Parallel–Computing
als Hilfsmittel im technisch-wissenschaftlichen Höchstleistungsrechnen
– Eine Einführung –

Wolfgang W. Baumann
Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)

7. Februar 2000

Zusammenfassung

In dieser Einführung werden die Grundlagen der Architektur und Anwendung von Parallel–Computern dargestellt. Dabei wird zunächst auf die Hardware eingegangen, dann auf die Softwarekonzepte und schließlich auf konkrete Implementierungen und Beispiele.

Als Hardware–Beispiele werden Workstation-Cluster und der Parallelrechner CRAY T3E am Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB) behandelt.

Die Programmierung von verteilten Systemen wird am Beispiel der Message-Passing-Bibliotheken MPI und PVM beschrieben.

Vorwort

Diese Einführung basiert in Teilen auf dem Material eines Einführungskurses in die Parallele Programmierung unter Verwendung von MPI, der unter Beteiligung des Autors im Jahr 1999 an der Christian-Albrechts-Universität Kiel gehalten wurde.

Inhalt

1 Einführung

Bevor im Folgenden auf Architektur und Anwendung von Parallel–Computern eingegangen werden soll, muß man zuerst die Frage stellen: Wozu braucht man eigentlich Parallel–Computer? Warum reicht die scheinbar unbegrenzt steigerbare Leistung der “normalen” (sequentiellen) Computer nicht mehr aus?

Der Grund ist, daß die Anforderungen komplexer, moderner Anwendungen an die Rechenleistung enorm hoch geworden sind und auch weiter steigen werden. Beispiele hierfür sind Simulationen in der Strömungs- und Fahrzeugtechnik, die Bild- und Signalverarbeitung, Mustererkennung, und eine Vielzahl weiterer Problemlösungsverfahren aus den Bereichen der Natur- und Ingenieurwissenschaften.

Die Leistung der heute vorhandenen Rechnersysteme kann aber kaum noch oder nur unter immensem Entwicklungsaufwand signifikant gesteigert werden, da schon jetzt die dafür wichtigsten Faktoren — Packungsdichte und Taktfrequenz der Chips — nahezu ihre Grenzen erreicht haben, die in Materialeigenschaften und (verfahrens-)technischen Möglichkeiten begründet liegen. Daher hat seit einiger Zeit ein lange vernachlässigter Aspekt der Computer– Architektur rasant an Bedeutung gewonnen: die Parallel–Computer.

1.1 Historisches

Die Verarbeitungsgeschwindigkeit von Computern hat sich seit den 50ern alle fünf Jahre etwa verzehnfacht. Die erste Computer-Generation verwendete Relais mit einer Schaltzeit von etwa 1 μs. In den 60ern kamen die ersten Computer mit Transistoren auf, die Schaltzeiten wurden auf ca. 0.3 μs verrringert. Durch die Entwicklung integrierter Schaltungen wurde die Schaltzeit von 10 ns 1965 auf etwa 1 ns 1975 gesenkt. Von 1950 bis 1975 hatte sich die Geschwindigkeit der Komponenten, gemessen an der Schaltzeit, um etwa den Faktor 1000 erhöht. In der gleichen Zeit verbesserte sich die Leistungsfähigkeit der Systeme, gemessen in Fließkomma–Operationen pro Sekunde (FLOPS)[1], um einen Faktor näher bei 100.000.

Bei Computersystemen spricht man heute im Desktop-Bereich von GigaFLOPS (10^9 FLOPS), im High-End-Bereich von TeraFLOPS (10^{12} FLOPS). Das Ziel der aktuellen Entwicklung, angeführt vom amerikanischen *ASCI*–Programm *(American Strategic Computing Initiative)* ist ein Rechner mit 30 TeraFLOPS im Jahre 2001 und mit 100 TeraFLOPS im Jahre 2004 für die Bearbeitung von Problemen der obersten Leistungsklasse, den sogenannten *Grand Challenges*. Ein PetaFLOPS–Computer (10^{15} FLOPS) dürfte in der der zweiten Hälfte des kommenden Jahrzehnts zu erwarten sein. Solche Leistungen können nicht mit Einzelprozessoren sondern nur von parallelen Architekturen mit einer großen Anzahl von Prozessoren erreicht werden.

1.2 Von–Neumann goes parallel

Die große Masse der heute eingesetzten Computersysteme basiert auf der sogenannten Von-Neumann-Architektur. Dabei gibt es einen Prozessor, der für die Ausführung von Befehlen zuständig und mit den Peripheriegeräten sowie mit dem Speicher über einen Bus verbunden ist (siehe Abb. 1).
Im Laufe der Programmausführung werden nun die im Speicher befindlichen Befehle an den Prozessor übertragen, der sie ausführt und danach den nächsten Befehl erhält. Zu einem gegebenen Zeitpunkt wird also immer nur ein Befehl abgearbeitet.

Das ändert sich auch nicht, wenn sog. Multi–Tasking–Systeme eingesetzt werden, die ein scheinbar gleichzeitiges Ausführen verschiedener Programme ermöglichen. Dabei wird lediglich jedem Programm ein gewisser Anteil an der Rechenzeit zugeteilt, während die anderen Programme warten müssen. Um (Teil-)Programme tatsächlich parallel ablaufen zu lassen, werden mehrere Prozessoren benötigt, die mehr oder weniger unabhängig voneinander sind und Befehle real gleichzeitig ausführen können.

Die Vorteile, die dadurch erreicht werden sollen, sind:

[1] Ein Glossar mit wichtigen Begriffen findet man in WWW unter `http://nhse.npac.syr.edu/hpccgloss/`

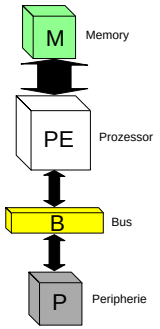


Abbildung 1: Schema eines von-Neumann-Rechners

- **Hohe Rechenleistung.** Durch das parallele Ausführen mehrerer (Teil-) Aufgaben wird die Gesamtausführungszeit verkürzt.
- **Leichte Erweiterbarkeit und Anpaßbarkeit.** Durch späteres Hinzufügen weiterer Komponenten kann man die Leistungsfähigkeit erhöhen, um größere Aufgaben zu bewältigen. Außerdem kann das System durch Austausch der Komponenten dem technischen Fortschritt Rechnung tragen.
- **Hohe Sicherheit.** Das Vorhandensein mehrerer Komponenten bedeutet, daß bei Ausfall einiger davon die anderen immer noch benutzbar bleiben (evtl. mit verminderter Leistung).
- **Gutes Preis–/Leistungsverhältnis.** Da die Rechenleistung der einzelnen Komponenten addierbar sein sollte, ist es nicht unbedingt notwendig, die höchstentwickelten (und damit auch teuersten) Technologien für die einzelnen Komponenten zu verwenden, um wirklich hohe Leistung zu erhalten.

1.3 Ziel des parallelen Rechnens

Das wesentliche Ziel des parallelen Rechnens ist es, das gleiche Ergebnis wie auf einem Einzelprozessor zu erhalten. Das zu lösende Problem wird dabei auf N_P Prozessoren aufgeteilt, und im Idealfall läuft das Programm dann N_P-mal schneller als auf einem Prozessor. Man spricht dann von idealer *Skalierbarkeit*. In der Praxis ist das kaum erreichbar wegen des unvermeidlichen Overheads beispielsweise für Kommunikation und wegen rein sequentiell ablaufender Teile wie der Zugriff auf Dateien. Die sinnvolle und effiziente Aufteilung eines Problems ist eine entscheidende Voraussetzung für gute Performance auf dem Parallelrechner. Es sollte möglichst sichergestellt sein, daß jeder Prozessor gleich viel Arbeit tut, dann ist das Problem *last-balanciert*. Außerdem sollte sichergestellt sein, daß jeder Prozessor soviel Zeit wie möglich mit Berechnungen und möglichst wenig mit Kommunikation beschäftigt ist.

2 Grundlagen paraller Architekturen

Im Bereich der parallelen Computersysteme gibt es unterschiedliche Konzepte. Wir betrachten hier Parallelrechner, die durch Zusammenfügen mehrerer gleich- oder andersartiger Prozessoren[2] entstehen. Der Vollständigkeit halber werfen wir auch noch einen kurzen Blick auf prozessor-interne Parallelität (Abschnitt 2.1.3).

[2] Wir sprechen hier und im Folgenden auch von den *Prozessor-Elementen*, kurz *PEs* eines Parallelrechners.

2.1 Klassifizierung

Eine weitverbreitete Klassifikation paralleler Architekturen geht auf *Flynn* zurück, der eine Einteilung anhand von Daten- und Befehlsströmen macht (Abb. 2). Nach dieser Klassifikation wird ein konventioneller serieller Rechner als *Single Instruction Single Data (SISD)* klassifiziert, da der Prozessor in jedem Schritt *einen* Befehl auf *einem* Datenelement ausführt. Die wichtigsten Parallelrechnertypen werden im Folgenden besprochen.

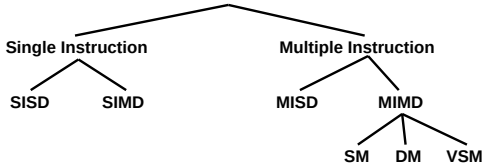


Abbildung 2: Klassifikation von Rechnerarchitekturen nach Flynn

2.1.1 SIMD

Beim *SIMD–Konzept (Single Instruction Multiple Data)* führen alle beteiligten Prozessoren jeweils die gleichen Anweisungen aus, jedoch auf unterschiedlichen Daten. Das entsprechende Programmierkonzept (*Data–Parallel–Programming*) ermöglicht z.B. die parallele Transformation eines Arrays nach einem für alle Elemente gleichen Algorithmus. Da hierbei aber nur eine sehr begrenzte Art von Parallelität möglich ist, konzentriert sich die Entwicklung auf das *MIMD*–Konzept.

2.1.2 MIMD

Beim MIMD–Konzept ((Multiple Instruction Multiple Data) sind die Prozessoren völlig unabhängig voneinander, d.h. jeder Prozessor führt sein eigenes Programm auf jeweils unterschiedlichen Daten aus. Damit die Prozessoren sinnvoll zusammenarbeiten können, müssen sie untereinander Daten austauschen. Dies geschieht durch Speicherzugriffe. Üblicherweise unterteilt man die MIMD-Architekturen nach ihrer Relation zwischen Prozessoren und Speicher. In Abschnitt 2.2 werden die drei Haupttypen der MIMD-Architekturen anhand der Speichermodelle beschrieben.

Eine einschränkende Variante des allgemeinen MIMD-Konzepts ist das *SPMD-Konzept (Same Program Multiple Data)*. Dabei führen alle Prozessoren das gleich Programm auf unterschiedlichen Daten aus. Dies ist in aller Regel auch die naheliegendste Vorgehensweise, da sich ein Problem meist auf natürliche Weise in Teilprobleme zerlegen läßt. Beispielsweise kann ein räumliches Gitter für ein physikalisches Problem durch Blockzerlegung in kleinere Blöcke zerlegt werden. Die Blöcke werden auf die PEs verteilt und das Problem wird auf jedem Teilgitter gelöst.

2.1.3 Prozessor-interne Parallelität

An dieser Stelle soll noch der Aspekt der prozessor-internen Parallelität erwähnt werden. Moderne CPUs haben mehrere Funktionseinheiten (FE), z.B. für Befehlsentschlüsselung, Fließkomma-Addition (FP-Add), Fließkomma-Multiplikation (FP-Mult), Integer-Verarbeitung (Int), logische Befehle (Logic), etc., die parallel auf Instruktionsebene arbeiten können.

Werden die verschiedenen Stadien der Abarbeitung *eines* Befehls auf verschiedene Funktionseinheiten verteilt, die dann mehrere dieser Befehle zeitlich verschränkt bearbeiten, so spricht man von *Pipelining*. Damit erreicht man, daß bei voller Ausnutzung in jedem Taktzyklus ein Ergebnis produziert werden kann, obwohl ein einzelner Befehl gewöhnlich mehrere Takte für die Ausführung benötigt.

Sind einzelne Funktionseinheiten mehrfach vorhanden, oder können mehrere Befehle ohne Interferenz gleichzeitig bearbeitet werden, z.B. FP-Add und FP-Mult, so kann der Prozessor in diesen Fällen pro Taktzyklus mehr als ein (Fließkomma-)Ergebnis liefern. Solche CPUs bezeichnet man als *superskalar*.

2.2 Speichermodelle

2.2.1 Gemeinsamer Speicher

Die einfachste und vielleicht einleuchtendste Möglichkeit auf gemeinsame Daten zuzugreifen ist ein gemeinsamer Speicher (*Shared–Memory (SM)*). Bei dieser “Erweiterung” des Von–Neumann–Modells

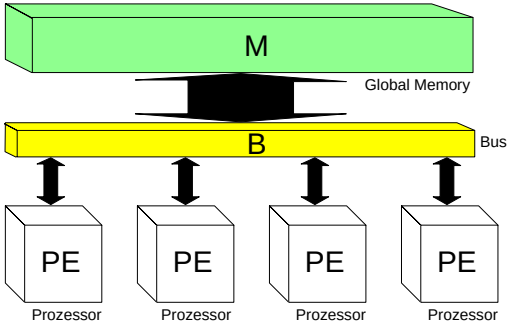


Abbildung 3: Schema eines Shared-Memory-Rechners

sind alle Prozessoren an den gemeinsamen Bus angeschlossen, über den sie auf den gemeinsamen Speicher zugreifen, Abb. 3. Die Inter-Prozessor-Kommunikation erfolgt dadurch, daß ein Prozessor Daten in einen Speicherbereich schreibt und ein anderer Prozessor diese liest. Die Programmierung für diese Architektur ist einfach, da es keine explizite Kommunikation gibt. Zu einer bestimmten Zeit darf aber immer nur ein Prozessor auf den Bus zugreifen, weil es sonst zu Überlagerungen kommt. Dadurch entstehen zwei Probleme:

- ☞ Einem Prozessor darf der Zugriff nur dann erlaubt werden, wenn der Bus frei ist. Dieses kann durch entsprechende Mechanismen innerhalb der Programmierkonzepte garantiert werden.
- ☞ Es kommt zu Verzögerungen, weil ein Prozessor auf einen Speicherzugriff warten muß. Mit steigender Anzahl von Prozessoren verschärft sich dieses Problem erheblich, so daß ab einer gewissen Stufe die langen Wartezeiten den Vorteil der Parallelität zunichte machen.

