen soll.

Fazit und Alternativen

Es wurde gezeigt, daß es unter PVM einfach ist, ein vorhandenes heterogenes Netz von Workstations als einen virtuellen Parallelrechner zu nutzen. Aufgrund der weiten Verbreitung von PVM ist es möglich, die unter diesem System entwickelten parallelen Programme auch auf echte Parallelrechner und massiv-parallele Systeme zu portieren.

Wie an dem Beispiel eines parallelen *Branch & Bound*–Algorithmus jedoch deutlich wurde, stößt diese Portierbarkeit schnell auf Grenzen, wenn ein unregelmäßiges Problem gelöst werden soll und dabei dynamische Lastverteilung eine große Rolle spielt. Kommunikation und Lastverteilung können auf einem MPP nur effizient gestaltet werden, wenn möglichst alle Eigenschaften der Hardware transparent sind, was aber gegen die Konzeption von PVM spricht.

Eine interessante Alternative bietet sich hier durch das *Message Passing Interface* (MPI) an. MPI ist ein Standard für portable Parallelprogrammierung [4]. Er wurde im Mai 1994 endgültig von einem Gremium, an dem fast alle Parallelrechnerhersteller beteiligt waren, festgelegt. Er beschreibt etwa 120 Routinen, die eine MPI–Implementation auf einem parallelen System zur Verfügung stellen muß. Da dieser Standard aus der Welt der Parallelrechner entstanden ist, werden z.B. kein gemeinsamer Bus, sondern Prozessor-Topologien bei den Kommunikationsroutinen zugrundegelegt.

Für eine effiziente Implementation der MPI–Routinen auf einem bestimmten parallelen System ist der entsprechende Hardware-Hersteller verantwortlich, MPI legt lediglich die Schnittstelle fest. Es existiert mittlerweile auch schon eine Implementation für Workstation-Cluster [5]. Der Anwender hat so die Möglichkeit, ein paralleles Programm zu schreiben, was bei der Portierung die Eigenschaften der entsprechenden Parallelrechner möglichst optimal ausnutzt.

Bei all den Hilfestellungen, die die hier vorgestellten Systeme bieten, darf nicht vergessen werden, daß in erster Linie die eigentliche Parallelisierung eines Problems, welche nach wie vor der Anwender entwickeln muß, die Effizienz eines parallelen Programms bestimmt, inwieweit diese Parallelisierung auf der Hardware umgesetzt wird, ist nur zweitrangig.

Literatur

[1] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994, auch `ftp://netlib2.cs.utk.edu/pvm/book/pvm-book.ps`

[2] PVM-System, Sourcen für viele Architekturen, `ftp://elib.zib-berlin.de/pub/netlib/pvm/` oder `http://www.epm.ornl.gov:80/pvm`

[3] University of Knoxville in Tennessee, ParaGraph: Performance-Analyser, `ftp://elib.zib-berlin.de/pub/netlib/paragraph/`

[4] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, `ftp://info.mcs.anl.gov/pub/mpi/mpi-report.ps`

[5] Argonne National Laboratory, *mpich: MPI–Implementation für Unix-Workstations*, `ftp://info.mcs.anl.gov/pub/mpi/`