Einen Lösungsansatz bietet der *Cross-Bar-Switch*, der jedem Prozessor auf jede Speichereinheit Zugriff über einen eigenen Pfad bietet. Da jedoch die Komplexität dieser Schaltung mit zunehmender Prozessoranzahl und Speichergröße immens zunimmt (und damit auch die Kosten) sind diesem Weg technische und finanzielle Grenzen gesetzt. Dennoch bietet diese Technik für hinreichend kleine Systeme große Vorteile, da der Datenaustausch praktisch genauso (schnell) wie ein gewöhnlicher Speicherzugriff vor sich geht.

2.2.2 Verteilter Speicher

Um das Synchronisationproblem zu entschärfen, kann man den Speicher verteilen und jedem Prozessor seinen eigenen Speicher angliedern. Dabei kann in der Regel jeder Prozessor nur seinen eigenen Speicher direkt adressieren (*Distributed Memory Concept (DM)*). Wenn der Speicher zwar physikalisch verteilt ist, es aber einen logisch globalen Adreßraum gibt, so spricht man von einem *Virtual Shared*

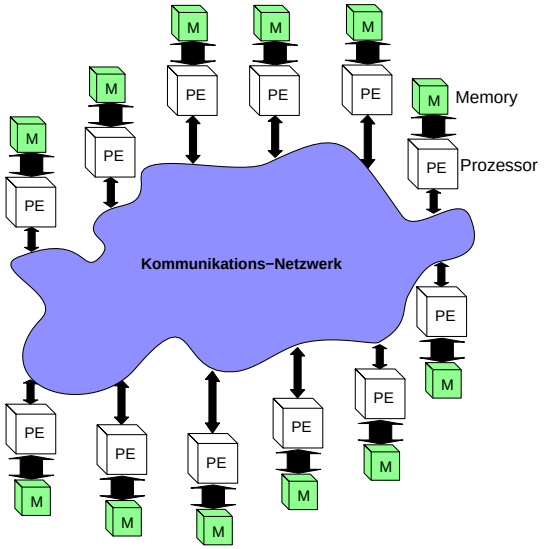


Abbildung 4: Schema eines Distributed-Memory-Rechners

Memory (VSM). Der Zugriff auf entfernten Speicher wird ermöglicht durch unterstützende Hardware, die die Kommunikation unabhängig vom entfernten Prozessor durchführt.
Die Basiseinheit eines Distributed-Memory-Systems ist – vereinfacht gesagt – ein Prozessor mit seinem lokalen Speicher. Diese kann man praktisch beliebig zusammensetzen und durch ein Kommunikationsnetzwerk verbinden (Abb. 4) und erhält so einen Parallel-Computer mit hoher Rechenleistung aus relativ preisgünstigen Einzelkomponenten.
Für den Datenaustausch wird ein *Message–Passing–System* eingeführt, welches das Versenden von Nachrichtenpaketen über entsprechende Verbindungen zwischen den Prozessoren ermöglicht. Dieses wird genutzt, wenn ein Prozessor Daten aus dem Speicher eines anderen benötigt. Dabei muß eine Nachricht an den anderen Prozessor geschickt werden, in der eine Kopie der benötigten Daten angefordert wird. Message-Passing muß programmiert werden und erfordert in der Regel einen nicht zu vernachlässigenden Aufwand, siehe Kapitel 6.

Das MIMD-Konzept mit verteiltem Speicher ist auf verschiedene Arten umgesetzt worden, von denen zwei im Rahmen dieser Übersicht näher behandelt werden: Massiv-Parallelrechner (MPP) und Workstation-Cluster (NOW), siehe hierzu Abschnitt 3.

2.2.3 Hybrid-Lösungen

Die Kombination von leicht zu programmierenden Shared-Memory-Rechnern mit der Skalierbarkeit von Distributed-Memory-Architekturen wird von allen namhaften Herstellern in Form von SMP-Clustern realisiert. Hierbei werden Shared-Memory-Knoten (Mehrprozessor-Maschinen mit 16-128 PEs und gemeinsamem Speicher) durch Hochgeschwindigkeits-Netze verbunden. Auf dem Knoten kann Shared-Memory programmiert werden, zwischen den Knoten muß in der Regel Message-Passing verwendet werden. Solche Lösungen werden auch als *NUMA (Non-Uniform Memory Access)*–Architekturen bezeichnet, da der Zugriff auf den gemeinsamen lokalen Speicher viel schneller erfolgen kann als der Zugriff auf den Speicher eines anderen Knotens.

Moderne (RISC-)Prozessoren besitzen Speicherhierarchien mit meist mehrstufigen Caches. Architekturen, die solche Prozessoren verwenden, müssen dafür Sorge tragen, daß Memory- und Cache-Inhalte bei Änderungen von Speicherelementen aktualisiert und konsistent gehalten werden, auch über Knotengrenzen hinweg. Architekturen, die unterstützende Hardware dafür beinhalten, bezeichnet man als *ccNUMA (cache-coherent NUMA)*.

2.3 Weitere Komponenten

2.3.1 Kommunikations-Netzwerk

Es ist klar, daß lokale Speicherzugriffe viel schneller erfolgen können als die bei parallelen Anwendungen unvermeidlichen Zugriffe auf entfernte Speicher, die damit auch zwangsläufig Kommunikation erfordern. Oftmals läßt sich Parallelität nur auf feingranularer Ebene nutzen und erfordert deshalb entsprechend viel Kommunikation für die Synchronisation und den Datenaustausch. Die Synchronisationsgeschwindigkeit wird dabei durch die Netzwerk-*Latenz* bestimmt, die Schnelligkeit des Datenaustauschs hängt von der *Bandbreite* ab.

Der Kommunikationsaufwand auf einer Renderfarm für Computerfilme bewegt sich zum Beispiel am unteren Ende der Skala der Kommunikations-Anforderungen, wogegen sich Finite-Elemente-Rechnungen und CFD-Probleme am oberen Ende bewegen mit ihrem Bedarf an hohen Bandbreiten und niedrigen Latenzen. Die Kombination dieser Faktoren ist das Verhältnis von Kommunikationszeit zu Rechenzeit. Die Bandbreite und Verzögerungen in der Synchronisation werden weniger signifikant, wenn die Rechenzeit zwischen den Kommunikationsschritten lang ist (siehe auch *Granularität* in Abschnitt 4.2.3).

Eine wesentliche Voraussetzung für eine leistungsfähige Parallelarchitektur ist daher ein effizientes Verbindungsnetzwerk mit geringen Latenzzeiten und großer Bandbreite. Die gesamte Kommunikationsbandbreite des Netzwerkes hängt ab von der Verbindungstopologie des Netzes und von der Ausgeklügeltheit des Routing-Algorithmus (Auf welchem Weg durch das Netz werden Pakete von **A** nach **B** geschickt?). Bei heterogenen Parallel-Rechnern (z.B. bei Clusterlösungen) wird als Verbindungsnetz meist das normale Daten-Netz (Ethernet, Fast Ethernet, Gigabit Ethernet, ATM, ...) benutzt, ggfs. im Zusammenhang mit einem Switch. Verbesserte Netzperformance erhält man bei Verwendung spezieller Netzadapter wie z.B. Myrinet oder SCI, die eine wesentlich kleinere Latenzzeit besitzen. Homogene Parallel-Rechner haben meist proprietäre Verbindungsnetze.

Netzwerk-Topologien Das Verbindungsnetzwerk ist auch umso leistungsfähiger, je mehr Knoten direkt miteinander verbunden sind. Der damit verbundenen Hardwareaufwand wird jedoch oft nicht ausgenutzt. Bevorzugte Netztopologien, insbesondere bei hochskalierenden homogenen Parallel-Architekturen sind heute Gittertopologien. Einfache Gitter wie 2D-Gitter oder Würfel kommen aber kaum zum Einsatz.

Torus-Struktur Verbindet man die einander gegenüberliegenden Ränder bzw. Flächen eines 2D-Gitters bzw. Würfels direkt miteinander, so erhält man Torus-Topologien, siehe Abb. 5. Ein Torus stellt kurze Verbindungswege bei hoher Bandbreite sicher. Er ist außerdem flexibel bezüglich der Erweiterbarkeit des Gesamtsystems, wobei die Leistungsfähigkeit des Netzes nicht wesentlich beeinträchtigt wird.

(Hyper–)Würfel–Struktur Allgemein sind Hyperwürfel–Strukturen nichts anderes als Projektionen von Würfeln höherer Dimension als der dritten in das Dreidimensionale. Als Beispiel sei hier nur der vierdimensionale Hyperwürfel genannt (siehe Abb. 6). Er besteht aus 16 Prozessoren mit jeweils vier Verbindungen und zeichnet sich durch besonders kurze Wege im Gesamtnetz aus. Hyper-Würfel sind aber weniger flexibel beim Ausbau eines Systems, weil Verbindungen ungenutzt bleiben, wenn der Würfel nicht voll ausgebaut ist.

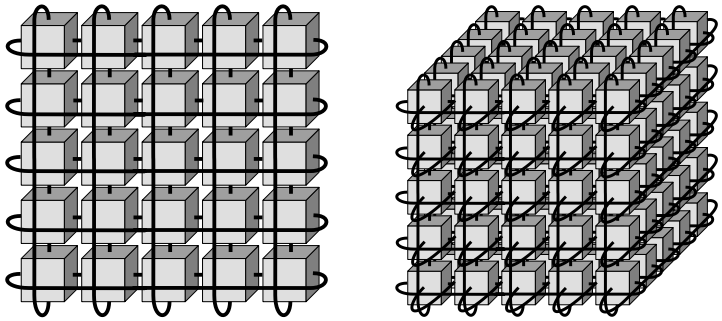


Abbildung 5: Schema eines 2-dimensionalen und eines 3-dimensionalen Torus

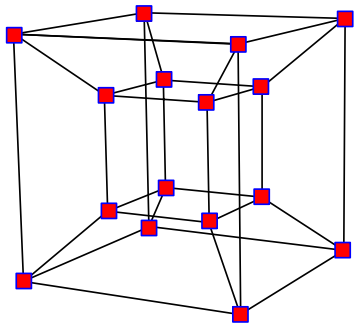


Abbildung 6: Vierdimensionale Hyperwürfel–Struktur

2.3.2 Barrieren-Netzwerk

Für den geordneten Ablauf von parallelen MIMD-Anwendungen ist es notwendig, daß sich die beteiligten Prozesse synchronisieren können, d.h alle müssen an einem Synchronisationspunkt (eine sogenannte *Barrier*) angekommen sein. Neben der logischen UND-Verknüpfung als Barriere im eigentlichen Sinne ist auch die ODER-Verknüpfung hilfreich. Dieser *Eureka*-Modus eignet sich z.B. für das parallele Suchen: sobald einer der Prozesse fündig geworden ist, kann er dies den anderen mitteilen und diese können dann die Suche sofort beenden.

Ein hardware-unterstütztes Barrieren-Netzwerk gestattet sehr schnelle und effiziente Synchronisation und ist besonders bei feingranularer Parallelität der Anwendung von erheblichem Vorteil. Barrieren-Netzwerke findet man in einigen homogenen Massiv-Parallelarchitekturen wie z.B. der Cray T3E, siehe Abschnitt 3.2

3 Beispiel-Architekturen

Im Rahmen dieser Einführung soll als Beispiel für die Bandbreite der verfügbaren Architekturen von Massivparallel-Rechnern auf zwei recht unterschiedliche Systeme eingegangen werden: eine PC- oder Workstation-Farm und das MPP-System CRAY T3E. Diese beiden Systeme unterscheiden sich in vier Hauptbereichen, die durch die Stichworte geringe Latenz und hohe Bandbreite, Single-System-Image Betriebssystem, Programmiermodell sowie I/O-Fähigkeiten charakterisiert sind.

3.1 Cluster-SystemeNOW

Es gibt seit einiger Zeit intensive und erfolgreiche Bestrebungen, die Leistungsfähigkeit bereits vorhandener Netzwerke aus Workstations zu einem Parallel–Computer zu vereinen. Bekannte Bezeichnungen für solche Cluster-Systeme sind *Loosely Coupled Systems* oder *Network of Workstations (NOW)*. Die Workstations können gleiche, ähnliche oder auch unterschiedliche Systeme haben. Der gängigste Fall ist das Zusammenschalten von Rechnern eines Pools aus z.B. Linux-PCs oder Workstations des gleichen Systemherstellers.

Jeder Rechenknoten hat sein eigenes (Betriebs-)System, seine eigenen Filesysteme sowie andere eigene Ressourcen. Für den Parallelbetrieb solche eines Clusters existieren spezielle Message Passing Bibliotheken, die Funktionen zur Nachrichtenübermittlung im Netzwerk zur Verfügung stellen (siehe Abschnitt 6). Dies hat den großen Vorteil, daß Benutzer solcher Netzwerke praktisch ohne zusätzliche Kosten die Vorteile der Parallelität nutzen können, da die Message Passing Software meist *Public Domain* ist.

Vorteile:

- Keine Extra-Investitionen, weil die Maschinen sowohl als Stand-alone Arbeitsplätze als auch als Parallelrechner betrieben werden können.
- Flexible Ausbaumöglichkeit, weil weitere Rechner einfach und beliebig in das vorhandene Netz integriert werden können.
- Hoche Ausfallsicherheit des Gesamtsystems.
- Kurze Latenzzeit des einzelnen Prozesses zu seinem Betriebssystem.
- PE-lokaler I/O ist schnell, da direkter Weg zur lokalen Platte.

Nachteile:

- Hoher Speicherplatzbedarf, da das Betriebssystem repliziert wird.
- Schwierig zu administrieren, da jeder Prozessor sein eigenes Rechnersystem darstellt.
- Schwieriges Prozeß-Management für Anwendungen mit großer PE-Anzahl; führt zu schlechter Skalierung bei mehr als (rund) 10 PEs.
- Globales Scheduling erfordert zusätzliche Mechanismen.
- Kein Single-System-Image: gemeinsame Filesysteme nur über NFS oder ähnliches.
- Große Latenzzeiten bei oft mäßigen Bandbreiten der Standard-Netzwerk-Interfaces führen zu Performance-Problemen bei feingranularer Parallelisierung.

3.2 Der Massiv-Parallelrechner CRAY T3E

In diesem Abschnitt soll auf den am Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB) installierten Massiv-Parallelrechner CRAY T3E eingegangen werden, der insbesondere für Berliner Projekte mit Höchstleistungsrechenbedarf zur Verfügung steht und unter anderem von allen Berliner Universitäten genutzt wird. Das System wird etwas ausführlicher beschrieben, da seine Architektur weniger bekannt und geläufig ist als die Architektur eines gängigen PCs.

3.2.1 Hardware

Der CRAY T3E ist ein MIMD-Rechner mit SPMD-Programmiermodell. Er stellt einen gemeinsamen physikalischen Adressbereich von bis zu 2048 Prozessoren bereit, die über ein 3D-Torus-Netzwerk (Bild 5) miteinander verbunden sind. Das System ist in Einheiten von 4 bzw. 8 Knoten skalierbar. Jeder Knoten des Systems besteht aus einem Alpha 21164 Prozessor, einem System-Control-Chip, lokalem Speicher bis 2 GB und einem Netzwerk-Router.

Der super-skalare Alpha-Prozessor arbeitet je nach Modell mit 300, 450 oder 600 MHz in 64-Bit-IEEE-Arithmetik und kann pro Taktzyklus vier Instruktionen ausführen, von denen je eine ein *FADD* und ein *FMULT* sein kann. Daraus resultieren je nach Taktrate 600 bis 1200 MFlops. Die CPU besitzt je einen 8 kB L1-Cache für Daten und für Instruktionen und einen 96 kB L2 Cache. Cacheing erfolgt nur für lokale Daten, die Cache-Kohärenz ist hardware-unterstützt. Zur Verbesserung der lokalen Speicherbandbreite gibt es anstelle eines externen Caches sogenannte *Stream-Buffers*, die Vektor-Zugriffe erkennen und durch *Read-Ahead* beschleunigen. Ein T3E-Knoten besitzt außerdem einen großen Satz externer sogenannter *E-Register*, über die entfernte Kommunikation und Synchronisierung abgewickelt werden.

Die Torus-Verbindungen liefern eine Bandbreite von brutto 600 MB/s in jede Richtung, womit sich eine Durchsatzleistung von knapp 3 GB/s pro Netzwerk-Knoten ergibt. Die Latenzzeit des Verbindungsnetzwerks ist nominell 1μs, wie sie von der CRAY-proprietären *Shared Memory Access (shmem)*-Library auch erreicht wird. Die effektive Latenzzeit für andere Message-Passing-Bibliotheken (PVM, MPI) ist eine Größenordnung höher wegen des Overheads, der unter anderem mit Pufferung und Deadlock-Erkennung verbunden ist (siehe hierzu auch Kapitel 6).

Die Ein-/Ausgabe geschieht für je 4 Rechenknoten über einen I/O-Kontrollknoten und das *GigaRing*-Subsystem (basiert auf dem IEEE-SCI-Standard) mit einer Bandbreite von 2GB/s (brutto; netto maximal 1.4 GB/s).

Das T3E-System ist *self-hosted* (kann ohne Front-End-Rechner betrieben werden), neben den eigentlichen *Applikations-PEs* sind zusätzliche *Support-PEs* vorgesehen.

3.2.2 Betriebssystem UNICOS/mk

Das Betriebssystem *UNICOS/mk* realisiert ein verteiltes *CHORUS*-Microkernel basiertes und damit skalierbares System. Es bietet dem Benutzer, der Applikation und der Administration eine einheitliche Sicht des Gesamtsystems (*Single-System-Image*) wie es von jeder UNIX-Workstation bekannt ist. Auf jedem Knoten läuft ein kompakter Mikrokernel für die wichtigsten Basisfunktionen (z.B. Speicherverwaltung, Basis-I/O, Interprozeßkommunikation etc.). Alle anderen Betriebssystemfunktionen sind modular aufsogen. Serverprozesse verteilt, die auf den oben erwähnten Support-PEs laufen, diese nennt man *Operating System-PEs (OS-PEs)*. Ein Teil der Server-Knoten ist als sogenannten *Command-PEs* außerdem für interaktives Arbeiten vorgesehen. Der überwiegende Teil der PEs steht für die Abarbeitung paralleler Applikationen zur Verfügung (Applikations-PEs). Aus Hardware-Sicht gibt es keine Unterscheidung zwischen den PE-Klassen (APP, OS, CMD), die Zuordnung der PEs zu den Klassen geschieht nach Konfiguration beim Boot des Systems. Redundante PEs (*Hot-Spares*) übernehmen bei Bedarf die Aufgaben ausfallender PEs. Das Single-System-Image hat den Vorteil, daß man es nur mit **einem** Betriebssystem zu tun hat, wenn es um Upgrade, Überwachung, Boot und Tuning geht.

Speicher- und Prozeßverwaltung Anstatt virtueller Speicherverwaltung und Paging ist in UNICOS/mk Swapping implementiert. Das bedeutet, daß kein Prozeß größer als der physikalisch auf dem PE vorhandene Speicher sein darf. Diese leichte Einschränkung ist angesichts des System- und I/O-Overheads bei virtueller Speicherverwaltung zugunsten von Performance-Gesichtspunkten tolerierbar. Prozesse können (auch als Gruppe) innerhalb des Netzes verschoben und auf Platte ausgelagert (*Checkpoint*) werden.

Datei-Systeme Es gibt nur gemeinsame, globale, aber keine PE-lokalen Filesysteme (*Shared File Systems*.

3.2.3 Programmierung

Die Hardwareunterstützung von Synchronisation (Barrier, Eureka) und Austausch (Fetch and Increment, Atomic Swap) gestattet extrem feingranulare Parallelisierung. Unterstützte Sprachen sind Fortran90, C und C++, für die Message-Passing verfügbar ist (MPI, PVM). Wegen des gemeinsamen Adreßraums werden auch Shared-Memory-Programmier-Modelle wie in HPF unterstützt. Eine Sonderstellung nimmt die *Shared Memory Access Library (shmem)* ein, die den Datenaustausch unter unmittelbarer Ausnutzung der Shared Memory Hardware ermöglicht. Diese Routinen arbeiten ohne Zwischenpuffer direkt zwischen lokalem und entferntem Speicher, ohne den entfernten Porzessor zu involvieren (sogen. *einseitige Kommunikation*).

3.2.4 T3E: Versuch einer Wertung

Supercomputing stellt nicht nur Maximal-Anforderungen an die Prozessoren sondern auch an Arbeitsspeicher, Plattenspeicher, Netzwerk, Archivspeicher, etc. und deren jeweils bestimmende Leistungsgrößen (Größe, Bandbreite, Schnelligkeit, Latenz, usw.) sowie an die (System-)Software. Eine der wesentlichen Eigenschaften der T3E-Architektur ist die Balanciertheit in allen Ausbaustufen, also die gleichermaßen starke Skalierbarkeit aller Systemkomponenten bei ausgewogener Gesamtleistung. Diese Eigenschaften sind nicht durch einfache Kombination von Standardkomponenten erreichbar, sondern sie erfordern eine Reihe von proprietären Hardware- und Systementwicklungen. Damit ist ein solches System, trotz Verwendung von Standardbausteinen wie z.B. RISC Microprozessoren und CMOS DRAM Speicherchips nicht für den Preis einer entsprechenden Anhäufung von Standard-PCs machbar: Spitzenleistung hat eben ihren Preis.

3.3 Abschließende Bemerkung

Neben den beiden erwähnten Beispielsystemen gibt es noch eine Vielzahl weiterer Architekturen, die jedoch den Rahmen dieser Einführung sprengen würde. Zu dieser Vielfalt von Architekturen kommt eine ebenfalls große Zahl verschiedener Wege der Programmierung. Wir werden auf die wichtigsten Programmierkonzepte im nächsten Kapitel eingehen, wobei der Schwerpunkt auf Systemen mit verteiltem Speicher liegen wird.

4 Konzepte der parallelen Programmierung

Es wird einleuchtend, daß die neuartige Architektur von Parallel–Computern auch ein neue Konzepte für deren Programmierung erfordert. Hier sind im Laufe der Zeit eine Reihe von Ansätzen gemacht worden, die auch und vor allem den sich rasant ändernden Bedingungen Rechnung tragen, unter denen sich die Parallel–Computer–Forschung entwickelt hat.

Die historische Entwicklung hat dabei wieder einmal bestätigt, daß sich die vom Software-Engineering her bekannten Konsequenzen beim Entwurf und der Codierung von Programmen bezüglich Effizienz und Portabilität auch und besonders hier ergeben. Portabilität wird meist erkauft durch einen Verlust an Effizienz, während optimale Performance in aller Regel nur durch herstellerspezifische Hilfsmittel zu erreichen ist, und den Code nicht mehr portabel macht.

Zu einer Zeit, in der es nur eine sehr kleine, hochspezialisierte und wenig beachtete Menge von Parallel–Forschern gab, wurde für jede neue Architektur eine Sprache entwickelt, die speziell darauf zugeschnitten war. Bekannt geworden ist hier vor allem die Sprache OCCAM für die sich später immer mehr verbreitenden transputerbasierten Systeme.

Es stellte sich aber bald heraus, daß es zwei große Probleme gab, die den Vorstoß der Parallel–Welt in den “main stream” der Computer–Entwicklung verhinderten :

- ☞ Der Programmierer mußte für jeden Rechner eine neue Sprache erlernen.
- ☞ Die Programme waren sehr beschränkt bzw. gar nicht portabel.

Um den Umstieg auf die parallele Programmierung zu erleichtern, wurden nun die notwendigen parallelen Konstrukte in herkömmliche Programmiersprachen (Fortran, C, Pascal, ...) eingebunden. Die meisten dieser Compiler waren allerdings wiederum auf einzelne Maschinentypen spezialisiert, die Programme daher nicht portabel.

In den ausgehenden 80er Jahren strebte man schließlich auf eine Lösung zu, die eine weitgehende Portabilität erlauben sollte: die sogenannten *Message Passing Libraries* realisieren die parallelen Konzepte flexibel auf jeder Architektur so, daß die jeweils spezifischen Möglichkeiten gut ausgenutzt werden. Dabei wird oft auch ein heterogenes Netzwerk aus verschiedenen Architekturen unterstützt. Dem Programmierer hingegen werden standardisierte Schnittstellen angeboten, die architektur–unabhängig sind und in verschiedenen “Wirtssprachen” (meist C und FORTRAN) eingebaut werden können. Auf diese Systeme wird später noch näher eingegangen, da sie eine zentrale Bedeutung erlangt haben.

Im Jahre 1992 wurde aber auch die Idee der in die Sprache integrierten Parallel–Konstrukte wiederaufgegriffen. Auch hier sollte eine allgemeine Programmierer–Schnittstelle für möglichst viele Architekturen geschaffen werden. Eine große Vereinigung von sowohl Anbietern als auch Anwendern, das *High Performance Fortran Forum (HPFF)* erarbeitete einen Standard für eine parallele Erweiterung von Fortran: das *High Performance Fortran (HPF)*, siehe hierzu auch Kap. 5.1. Seit seiner Verabschiedung 1993 haben einige Firmen Compiler herausgebracht, die für die meisten Plattformen verfügbar sind.

Es bleibt jedoch festzustellen, daß selbst mit so flexiblen Systemen wie sie gerade erwähnt wurden, **ein** Problem nicht gelöst werden kann. Diese “Achillesferse” des Parallelkonzeptes ist die Programmentwicklung selbst. Sie erweist sich nämlich als erheblich aufwendiger und schwerer abschätzbar als die gewohnte Erstellung eines sequentiellen Programms. Die Gründe dafür sind vielfältig:

- ☞ Es tauchen **neue Fehlerarten** auf, z.B. Synchronisations-/Kommunikations-Fehler.
- ☞ Das **Verhalten** paralleler Programme ist **schwerer nachzuvollziehen** und oft **nicht reproduzierbar**.
- ☞ Die **Aufteilung des Problems** und die **Zuordnung der Prozesse zu den Prozessoren** haben einen sigifikanten Einfluss auf die *Performance* des Programms.

Es muß daher vor der Entscheidung zur parallelen Implementierung eines Problems sorgfältig geprüft werden, ob der erhoffte Geschwindigkeitsvorteil den höheren Aufwand lohnt. Für Probleme ab einer bestimmten Größenklasse, insbesondere für Grand-Challenge-Probleme, ist eine parallele Implementierung unerläßlich, da die Anforderungen (nicht nur CPU-Leistung, sondern auch Arbeitsspeicher etc.) nur von Massivparallel-Rechnern befriedigt werden können.

4.1 Hauptfragen

Parallelisierung bedeutet ja in erster Linie die Aufteilung von (Rechen-)Arbeit auf mehrere gleichzeitig arbeitende Prozessoren, um das selbe Ergebnis zu erzielen wie mit einem Prozessor. Dabei stellen sich eine Reihe von Fragen, deren Antworten entscheidenden Einfluß auf die parallele Implementierung haben.

- **?** Kann das Problem in Teilaufgaben zerlegt werden, die ohne Datenzugriffskonflikte bearbeitet werden können?
- **?** Erkennung von Datenzugriffskonflikten: Wird in einer Gruppe von Anweisungen auf bestimmte Daten lesend *und* schreibend zugegriffen?

- **?** Können die Datenzugriffskonflikte durch Änderung der Ausführungsreihenfolge aufgelöst werden?
- **?** Ist die Änderung der Ausführungsreihenfolge algorithmisch zulässig?

4.2 Ansätze zur Parallelisierung

Die erste Frage, die sich stellt ist

- **?** Kann das Problem in Teilaufgaben zerlegt werden, die ohne Datenzugriffskonflikte bearbeitet werden können?

Wenn dies **nicht** möglich ist, so liegt der allgemeine Fall vor, der im Zusammenhang mit Datenzerlegung (Abschnitt 4.2.3) besprochen wird. Kann die obige Frage mit “ja” beantwortet werden, so ist zu prüfen:

- **?** Sind die unabhängigen Teilaufgaben gleich?

Wenn ja, dann liegt der einfachste Fall von Parallelität vor, die *triviale Parallelität*. Wenn die unabhängigen Teilaufgaben unterschiedlich sind, so liegt *funktionale Parallelität* vor.

4.2.1 Triviale Parallelisierung

Beispiele für triviale Parallelität sind Parameterstudien mit einem Programm oder die Erzeugung von Bildern für einen Film, bei der jeweils ein komplettes Bild berechnet wird. Trivial parallele Probleme stellen keine Ansprüche an die Programmierung oder die Kommunikation, da sie voneinander unabhängig sind. Die Teilaufgaben (Tasks) können mit dem sequentiellen Programm bearbeitet werden. Der erreichbare Speedup (siehe auch Abschnitt 4.5) ist sehr gut. Trivial parallele Probleme eignen sich sehr gut für die Bearbeitung auf einem Workstation-Cluster oder einer Compute-Farm. Die Methode ist allerdings nicht sehr flexibel im Hinblick auf Parallelisierung. Oft ist man daran interessiert, eine große Simulation durchzuführen anstatt viele kleine Simulationen mit verschiedenen Parametern. Wenn das zu simulierende Problem zu groß für einen seriellen Rechner ist, dann muß das Problem selbst parallelisiert werden.

4.2.2 Funktionale Zerlegung

Bei funktionaler Parallelität wird die Gesamtarbeit aufgeteilt in eine Reihe von Teilaufgaben, und jede Teilaufgabe wird auf einem anderen Prozessor bearbeitet. Der am häufigsten auftretende Fall ist der einer *Pipeline*, wie wir sie auch schon im Fall prozessorinterner Parallelität (Abschnitt 2.1.3) kennengelernt haben.

Beim Pipelining wird jedem Prozessor seine Teilaufgabe zugewiesen, und die Daten “fließen” durch die Pipeline. Abb. 7 zeigt eine Pipeline der Bildverarbeitung für Mustererkennung.

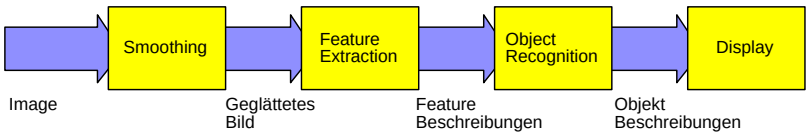


Abbildung 7: Bildverarbeitungs-Pipeline

Sobald ein Prozessor die Verarbeitung eines Bildes beendet hat

- ➠ reicht er die Bilddaten weiter an den nächsten Prozessor, damit dieser seine Arbeit an dem Bild ausführen kann
- ➠ erhält er vom vorhergehenden Prozessor das nächste Bild

Eine Pipeline arbeitet also im Prinzip wie ein Fabrikfließband.

Funktionale Parallelität ist in der Anwendung nur eingeschränkt nützlich, denn es gibt Probleme mit der Skalierbarkeit und der Lastverteilung.

Startup/Shutdown Solange die Pipeline zu Beginn noch nicht vollständig gefüllt ist, laufen die restlichen Prozessoren im Leerlauf (idle). Ebenso haben Prozessoren nach der Bearbeitung des letzten Bildes nichts mehr zu tun. Startup/Shutdown beeinträchtigen die Performance der Pipeline umso weniger, je mehr Bilder insgesamt zu verarbeiten sind.

Skalierbarkeit Sobald die Gesamtaufgabe (das Programm) in seine Teilaufgaben zerlegt ist, ist auch die Anzahl der nutzbaren Prozessoren festgelegt. Die Verfügbarkeit von mehr Prozessoren hat keine Auswirkung, die Skalierbarkeit ist also begrenzt. Zur Nutzung einer größeren Anzahl von Prozessoren muß das Programm auf eine entsprechend größere Anzahl von Teilaufgaben umgeschrieben werden. Dies ist jedoch nicht immer möglich.

Lastverteilung Bei der Implementierung einer Pipeline muß darauf geachtet werden, daß jeder Prozessor ähnlich viel Arbeit bekommt. Andernfalls sind die Prozessoren nicht mehr *lastbalanciert*, einige tun mehr Arbeit als andere. Dies führt zu Bottlenecks in der Pipeline und läßt einige Prozessoren unbeschäftigt, solange die anderen noch ihre Arbeit beenden.

Ein möglicher Weg, die Probleme der Lastverteilung und Skalierbarkeit zu reduzieren, liegt darin, eine zweite Ebene der Parallelität auf Task-Ebene einzuführen. Dabei wird jeder Teilaufgabe der Pipeline eine Anzahl von Prozessoren zugeordnet, siehe Abb. 8.

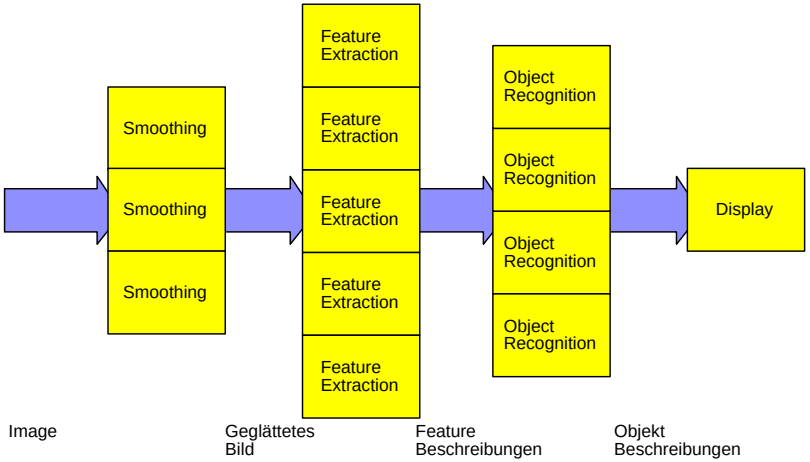


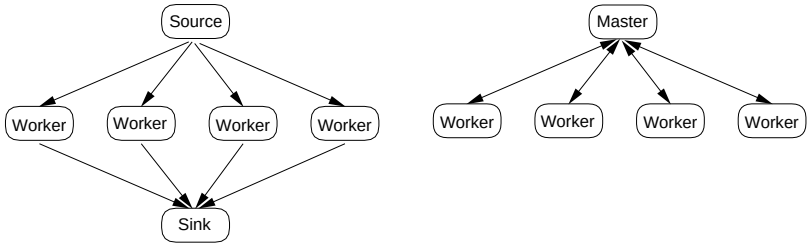
Abbildung 8: Bildverarbeitungs-Pipeline mit parallelen Funktionen

Diese Art Parallelität ist analog zum Konzept der *Taskfarm*. Die Grundlegende Struktur einer Taskfarm ist in Abb. 9 a) dargestellt.

- ☞ Der Source-Prozeß verteilt eine erste Anzahl von Teilaufgaben an die Worker-Prozesse und verteilt dann auf Anforderung die restlichen.

- ☞ Die Worker-Prozesse erhalten eine Teilaufgabe, bearbeiten sie und geben ihr Ergebnis an den Sink-Prozeß weiter.
- ☞ Der Sink-Prozeß empfängt die Resultate der erledigten Tasks, führt die Teilergebnisse zusammen und veranlaßt bei der Source das Senden der nächsten Teilaufgaben an die Worker.

Da Source- und Sink-Prozeß typischerweise viel weniger Arbeit als ein Worker zu tun haben, werden sie oft zu einem Prozeß zusammengefaßt. Die vereinfachte Struktur wird auch als Master/Worker- oder Master/Slave-Struktur bezeichnet (Abb. 9 b)).



a) Allgemeine Taskfarm b) Master/Slave

Abbildung 9: Allgemeine Taskfarm (a) und Master/Slave-Struktur (b)

Das Taskfarm-Konzept eignet sich gut zur dynamischen Lastbalancierung. Diese ist notwendig, wenn Tasks unterscheidlich lange, im Vorhinein unbekannte Bearbeitungszeiten benötigen. Das System ist inhärent unbalanciert, und in manchen Fällen arbeitet noch eine Teilmenge von PEs an ihrem letzten Task, während andere schon lange fertig sind. Dies kann umgangen werden, wenn man nach Möglichkeit die großen Tasks zuerst vergibt. Abb. 10 veranschaulicht diese Situation.

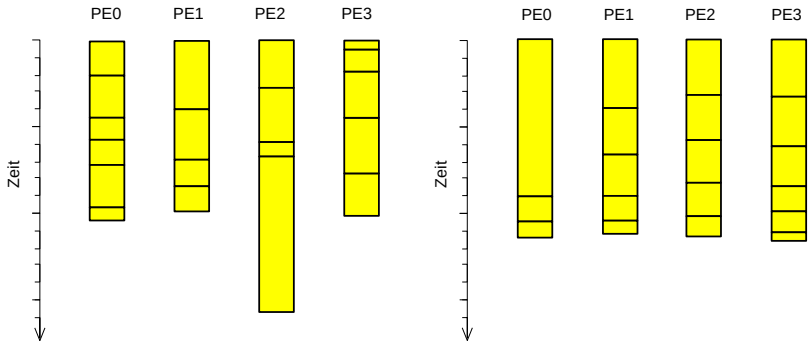


Abbildung 10: Zufällige Verteilung der Tasks auf die Worker (links) und Vergabe der größten Tasks zuerst (rechts)

4.2.3 Datenzerlegung

Bisher sind die Fälle besprochen worden, in denen die erste der unter Anschnitt 4.1 gestellten Fragen mit “ja” beantwortet werden konnte. In diesen Fällen konnte ein Problem in Teilaufgaben zerlegt werden, die ohne Datenzugriffskonflikte gelöst werden können. Der allgemeine und am häufigsten auftretende Fall ist aber die Zerlegung der Daten *mit* Abhängigkeiten beim Zugriff.

Anstatt die Teilaufgaben auf die Prozessoren zu verteilen erhält jeder Prozessor seinen eigenen Teil der Daten zur Bearbeitung. Die Gesamtmenge der Daten wird zunächst auf die Prozessoren aufgeteilt, die Berechnungen werden mit den Daten durchgeführt und die Ergebnisse schließlich auf irgend eine Weise kombiniert. Die Datenverteilung muß auch hier so durchgeführt werden, daß jeder PE ähnlich viel Arbeit zu tun hat[3]. Bei der Datenverteilung müssen auch die anderen unter 4.1 gestellten Fragen beantwortet werden:

- **?** Erkennung von Datenzugriffskonflikten: Wird in einer Gruppe von Anweisungen auf bestimmte Daten lesend *und* schreibend zugegriffen?
- **?** Können die Datenzugriffskonflikte durch Änderung der Ausführungsreihenfolge aufgelöst werden?
- **?** Ist die Änderung der Ausführungsreihenfolge algorithmisch zulässig?

Können wir die beiden letzten Fragen mit “ja” beantworten, so ist das Problem parallelisierbar. Dieses sicherzustellen erfordert jedoch oft einen beträchtlichen Aufwand.

Ein häufig auftretendes Beispiel für die Parallelisierung durch Datenzerlegung ist die *Gebietszerlegung (Domain Decomposition)*. Viele Probleme werden auf einem Gitter definiert, und die Lösung des Problems wird dann auf den Gitterpunkten berechnet. Zerlegt man das Gitter in (disjunkte) Blöcke und verteilt diese auf mehrere Prozesse, so kann das Problem in viel kürzerer Zeit gelöst werden, oder es kann ein viel größeres berechnet werden, als vorher möglich war.

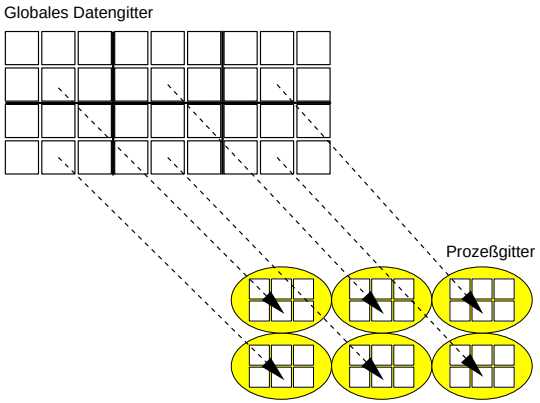


Abbildung 11: Verteilung eines Datengitters auf 6 Prozesse

Prozeßgitter Bei der Verteilung der Gitterblöcke muß beachtet werden, daß die Prozesse selbst ebenfalls auf einem logischen Gitter angeordnet werden müssen (*mapping*), damit die Nachbarschaftsbeziehungen zwischen den Gitterblöcken auch durch die Prozesse wiedergegeben werden. Jeder Prozeß

[3] Dies bedeutet nicht unbedingt, daß jeder PE gleich viel Daten bekommt.

muß wissen, wo sich die Nachbarblöcke zu seinen eigenen Daten befinden. Abb. 11 zeigt die Verteilung eines 2-D-Gitters auf 6 Prozesse und deren (logische) Anordnung.

Gitterpunkt-Update In jeder Iteration ist jeder der Prozesse verantwortlich für die Neuberechnung seiner lokalen Daten. Üblicherweise werden für jeden Datenpunkt Berechnungen ausgeführt, die die aktuellen Werte in diesem Element und in einer Anzahl von benachbarten Elementen verwenden. Dies ist beispielsweise bei numerischen Verfahren mit finiten Differenzen oder finiten Volumen der Fall. Nicht alle benötigten Nachbarelemente werden sich aber notwendigerweise im lokalen Speicherbereich des gleichen Prozesses befinden. Um den Zugriff auf die benötigten Datenelemente zu ermöglichen, zerlegt man nicht in disjunkte Blöcke, sondern läßt diese sich um eine gewisse Anzahl von Gitterzellen überlappen und speichert mit jedem Gitterblock auch den Überlapp-Bereich.

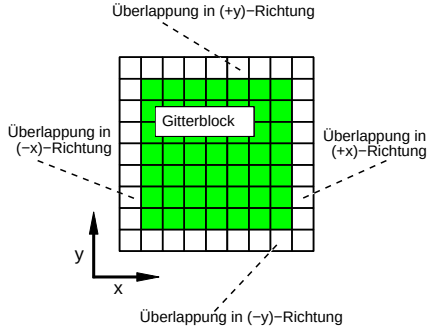


Abbildung 12: Lokales Daten-Array eines Gitterblocks mit seinen überlappenden Randzellen

Abb. 12 zeigt ein lokal gespeichertes Datenarray, das den Gitterblock einschließlich seiner überlappenden Randzellen beinhaltet. Diese zusätzlichen Randzellen werden auch als *ghost cells* bezeichnet. Das lokale Speichern aller für die Neuberechnung notwendigen Daten gestattet einen wesentlich schnelleren Zugriff als den auf entfernt gespeicherte Daten.

Randaustausch Die Randelemente des Daten-Arrays jedes Prozesses enthalten Daten eines Nachbarprozesses. Daher muß jeder Prozeß Kopien seiner Randelemente zu seinen Nachbarprozessen schicken, damit diese ihre Überlappbereiche aktualisieren können (*boundary swap*), siehe Abb. 13. Der im Bild gezeigte Prozeß *A* aktualisiert ein internes Element und benötigt deshalb keine Daten aus einem Überlappbereich. Prozeß *B* dagegen aktualisiert ein Element, welches Daten aus dem Überlapp mit Prozeß *A* erfordert. Prozeß *A* muß den Randaustausch mit Prozeß *B* durchführen, bevor Prozeß *B* auf diese Daten zugreifen darf.

Granularität Der Überlappbereich wird sich auf die Richtungen beschränken, in denen auf Nachbarelemente zugegriffen werden muß. Seine Größe hängt ab von der Anzahl der Nachbarpunkte, die in einer Richtung für den Update eines Datenelements notwendig sind. Je größer der Überlappbereich bei einer gegebenen Blockgröße ist, desto größer ist auch der Kommunikationsaufwand.

Da Kommunikation immer einen zusätzlichen Aufwand darstellt (zusätzlich zum Rechenaufwand), sollte es das Ziel jeder Blockzerlegung sein, das Verhältnis von Kommunikationaufwand zu Rechenaufwand so klein wie möglich zu halten. Da der Rechenaufwand in einem Block von seinem Volumen und der Kommunikationsaufwand von seiner Oberfläche abhängt, gilt es, das Oberflächen- zu Volumenverhältnis möglichst klein zu halten. Geht man von einer gleichen Anzahl von Randelementen in jeder Richtung aus (im einfachsten Fall genau *eins*), so erreicht man dies durch möglichst quadratische (2D) bzw.

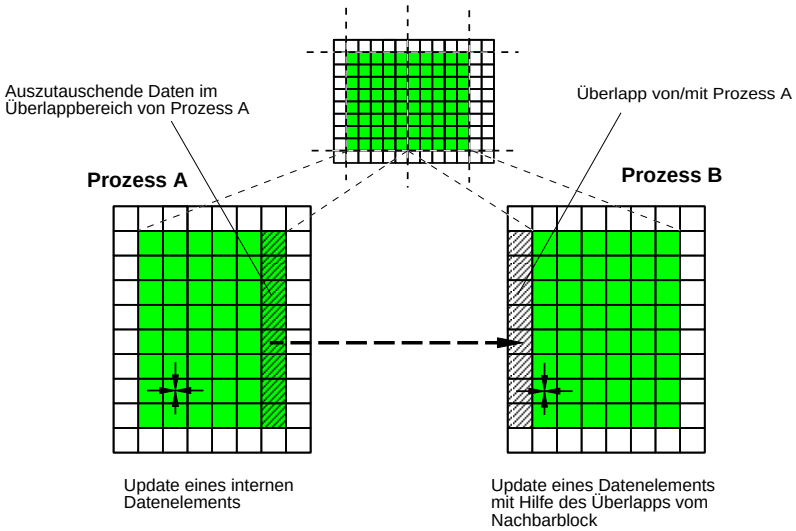


Abbildung 13: Randaustausch

würfelförmige (3D) Zerlegungen. Das Verhältnis wird auch umso besser, je größer ein Block ist. Zerlegungen mit relativ geringem Kommunikationsaufwand bezeichnet man auch als *grob granular*, solche mit relativ hohem Kommunikationsaufwand nennt man *fein granular*. Eine etwas andere Sichtweise des Beriffes *Granularität* betrachtet statt des Aufwandes für die Kommunikation deren Häufigkeit. Je häufiger ein Kommunikationsschritt durchgeführt werden muß, desto feiner ist die Granularität der Zerlegung.

4.3 Datenstrukturen und Algorithmen

Das Ziel der Parallelisierung ist bekanntlich, die für eine Problemlösung notwendigen Daten und die Arbeit so auf eine Anzahl von Prozess(or)en zu verteilen, daß das Ergebnis in möglichst kurzer Zeit (wall clock time) erreicht wird. Das Ergebnis muß das gleiche sein wie bei der Rechnung auf einem Prozessor, der Algorithmus kann jedoch variieren (siehe Abschnitt ”Heterogene Probleme” im Folgenden).

In Kapitel 4.2 sind bereits die wichtigsten Ansätze zur Parallelisierung besprochen worden. Im ingenieur- und naturwissenschaftlichen Bereich hat man es sehr oft mit der Berechnung diskreter Probleme zu tun, bei denen Differentialgleichungen oder Eigenwertprobleme auf diskreten Gittern gelöst werden.

Die Probleme lassen sich unterscheiden nach der Homogenität der (mathematischen) Modelle.

Homogene Probleme liegen vor, wenn man ein bestehendes serielles Lösungsverfahren so parallelisieren will/muß, daß genau die gleichen mathematisch/numerischen Operationen ausgeführt werden wie im seriellen Fall, das Ergebins also mathematisch exakt sein soll. Typische Beispiele sind Eigenwertprobleme wie z.B. in der Quantenchemie. Die Zerlegung der Probleme erfordert beträchtliche Eingriffe in die zugrundliegenden Lösungsverfahren, weshalb die Zahl parallelisierter Lösungsverfahren noch sehr gering und Gegenstand aktueller Forschung und Entwicklung ist.

Heterogene Probleme sind solche mit natürlichen, problembedingten Grenzflächen (Klimamodelle wie Ozean-Eis-Atmosphäre, Strömungs-/Struktur-Kopplung in der Aerodynamik), oder Probleme, die sich durch (z.B. geometrische) Blockzerlegung parallelisieren lassen (siehe Abschnitt 4.2.3).

Man geht dann so vor, daß die homogenen Teilgebiete oder Blöcke mit dem gewohnten homogenen, seriellen Algorithmus und Lösungs-Verfahren gelöst werden, und es findet ein Austausch über die Ränder statt (normalerweise mehrfach).

Da die grundlegenden Gleichungen ohnehin nur iterativ (und damit nur näherungsweise) gelöst werden können, und die genaue Iterationsstrategie immer noch viel Erfahrung (”schwarze Magie”) erfordert[4], ändert sich nichts Grundsätzliches an der Strategie, es kommen lediglich gewisse Freiheitsgrade hinzu:

Welches Lösungsverfahren für die Blöcke? Wie lange iteriert man auf dem Block? Wann tauscht man Daten aus? Wie iteriert man übergeordnet? Macht man den Austausch über alle Ränder immer für alle gleich, oder macht man das (problemabhängig) für einige mehr oder weniger oft?
Hier spielen auch noch Multilevelverfahren (Multigrid) hinein: welche Strategie verfolgt man auf den verschiedenen Gitterleveln?

Es muß betont werden, daß es für die Fragen keine allgemeingültigen Antworten gibt. Dennoch ist die Parallelisierung durch Domain Decomposition sehr beliebt, weil zur Lösung der Gleichungssysteme auf die gewohnten seriellen Verfahren zurückgegriffen werden kann. Bei dieser Vorgehensweise muß natürlich stets sichergestellt werden, daß sich die ausiterierte und konvergierte Lösung höchstens im Rahmen der Konvergenzgenauigkeit von der seriellen Lösung unterscheidet, während der Iterationsverlauf durchaus teilweise beträchtlich unterschiedlich sein kann. In technisch-wissenschaftlichen Anwendungsproblemen ist man oft weniger an einer mathematisch exakten Lösung interessiert als an einer Lösung, die im physikalisch-technischen Sinn ”genau” ist.

4.4 Overhead in parallelen Programmen

Jedes parallele Programm beinhaltet einen Overhead, der die erreichbare Performance verschlechtert. Wir betrachten als Overhead alles, was sich entweder nicht oder nur unvollständig parallelisieren läßt oder was durch die Parallelisierung an Aufwand hinzu kommt. Einige der im Folgenden angeführten Punkte werden an anderer Stelle näher besprochen.
Sequentielle Anteile Nahezu jedes Problem enthält Anteile, die sich nicht parallelisieren lassen. Dazu können die Initialisierung und die Ein-/Ausgabe gehören.
Zusätzlicher Rechenaufwand Der optimale serielle Algorithmus
– kann in manchen Fällen nur ungenügend oder gar nicht parallelisiert werden kann, weshalb dann ein weniger effizienter verwendet werden muß
– erfordert in der Summe zwangsläufig mehr Rechenoperationen, weil einmal berechnete Zwischenergebnisse nicht von den anderen Prozessoren weiterverwendet werden können, sondern dort lokal berechnet werden müssen.
Die Berechnung von Feld-Indices kann aufwendiger sein, z.B. weil Indices PE-abhängig werden.
Ungleiche Lastverteilung Unterschiedliche Prozessoren haben unterschiedlich viel Arbeit (siehe ”Lastverteilung” in Kap. 4.2.2).
Kommunikationsaufwand Der Aufwand für die Kommunikation ist ein zusätzlicher. Im besten Fall gelingt es, ihre Durchführung mit Rechenarbeit zu überlappen, sofern die Hardware dies unterstützt.
Synchronisation Mehrere Prozessoren müssen in ihrer Ausführung gleichzeitig an einem bestimmten Punkt des Programms angekommen sein (*Barrier*). Synchronisationsbedingte Wartezeiten von Prozessoren können eine der Hauptursachen für die Ineffizienz von parallelen Programmen sein.
Globale Kommunikation Bestimmte Operationen erfordern die Kommunikation aller oder vieler Prozesse mit einem oder wenigen. Dazu gehören Reduktions-Operationen wie globale Summen bei der

[4] Ein Zitat des CFD-Gurus Patrick J. Roache: "The newcomer to computational fluid dynamics (CFD) is forewarned: in this field there is at least as much artistry as science."

Residuumsbildung oder globale Extremwerte sowie Broadcast-Operationen beispielsweise zur Verteilung von Anfangswerten. Auch Synchronisationen können der globalen Kommunikation zugerechnet werden.

Verhältnis von Rechenzeit zu Kommunikationszeit Die Hinzunahme weiterer Prozessoren bei gleichbleibender Problemgröße oder eine ungünstige Aufteilung der Daten in Blöcke (großes Seitenverhältnis) verschlechtert das Volumen/Oberflächen-Verhältnis und damit das Verhältnis von Rechen- zu Kommunikationszeit (siehe auch Abschnitt "Granularität" in Kap. 4.2.3).

4.5 Effizienzanalyse und Performance-Messung

Gene Amdahl formalisierte 1967 erstmals eine Regel, die später als *Amdahl's Law* bezeichnet wurde. Das Gesetz von Amdahl besagt im Wesentlichen, daß die Ausführungsgeschwindigkeit eines parallelen Programms von dem Teil des Codes bestimmt wird, der am wenigsten effizient, nämlich seriell, abläuft. Um hohe Effizienz zu erreichen muß der Bereich eines Codes, der nicht parallelisiert werden kann, möglichst klein gehalten werden.

Wir geben zunächst einige Definitionen, die bei der Effizienzanalyse hilfreich sind.
Der *Speedup* beschreibt, um wieviel ein Problem auf N_{proc} Prozessoren schneller abgearbeitet wird als auf einem.

$$\text{Speedup}(N_{proc}) = \frac{\text{Time}(1)}{\text{Time}(N_{proc})}$$

Dabei ist Time(1) die Zeit für den besten seriellen Algorithmus auf einem Einzelprozessor und Time(N_{proc}) ist die Zeit für den parallelisierten Algorithmus auf N_{proc} Prozessoren.
Die *Effizienz* setzt den erreichten Speedup in Beziehung zum maximal möglichen.

$$\text{Effizienz} = \text{Speedup}/N_{proc}$$

Der Speedup ist gewöhnlich kleiner als der Idealwert N_{proc}, deshalb ist die Effizient dann auch kleiner als 1 (oder 100%).
Amdahls Gesetz betrachtet den Regelfall, bei dem ein Algorithmus nur unvollständig parallelisierbar ist. Wenn $(1-p)$ der Teil der Rechenzeit ist, in dem sequentiell gerechnet werden muß, dann kann man die sequentielle Rechenzeit formal aufspalten in

$$\text{Time}(1) = \text{Time}(1) \times (1-p) + \text{Time}(1) \times p$$

Nimmt man an, daß der Anteil Time$(1) \times p$ ideal parallelisierbar ist, so ist der maximal erreichbare Speedup

$$\text{Speedup}_{Amdahl} = \frac{\text{Time}(1)}{\text{Time}(1) \times (1-p) + \text{Time}(1) \times p/N_{proc}} = \frac{1}{1-p+p/N_{proc}}$$

mit dem Grenzwert für große Prozessorzahlen von

$$\text{Speedup}_{Amdahl}(N_{proc} \to \infty) = \frac{1}{1-p}$$

Die folgende Tabelle gibt einen Überblick über den Speedup gemäß Amdahl-Gesetz.

	N_{proc}								
p (||-Anteil)	2	4	8	32	64	256	512	1024	Limes
0.70	1.54	2.11	2.58	3.17	3.22	3.30	3.32	3.33	3.33
0.80	1.67	2.50	3.33	4.44	4.71	4.92	4.96	4.99	5.00
0.85	1.74	2.76	3.90	5.66	6.12	6.52	6.59	6.63	6.67
0.90	1.82	3.08	4.71	7.81	8.77	9.66	9.83	9.91	10.0
0.95	1.91	3.48	5.93	12.6	15.4	18.6	19.3	19.6	20.0
0.97	1.94	3.67	6.61	16.6	22.2	19.6	31.4	32.3	33.3
0.99	1.98	3.88	7.48	24.4	39.3	71.1	83.8	91.2	100
0.995	1.99	3.94	7.73	27.7	48.7	113	144	167	200
0.999	2.00	3.99	7.94	31.4	60.2	204	388	506	1000

Superlinearer Speedup Speedup und Effizienz können unter bestimmten Umständen für gut konditionierte Probleme auch bessere Werte als die theoretischen Grenzwerte erreichen, also Speedup $> N_{proc}$ und Effizienz > 1. Dies liegt dann in der Praxis aber z.B. in der Prozessorarchitektur begründet, weil bei der Verteilung eines Problems fester Größe auf immer mehr Prozessoren die effektive (gesamte) Größe des verwendeten Caches zunimmt[5], und das Problem schließlich vollständig im Cache gelöst werden kann.

4.6 Programmiermodelle

Wegen der Komplexität massiv-paralleler Hardware und ihrer inhärenten Unterschiede zu einer Single-Prozessor/Single-Memory-Architektur ist es nicht möglich, rein traditionelle Programmiersprachen zu verwenden und gute Performance zu erreichen. Im Lauf der Zeit haben sich zwei Programmier-Paradigmen herausgebildet, die auf breiter Basis akzeptiert wurden, insbesondere auf größeren Maschinen:

Daten-Parallel - Hierbei kontrolliert ein einziges Programm die Verteilung und Verarbeitung der Daten auf allen Prozessoren. Eine daten-parallele Sprache unterstützt typischerweise Feld-Operationen und gestattet die Verwendung ganzer Felder in Ausdrücken. Der Compiler erzeugt die Anweisungen zur Verteilung der Feldelemente auf die verfügbaren Prozessoren. Jeder Prozessor ist dann für seine Teilmenge der Daten 'verantwortlich', die er in seinem lokalen Speicher hält. Daten-parallele Applikationen sind im Prinzip auf MIMD- und auf SIMD-Hardware lauffähig. Beispielhafte Standards sind die Fortran-Erweiterung *High Performance Fortran (HPF)* und das sprachunabhängige *OpenMP* (nur für Shared-Memory-Architekturen), siehe auch Kap. 5.

Message Passing - Hierbei werden individuelle, in einer Standard-Programmiersprache geschriebene Programme auf jedem Prozessor gestartet, die Zugriff haben auf den jeweiligen lokalen Speicher. Es existiert eine Anzahl von Message-Passing-Umgebungen, die im Allgemeinen von den auf den Knoten laufenden Programmen aufrufbare Bibliotheksfunktionen zur Verfügung stellen. Zur Datenübertragung kann der Programmierer Sende- und Empfangs-Routinen verwenden, die von der Bibliothek bereitgestellt werden. Darüberhinaus sind meistens noch komplexere kollektive Operationen verfügbar. In der Praxis Bedeutung erlangt haben das *Message Passing Interface (MPI)* und *Parallel Virtual Machine (PVM)*, siehe Kap. 6.

In den folgenden Abschnitten wird auf die wichtigsten Realisierungen eingegangen. An dieser Stelle sei auf eine umfassende Sammlung von Informationen zum parallelen Rechnen in World Wide Web verwiesen: das *Internet Parallel Computing Archive*[6].

[5] **Aber**: "If you fit in L1 cache, you ain't supercomputing". Krste Asanovic
[6] Das Internet Parallel Computing Archive: `http://www.hensa.ac.uk/parallel/`

5 Daten-parallele Programmierung

Der Schwerpunkt dieser Einführung liegt auf der Parallelisierung mittels Message Passing, daher wird nur kurz und beispielhaft auf die daten-parallele Programmierung eingegangen.

5.1 High Performance Fortran

Das High Performance Fortran Forum (HPFF) veröffentlichte bald nach seiner ersten Sitzung (Januar 1992) im Mai 1993 die High Performance Fortran Spezifikation 1.0. Die Aufgabe des HPFF wurde dadurch erleichtert, daß einige seiner Mitglieder auch im Standardisierungs-Kommittee für Fortran90 mitarbeiteten. Die Arbeiten für den Fortran90-Standard (veröffentlicht 1991) beinhalteten bereits eine Fülle von Gesichtspunkten einer daten-parallelen Sprache, was sich in einer signifikanten Menge von Feld-Eigenschaften (*Array Features*) niederschlägt. HPF wiederum ergänzt Fortran90 um Konstrukte zur Datenverteilung, neue daten-parallele Eigenschaften und zusätzliche intrinsische und Bibliotheks-Funktionen zur Unterstützung paralleler Architekturen.

Das Programmiermodell von HPF ist SPMD, wobei die Parallelität auf der Ebene von Feldern, Feldzuweisungen und Schleifen unterstützt wird. Der Compiler erzeugt die notwendigen Kommunikationsaufrufe, es gibt kein explizites Message-Passing. Das (single) Programm hat ein globales Datenmodell (im Gegensatz zu z.B. MPI, wo die Datenstrukturen in kleinere Teile zerlegt werden müssen).
Die wichtigste Syntax-Erweiterung in HPF gegenüber Fortran90 ist eine umfassende Schleifen-Anweisung für datenparallele Zuweisung, `FORALL`. Damit wird die Array-Notation in Fortran90 ergänzt. `FORALL` ist Bestandteil von Fortran95 und überdies unterstützen eine Reihe von Fortran90-Compilern bereits dieses Konstrukt in seriellen Programmen. HPF erweitert Fortran90 auch um Direktiven, die es gestatten, eine abstakte Prozessor-Maschine zu beschreiben (`!HPF PROCESSORS`, siehe Bild 14), auf der Felder verteilt werden können (`!HPF DISTRIBUTE`), was im folgenden Beispiel als `BLOCK`s geschieht.

```
INTEGER a (16,16), b(16,16)
!HPF PROCESSORS (4,4)
!HPF DISTRIBUTE a(BLOCK,BLOCK), b(BLOCK,BLOCK)
!HPF ALIGN a(i,j) WITH b(i+1,j+1)
 ...
    a(i+1,j+1)=a(i,j)+b(i+1,j+1)
 ...
```

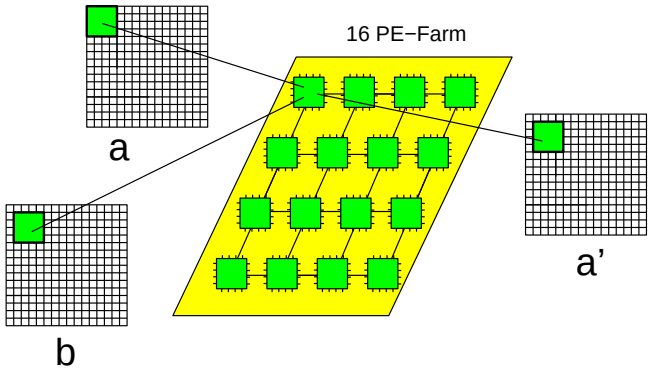


Abbildung 14: Datenverteilung und -Anordnung im HPF-Beispiel

Die Kommunikation kann für bestimmte Berechnungen minimiert werden, wenn die Felder aufeinander ausgerichtet werden (`!HPF ALIGN`). In einigen Fällen ist aber Kommunikation notwendig für das Überschreiben von `a`.
Wegen der Realisierung über Direktiven kann ein HPF-Programm im Prinzip und mit bestimmten Einschränkungen auch seriell laufen (einschließlich Debugging).

5.2 OpenMP

OpenMP umfaßt eine Sammlung von Compiler-Direktiven, Bibliotheks-Funktionen und definierten Umgebungsvariablen zur Ausnutzung der parallelen Eigenschaften von **Shared-Memory**-Maschinen. Das Ziel von OpenMP ist die Bereitstellung eines portablen, herstellerunabhängigen parallelen Programmiermodells.

OpenMP verwendet das sogen. *fork-join*-Modell für die parallele Ausführung. Ein Programm startet als ein einzelner *Thread* (ein feingranularer Prozeß), der *Master Thread*. Beim Erreichen des ersten parallelen Konstrukts erzeugt der Master Thread eine Gruppe von Theads und wird selbst Master der Gruppe. Nach Beendigung des parallelen Konstrukts synchronisiert sich die Gruppe, und nur der Master Thread arbeitet weiter. Die Parallelisierung erfolgt auf Schleifenebene.

Als Beispiel zwei Fortran-Unterprogramme, wie sie häufig bei der numerischen Berechnung diskreter zweidimensionaler Probleme auftreten.

Das erste initialisiert zwei Felder für die diskrete numerische Lösung `u(i,j)` und die exakte Lösung `f(i,j)`. Die Doppelschleife wird von den verfügbaren Prozessoren parallel verarbeitet (*parallel do*), wobei die Indexbereiche für `i` und `j` verteilt werden. Die beiden Felder sowie die Größen `n, m, dx, dy, alpha` stehen allen Prozessen global zur Verfügung (werden *geshared*), die Größen `i,j,xx,yy` sind pro Prozeß lokal (*private*).

```
      subroutine initialize (n,m,dx,dy,u,f,alpha)
!*****************************************************
! Initializes data
! Assumes exact solution is u(x,y) = (1-x^2)*(1-y^2)
!*****************************************************
      implicit none

      integer n,m
      real u(n,m),f(n,m),dx,dy,alpha

      integer i,j, xx,yy

      dx = 2.0 / (n-1)
      dy = 2.0 / (m-1)

! Initialize initial condition and RHS

!$omp parallel do
!$omp& shared(n,m,dx,dy,u,f,alpha)
!$omp& private(i,j,xx,yy)
      do j = 1,m
         do i = 1,n
            xx = -1.0 + dx * (i-1)        ! -1 < x < 1
            yy = -1.0 + dy * (j-1)        ! -1 < y < 1
            u(i,j) = 0.0
            f(i,j) = -alpha *(1.0-xx*xx)*(1.0-yy*yy)
     &           - 2.0*(1.0-xx*xx)-2.0*(1.0-yy*yy)
         enddo
```

```
      enddo
!$omp end parallel do

      return
      end
```

Die Fehler-Routine hat ähnliche Konstrukte für die Doppleschleife, in der an jedem Punkt die Abweichung der numerischen Lösung `u(i,j)` von der exakten Lösung `f(i,j)` berechnet wird. Hinzu kommt die Berechnung des Gesamtfehlers `error`, die in einer globalen Reduktionsoperation als Summe (+) über die Teilsummen von `error` auf allen Prozessen gebildet wird.

```
      subroutine error_check (n,m,u,f)
      implicit none
!*************************************************************
! Checks error between numerical and exact solution
!*************************************************************

      integer n,m
      real u(n,m),f(n,m),dx,dy

      integer i,j
      real temp,error

      error = 0.0

!$omp parallel do
!$omp& shared(n,m,u,f,error)
!$omp& private(i,j)
!$omp& reduction(+:error)
      do j = 1,m
         do i = 1,n
            temp  = u(i,j) - f(i,j)
            error = error + temp*temp
         enddo
      enddo

!$omp end parallel do

      error = sqrt(error)/(n*m)

      print *, 'Solution Error : ',error

      return
      end
```

Man erkennt, daß die Quelle des seriellen Programms lediglich um Kommentarzeiten mit den OpenMP-Direktiven erweitert werden muß. Ein weiterer Vorteil der Verwendung von OpenMP ist die Möglichkeit, die Parallelisierung schrittweise durchzuführen.

6 Message-Passing-Bibliotheken

6.1 Allgemeines und Gemeinsamkeiten

Parallele Prozesse werden gleichzeitig ausgeführt, können aber im Allgemeinen nicht unabhängig voneinander arbeiten, da zwischen ihnen Daten ausgetauscht werden müssen. Das Message-Passing-Modell (kurz *MP-Modell*) geht von einer Menge Prozessen aus, die nur lokalen Speicher verfügbar

haben, die aber mit anderen Prozessen kommunizieren können, indem die Nachrichten (*messages*) senden und empfangen. Die Art der Implementierung der Kommunikationsschicht (im Sinne eines Schichtenmodells der Kommunikation) ist nicht Bestandteil des MP-Modells. Dieses kann auf verschiedene Weise implementiert sein: im lokalen Speicher eines Einzelprozessores, im gemeinsamen Speicher eines Mehrprozessorsystems, in einer Anzahl von über das Internet verbundenen PCs oder von Prozessoren, die durch ein proprietäres Netz verbunden sind.

Diese Universalität hat auch zu der heutigen weiten Verbreitung von Message-Passing in parallelen Programmen geführt. Message-Passing gestattet wegen des expliziten parallelen Programmierens die direkte Implementierung paralleler Algorithmen. Dabei muß der Algorithmus ausdrücklich spezifizieren, auf welche Art und Weise Prozesse zusammenarbeiten sollen, um ein bestimmtes Problem zu lösen.

6.1.1 Historisches

Der erste Ansatz zur Schaffung eines Standards im Bereich der parallelen Programmierung mit Message Passing bestand in einer Sammlung von Macros, die von C aus benutzt werden konnten. Sie nannte sich PARMACS (*Parallel Macros*) und entstand am Argonne National Laboratory (USA). Unter dem gleichen Namen entstand eine Fortentwicklung, die wie das ebenfalls aus PARMACS entwickelte P4 (*Portable Programs for Parallel Processors*) nicht mehr aus Makros, sondern aus Funktionen besteht, welche mittels Libraries für C und FORTRAN zur Verfügung gestellt werden.

Die heute gebräuchlichsten Implementierungen eines MP-Modells sind *Message Passing Interface (MPI)* und *Parallel Virtual Machine (PVM)*. Das PVM-Projekt begann 1989 am Oak Ridge National Laboratory und später an der University of Tennessee und führte 1991 zur Veröffentlichung der Version 2 von PVM. Die aktuelle Version dieser Quasi-Standard-Bibliothek ist 3.4.2. Auch innerhalb Europas wurde Anfang der 90er Jahre ein Message-Passing-Standard entwickelt: *Message Passing Interface*. Der Standard MPI-1 wurde 1994 veröffentlicht, MPI-2 folgte in Jahr 1997. Beide Message-Passing-Systeme sind auf einer großen Anzahl von Plattformen implementiert.

6.1.2 Grundfunktionen

Grundlegende Gemeinsamkeiten von Message-Passing betreffen die auftretenden Kommunikationsmuster und die dafür notwendigen elementaren MP-Funktionen zur Abwicklung der Kommunikation. In den folgenden Abschnitten werden zunächst eine Reihe von Begriffen erläutert, die in den verschiedenen implementierten MP-Systemen Verwendung finden (*"Vokabeln lernen"*).

Kommunikationsmuster Die auftretenden Kommunikationsmuster lassen sich in die beiden folgenden Gruppen einteilen:

- Punkt-zu-Punkt-Kommunikation (einseitig, zweiseitig)

  einseitig: am Transport von Daten ist nur ein Prozess(or) beteiligt
  (Realisierung: Cray Shmem-Library, MPI-2)

  zweiseitig: an jedem Datentransport sind ein Sender und ein Empfänger beteiligt
  (Realisierung: PVM, MPI)

- Kollektive Kommunikation (mehrseitig, allseitig)

  baut auf der Punkt-zu-Punkt-Kommunikation auf
  (Beispiele: Barriere, Eureka, Broadcast)

Elementare Message-Passing-Funktionen Der Austausch von Daten zwischen Prozessoren erfordert die folgenden elementaren Funktionen:

- Bestimmen der Anzahl von Prozessoren, die an der Bearbeitung der Aufgabe beteiligt sind
- Bestimmen des Kennzeichens des aktuellen Prozessors

- ☞ Senden einer Message an einen Prozessor
- ☞ Empfangen einer Message von einem Prozessor

Hinzu kommen ggfs. eine Initialisierungs- und eine Abschlußfunktion, sowie zur Synchronisation der Prozesse eine Barriere (Funktion, an der alle Prozesse ankommen müssen, damit sie verlassen werden kann) und eine Heureka-Funktion (Mitteilen an alle Prozesse, daß ein bestimmter Zustand erreicht worden ist).

Messages Nachrichten (Messages) sind Datenpakete, die zwischen Prozess(or)en bewegt werden (Bild 15).

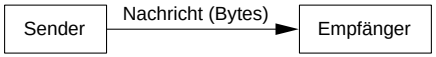


Abbildung 15: Transport einer Nachricht

Dem MP-System müssen die folgenden Informationen gegeben werden, durch die eine Nachricht beschrieben werden:

- **?** Welcher Prozessor sendet die Nachricht?
- **?** Wo liegen die Daten auf dem sendenden Prozessor?
- **?** Welcher Datentyp wird verschickt (Konvertierung, Struktur)?
- **?** Wieviele Daten werden verschickt?
- **?** Welche(r) Prozessor(en) sollen die Nachricht erhalten?
- **?** Wo sollen die Daten auf dem empfangenden Prozessor gespeichert werden?
- **?** Wieviele Daten kann der empfangende Prozessor speichern?

Das bedeutet, daß neben dem eigentlichen Nutzinhalt jeder Nachricht eine Reihe von Zusatzinformationen vorliegen müssen bzw. mit der Nachricht mitgeschickt werden müssen. Das MP-System muß seinerseits Informationen liefern können über den Verlauf und den Zustand der Kommunikation entsprechend dem dafür festgelegten Protokoll. Message-Passing ist somit in gewisser Weise analog zum Versenden von Nachrichten per Post oder Telefax im täglichen Leben (siehe auch *"Eselsbrücken"* weiter unten).

Pufferung von Nachrichten Die Sende- und Empfangs-Vorgänge können in verschiedenen Modi ablaufen. Ein wichtiger Aspekt ist dabei die Pufferung von Nachrichten. Das Senden einer Nachricht kann auf zwei Arten erfolgen, *gepuffert* oder *ungepuffert*. Beim *ungepufferten* Senden (Abb. 16) wird die Nachricht direkt in den Empfangspuffer des Empfängers kopiert.

Abbildung 16: Ungepuffertes Senden einer Nachricht

Beim *gepufferten* Senden (Abb. 17) wird die Nachricht zunächst in einen temporären lokalen Puffer des Senders kopiert und von dort aus zum Empfänger geschickt.
Ein möglicher Vorteil der Pufferung ist, daß der Inhalt des Sendepuffers weiterverarbeitet werden kann (z.B. durch Überschreiben), ohne auf den korrekten und vollständigen Empfang durch den Empfänger zu warten. Pufferung erfordert aber Allokierung zusätzlichen Speichers und zusätzliche Kopiervorgänge innerhalb des Speichers, ist also aufwendiger.

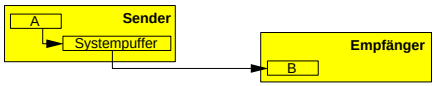


Abbildung 17: Gepuffertes Senden einer Nachricht

6.1.3 Message-Passing-Terminologie

Kommunikationsprozeduren können eine Reihe von Eigenschaften haben, die beachtet werden müssen.

Nicht-blockierend: Eine nicht-blockierende Prozedur kann zurückkommen, bevor ihre Operationen beendet sind. Das bedeutet auch, daß der Benutzer die Datenbereiche dieser Prozedur nicht mit Sicherheit anderweitig verwenden kann.

Blockierend: Wenn eine blockierende Prozedur zurückkommt, können ihre Datenbereiche wiederverwendet werden.

Lokal: Eine Prozedur heißt lokal, wenn sie keine Kommunikation mit einem anderen Prozeß beinhaltet.

Nicht-lokal: Eine Prozedur heißt nicht-lokal, wenn sie Kommunikation mit anderen Prozessen beinhalten könnte.

Kollektiv: Eine Prozedur heißt kollektiv, wenn alle Prozesse einer Gruppe sie aufrufen müssen.

Mit diesen Definitionen können wir allgemein Sendeprozeduren beschreiben.

Synchron: Eine synchrone Sendeprozedur kehrt erst zurück, wenn sie mit einer passenden Empfangsprozedur Kontakt aufgenommen hat. Sie ist nicht-lokal.

Gepuffert: Ein Sendeaufruf mit Pufferung kehrt zurück, sobald der Sendepuffer wiederverwendet werden kann. Der Aufruf ist lokal.

Standard: Die *Standard*-Sendeprozedur kann synchron oder gepuffert sein (in MPI). Sie ist nicht-lokal. (Ein Programm ist nur dann konform zum MPI-Standard, wenn alle *Standard*-Sendeprozeduren durch synchrone Sendeaufrufe ersetzt werden können.)

Ready: Ein *Ready*-Sendeaufruf darf nur dann erfolgen, wenn ein passender Empfangsaufruf bereits gemacht wurde (in MPI).

Die verschiedenen MP-Systeme sehen die mögliche Verwendung der unterschiedlichsten Sende- (und auch Empfangs-)Modi vor, die in verschiedenen Einsatzbereichen durchaus auch ihre Daseinsberechtigung haben. In der Praxis wird man sich jedoch auf ein oder zwei Modi beschränken. Zum besseren Verständnis ist es vorteilhaft, die folgenden Analogien aus dem täglchen Leben zu betrachten (*Eselsbrücken*).

Blockierendes synchrones Senden: Telefon.

Blockierendes gepuffertes Senden: Sprechen auf einen Anrufbeantworter.

Nicht-blockierendes synchrones Senden: Einschreiben mit Rückschein.

Nicht-blockierendes gepuffertes Senden: Brief, E-Mail.

Blockierendes Empfangen: Telefon.

Nicht-blockierendes Empfangen: Brief, E-mail, Anrufbeantworter.

In diesen Beispielen liegt der Systempuffer beim Empfänger. In MPI (s. Abschnitt 6.3) hat der Sender den Systempuffer.

6.2 Parallel Virtual Machine (PVM)

6.2.1 Einleitung

Das Message-Passing-System *Parallel Virtual Machine (PVM)* wurde zwar ursprünglich für die Anwendung in heterogenen Netzwerken aus verschiedenen Prozessoren entworfen, ist aber auch in der Lage, echt parallele Computer einzubinden und deren eigene (schnelle) Konstrukte zur Nachrichtenübermittlung zwischen den Prozessoren innerhalb des Parallel–Computers zu benutzen. In einer heterogenen Umgebung übernimmt PVM gegebenenfalls notwendige Datenkonvertierungen (z.B. bei unterschiedlichen Zahlendarstellungen oder unterschiedlichem Byte-Ordering).

Das PVM-Konzept vereint alle beteiligten Prozessoren zu einem *virtuellen Parallel-Computer*. Dabei wird auf jedem Prozessor ein Dämon (Hintergrundprozeß) gestartet, der die Nachrichten von den und an die Tasks (Prozesse) weiterleitet. Verschiedene Benutzer können auch überlappende virtuelle Maschinen definieren.

Von der PVM–Console aus, die zur Einrichtung der virtuellen Maschine gestartet werden muß, können Prozessoren aufgenommen und gelöscht, Tasks initiiert und terminiert, sowie Status-Abfragen verschiedener Art gemacht werden.

All dies ist auch vom Programm aus möglich, so daß die Konfiguration dynamisch verändert werden kann (etwa um zusätzlich benötigte Rechenleistung zu addieren). Insbesondere kann so auf den Ausfall von Prozessoren (die von PVM sofort automatisch erkannt und ausgeschlossen werden) reagiert werden. Es liegt allerdings in der Hand des Programmierers, daß z.B. eventuell verloren gegangene Tasks neu gestartet werden.

6.2.2 Nachrichtenaustausch

Die wichtigste Funktion, das Versenden von Nachrichten (*messages*), läuft i.a. in drei Stufen ab (die Namen der Routinen sind für das Fortran-Language-Binding):

1. Ein Puffer für die Nachricht wird kreiert, wobei der alte Puffer gelöscht wird (`pvmfinitsend`).
2. Arrays von Daten werden in den Puffer gepackt, wobei innerhalb eines Aufrufs der Datentyp konsistent sein muß (`pvmfpack`)
3. Die Nachricht wird an einen (`pvmfsend`) oder mehrere (`pvmfmcast`) Kommunikationspartner versandt. Die Routine kehrt sofort zurück, wenn sich die Nachricht auf dem Weg befindet (asynchron).

Das Packen und Versenden einer Nachricht kann auch in einem Schritt geschehen (`pvmfpsend`), wenn nur Daten gleichen Typs verschickt werden sollen.

Der Empfangsprozeß muß genau umgekehrt ablaufen:

1. Eine Nachricht wird empfangen (`pvmfrecv`).
2. Die Daten werden aus dem Eingangspuffer ausgepackt und in lokalen Arrays abgelegt. *WICHTIG:* Die Funktionsaufrufe von Packen und Auspacken müssen genau korrespondieren, andernfalls ist Chaos vorprogrammiert!

Auch hier kann das Empfangen und Auspacken in einem Schritt geschehen (`pvmfprecv`), wenn alle zu empfangenden Daten den gleichen Typ haben.

Eine Eigenschaft der oben genannten Empfangsfunktion ist, daß sie erst zurückkehrt, wenn sie die Nachricht empfangen hat. Da jedoch die Kommunikation asynchron verläuft, kann das einige Zeit

dauern oder sogar zu einem *deadlock* führen, falls die Nachricht verlorengegangen ist. Daher stellt PVM zwei weitere Funktionen bereit, um mit eingehenden Nachrichten umzugehen.

Beim nicht-blockierenden Empfang (`pvmfnrecv`) wird eine vorhandene Nachricht empfangen. Ist aber die gewünschte Nachricht nicht da, kehrt die Funktion mit dieser Information zurück. Die Probe-Funktion (`pvmfprobe`) zeigt dagegen nur an, ob die Nachricht da ist, ohne sie (im positiven Fall) zu empfangen.

6.2.3 Weitere Funktionen

Damit ein Prozeß überhaupt an der Kommunikation teilnehmen kann, muß er sich bei PVM anmelden. Dies geschieht mit der Funktion `pvmfmytid`. Dabei erhält er einen eindeutigen *Task Identifier (tid)*, über den er innerhalb von PVM angesprochen werden kann. Mit der Funktion `pvmfexit` meldet sich ein Prozeß wieder ab, wobei er allerdings nicht terminiert wird, sondern als normaler UNIX-Task bestehen bleibt.

Daneben gibt es auch Funktionen zum Starten (`pvmfspawn`) und Beenden (`pvmfkill`) von Tasks, sowie verschiedene Informationsfunktionen, z.B. `pvmfparent`, das den *tid* des Tasks angibt, von welchem der aktuelle Task gestartet wurde (*PvmNoParent* entspricht dem Start vom Unix-Prompt).

Das Starten von Prozessen kann auf drei verschiedene Arten geschehen:

- ☞ Der Prozeß wird von PVM automatisch auf der bisher am wenigsten ausgelasteten Maschine gestartet (Load Balancing).
- ☞ Der Benutzer (oder das Programm, das einen weiteren Prozeß startet) kann den Maschinentyp angeben, auf dem der neue Prozeß erzeugt werden soll.
- ☞ Der Benutzer (oder das Programm, das einen weiteren Prozeß startet) nennt den genauen Namen der Maschine, auf der der Prozeß gestartet werden soll.

PVM kennt neben den Routinen für Punkt-zu-Punkt-Kommunikation unter anderem auch Aufrufe zur Synchronisation (`pvmfbarrier`) und ermöglicht die Bildung dynamischer Prozeßgruppen (`pvmfjoingroup`), innerhalb derer die beteiligten Prozesse zusätzliche, gruppenbezogene IDs erhalten, die mit `pvmfgettid` abgefragt werden koennen.

Da weitere Funktionen für unsere einfachen Zwecke nicht benötigt werden, seien Interessierte an dieser Stelle auf die Dokumentation auf der PVM-Homepage[7] im WWW hingewiesen. Alle PVM-Routinen sind auch in den Manual-Pages dokumentiert. Die einführende Manual-Page findet man unter "`man pvm_intro`". Außerdem soll auch hier der Hinweis auf das *Internet Parallel Computing Archive*[8] beachtet werden.

6.2.4 Implementierung von PVM-Programmen

PVM stellt eine Reihe von Definitionen in einem Include-File zur Verfügung, das in jede Routine eingebunden werden muß, die PVM-Aufrufe enthält. Das Includefile heißt `pvm.h` für C-Programme und `fpvm3.h` für Fortran-Programme. Die PVM-Routinen sind systematisch benannt: `pvm_*` für die C-Calls und `pvmf*` für die Fortran-Calls. Die Parameterlisten der Fortran-Calls sind gegenüber den C-Calls um einen Parameter, den Error-Code, erweitert. Beim Link-Vorgang muß die PVM-Library mit der Option `-lfpvm3` (Fortran) bzw. `-lpvm3` (C) eingebunden werden, ausserdem müssen die Suchpfade für die Include-Files und die Bibliotheken angegeben werden, z.B.:

```
cc -I/path/to/pvm/include/ -L/path/to/pvm/lib/ARCH/ -lpvm3 -lgpvm3 myprog.c
```

Im Kapitel 7 findet man ein einfaches, illustrierendes Beispiel für ein PVM-Programm.

[7] Die PVM-Homepage im WWW: `http://www.epm.ornl.gov/pvm/pvm_home.html`
[8] Das Internet Parallel Computing Archive: `http://www.hensa.ac.uk/parallel/`

6.2.5 Ausführung von PVM-Programmen

Zur Benutzung vom PVM müssen Umgebungsvariablen gesetzt werden. Für die C-Shell z.B. Im CIP-Pool des HFI:

```
setenv PVM_ROOT /usr/shared/pvm3
setenv PVM_DPATH $PVM_ROOT/lib/pvmd
set path=( $path $PVM_ROOT/lib/ )
```

PVM-Programme werden interaktiv unter der PVM-Console (`pvm`) gestartet. Das Startscript heißt `pvm`, und sollte der Einfachheit halber immer mit dem Namen eines Hostfiles gestartet werden, welches die wichtigsten Voreinstellungen enthält:

```
pvm ./hostfile
```

mit einem beispielhaften Inhalt des Hostfiles wie folgt:

```
* ep=$PVM_ROOT/lib/LINUX/:$HOME/EDV2/parallel/ dx=$PVM_ROOT/lib/pvmd
cip5008
cip5009
```

Neben der Pfadangabe für die unter der PVM-Console auszuführenden Programme (`ep=`) und die PVM-Dämonen (`dx=`) werden die Namen der Hosts angegeben, die die virtuelle Maschine umfassen soll (hier: `cip5008` und `cip5009`). Der Start der parallelen Anwendung mir `nproc` Prozessen erfolgt dann mit dem Kommando

```
spawn -nproc a.out
```

in der PVM-Console. Weitere Informationen findet man in der oben erwähnten Dokumentation zu PVM.

6.3 Message Passing Interface (MPI)

6.3.1 Einleitung

Während PVM als Projekt einer kleinen Anwendergruppe begann und von Beginn an auf eine heterogene Netzwerkumgebung ausgelegt war, startete MPI als Gemeinschaftsunternehmung einer ca. 60-köpfigen Gruppe aus 40 Gruppen von Anwendern und den wichtigsten Hardware-Herstellern und konzentrierte sich zunächst auf homogene Rechnerumgebungen. Die Einbeziehung der Hersteller in die Standardisierungsarbeit begründete die rasche Verfügbarkeit und den Erfolg von MPI, das heute auf praktisch allen Rechnerplattformen herstellerseitig unterstützt wird. Das Ziel einer unfangreichen Funktionalität und der Source-Code-Portabilität und auch das Design von MPI haben Anteil an Erfolg und Akzeptanz in der Nutzerschaft, denn MPI beschreibt nur die Software-Schnittstelle für den Programmierer und befaßt sich nicht mit Details der Implementierung oder des Zusammenspiels mit dem Betriebssystem. Diese Freiheiten der Implementierungen gestatten die effiziente Nutzung unterschiedlichster Hardware und Systeme. MPI ist als Library realisiert. Obwohl MPI einen großen Funktionsumfang besitzt (ca. 140 Funktionen in MPI-1: *MPI is big*), genügen wenige unterschiedliche Aufrufe zur Abdeckung der vollen Funktionalität eines vollständigen Programms (*MPI is small*).

Die in diesem Abschnitt beschriebenen Routinen beziehen sich auf die erste Version des MPI-Standards, MPI-1. Die Arbeiten an MPI sind fortgesetzt worden und führten zum MPI-2-Standard, in dem hauptsächlich paralleler I/O, einseitige Kommunikation und Prozeßmanagement eingeführt wurden.

Während PVM mit seinen vielfältigen Möglichkeiten zum Prozeßhandling einschließlich dynamischer Gruppen die einfache Realisierung von unterschiedlichen Parallelisierungskonzepten gestattet, wird in MPI-1 die Größe des (virtuellen) Parallelrechners beim gemeinsamen Start aller Prozesse der parallelen Anwendung unveränderbar festgelegt (MPI-2 läßt diese Einschränkung fallen). Damit unterstützt MPI

direkt das SPMD-Modell, das allgemeine Message-Passing-Modell kann aber auch realisiert werden, z.B. durch ein übergeordnetes Hauptprogramm (Master), welches unterschiedliche Unterprogramme (Slaves) ruft.

6.3.2 Nachrichtenaustausch

Die Hauptaufgabe einer Message-Passing-Umgebung ist das Versenden von Nachrichten. MPI kennt eine Reihe von (anfänglich verwirrenden) unterschiedlichen Kommunikations-Formen und -Modi (siehe 6.1.3), die sich großenteils auf die Pufferung und Synchronisation beziehen (die im Folgenden verwendeten Namen der Routinen sind für das Fortran-Language-Binding).

Der Kommunikationsmodus von Punkt-zu-Punkt-Kommunikationsaufrufen wird in MPI durch den Aufruf unterschiedlicher Senderoutinen festgelegt. Wir betrachten hier zunächst die *blockierenden* Formen.

Das sogenannte *Standard-Send* (`MPI_SEND`) kann nun implementationsabhängig (leider!) *synchron* oder *gepuffert* sein. Das gepufferte Send (`MPI_BSEND`) kehrt sofort zurück, sobald die Daten im Sendepuffer sind, ist also unsicher und kann auch fehlschlagen, wenn keine Netzpuffer mehr frei sind (der Puffer wird implizit beim Aufruf angelegt). Das synchrone Send (`MPI_SSEND`) blockiert, bis die Daten beim Empfänger sind, ist also sicher, aber unter Umständen nicht performant. Es gibt noch weitere spezielle Sende-Aufrufe, auf die wir aber im Rahmen dieser Einführung nicht eingehen können.

Der Empfang einer Nachricht geschieht durch einen Aufruf von `MPI_RECV` (nur *ein* Modus vorhanden). Durch den Zwang, beim Sender- und beim Empfängerprozeß jeweils passend eine Sende- und eine Empfangsroutine zu rufen, wird ein Programmcode schnell unübersichtlich und insbesondere bei SPMD-Programmierung unsymmetrisch. MPI bietet daher zusätzlich einen kombinierten Sende-/Empfangsaufruf (`MPI_SENDRECV`), der in gleichlautender Form auf der Sender- und auf der Empfängerseite eingesetzt werden kann.

Der Typ der gesendeten Daten wird im Aufruf festgelegt durch Verwendung der *MPI-Datentypen*. Als Basis-Typen sind eine Reihe von grundlegenden (sprachabhängigen) Typen definiert (z.B. `MPI_INTEGER`, `MPI_REAL`, `MPI_DOUBLE_PRECISION`, `MPI_CHARACTER`, etc.). MPI kennt aber auch mächtige Konstrukte zur Definition von *abgeleiteten Datentypen*.

Beim *nichtblockierendem* Austausch (z.B. nichtblockierendes synchrones Senden `MPI_ISSEND`) wird der Abschluß des Sendevorgangs in einem separaten Aufruf geprüft (`MPI_WAIT` oder `MPI_TEST`). Damit kann man Kommunikation mit Rechenarbeit überlagern, wenn die Rechenaufrufe zwischen dem Sende-Aufruf und dem Abschluß-Aufruf eingefügt werden. Ein `MPI_ISSEND` mit unmittelbar folgendem `MPI_WAIT` entspricht der blockierenden Form des synchronen Sendens (`MPI_SSEND`). Entsprechende Konstruktionen gibt es für den Empfang von Nachrichten.

6.3.3 Weitere Funktionen

Alle MPI-Kommunikationsaufrufe benötigen eine Information über die Gruppe der Prozesse, innerhalb derer der Aufruf stattfinden kann, den sogenannten *Communicator*. Alle MPI-Prozesse werden bei der *Initialisierung* (`MPI_INIT`) dem globalen Communicator `MPI_COMM_WORLD` zugeordnet und erhalten innerhalb dieser Gruppe eine fortlaufende Nummer, den *Rank*, der mit `MPI_COMM_RANK` abgefragt werden kann. Es können beliebig weitere Gruppen für Untermengen von Prozessen gebildet werden. Die Größe einer Gruppe (eines Communicators) kann mit `MPI_COMM_SIZE` abgefragt werden. Jeder MPI-Prozeß muß am Ende `MPI_FINALIZE` rufen, damit MPI geordnet beendet wird und aufgeräumt wird (z.B. Freigabe aller Pufferbereiche). Danach ist keine Re-Initialisierung von MPI möglich.

Neben den Aufrufen für Punkt-zu-Punkt-Kommunikation kennt MPI auch vielfältige Möglichkeiten für *kollektive Kommunikation*, diese erfolgen immer für alle Prozesse eines Communicators. Wichtige kollektive Funktionen sind die Synchronisation durch eine Barriere (`MPI_BARRIER`), das Senden an eine Prozeßgruppe (`MPI_BCAST`) sowie das Verteilen (`MPI_SCATTER`) und Einsammeln (`MPI_GATHER`) von

Datenfeldern, auch in Versionen "von-allen-an-alle" (`MPI_ALLGATHER, MPI_ALL_TO_ALL`). In MPI ist auch eine globale *Reduktionsoperation* definiert (`MPI_REDUCE`), mit der durch Angabe einer Reduktionsfunktion ein Ergebnis aus Daten berechnet werden kann, die über eine Gruppe von Prozessen verteilt sind. Neben etlichen vordefinierten *Reduktionsfunktionen* wie globale Summe, Produkt oder Extrema (`MPI_SUM, MPI_PROD, MPI_MIN, MPI_MAX`) gibt es logische Operationen und die Möglichkeit, eigene Operationen zu definieren, die auch auf die oben erwähnten eigenen Datentypen angewendet werden können.

Abschließend soll auch noch erwähnt werden, daß MPI auch *virtuelle Topologien* durch eine Reihe von Funktionen unterstützt, mit denen Prozeßtopologien erzeugt, manipuliert und abgefragt werden können. Dies fällt unter die Überschrift *"MPI is big"* und soll daher hier nicht weiter besprochen werden. Interessierte seien dazu auf die reichlich vorhandene Dokumentation zu MPI verwiesen.

Diese Dokumentation zu MPI findet man auf der MPI-Homepage[9] im WWW. Alle MPI-Routinen sind auch in den Manual-Pages beschrieben. Die einführende Manual-Page findet man unter "`man MPI`". Außerdem soll auch hier der Hinweis auf das *Internet Parallel Computing Archive*[10] beachtet werden.

6.3.4 Implementierung von MPI-Programmen

Bei der Implementierung eines MPI-Programms müssen eine Reihe von Punkten beachtet werden. MPI stellt eine Reihe von Definitionen in einem Include-File zur Verfügung, das in jede Routine eingebunden werden muß, die MPI-Aufrufe enthält. Das Includefile heißt `mpi.h` für C-Programme und `mpif.h` für Fortran-Programme. Die MPI-Routinen sind systematisch benannt: `MPI_*` für die C-Calls und die Fortran-Calls, wobei der MPI-Standard bei der Schreibweise zwischen reinen Großbuchstaben für Fortran und gemischter Schreibweise für C unterscheidet. Beim Link-Vorgang muß die MPI-Library mit der Option `-lmpi` eingebunden werden, ausserdem müssen die Suchpfade für die Include-Files und die Bibliotheken angegeben werden, z.B.:

```
cc -I/path/to/mpi/include/ -L/path/to/mpi/lib/ -lmpi myprog.c
```

Manche MPI-Implementierungen wie das `MPICH` vom *Argonne National Laboratory* erleichtern diesen Schritt, indem sie Shell-Scripts (z.B. `mpicc` und `mpif77`) zur Verfügung stellen, in denen sämtliche notwendigen Einstellungen bereits gemacht sind. Der obige Aufruf lautet dann

```
mpicc myprog.c
```

sofern man den Installationspfad von MPI im Suchpfad hat.
Im Kapitel 7 findet man ein einfaches, illustrierendes Beispiel für ein MPI-Programm.

6.3.5 Ausführen von MPI-Programmen

Die Ausführung vom MPI-Programmen ist recht einfach. Der Programmsuchpfad muß um den Pfad der MPI-Executables erweitert sein (Beispiel C-Shell und die MPICH-Installation im HFI-CIP-Pool):

```
setenv MPICH_ROOT /usr/shared/mpich
set path=( $path $MPICH_ROOT/bin )
```

Dann können `nproc` Prozesse mit dem Kommando

```
mpirun -np nproc a.out
```

gestartet werden. Weitere Hinweise findet man in der oben angegebenen Dokumentation zu MPI.

[9] Die MPI-Homepage im WWW: `http://www.mpi-forum.org/`
[10] Das Internet Parallel Computing Archive: `http://www.hensa.ac.uk/parallel/`

7 Beispiele

Nach diesen theoretischen Betrachtungen der Funktionalitäten von PVM und MPI sollen die nun folgenden Beispiele die Verwendung der grundlegenden Konstrukte in einem Programm (Fortran- und C-Versionen) demonstrieren. Das Programm ist so einfach, daß die Programmtexte selbsterklärend sein sollten.

Ein einfaches MPI-Programm in C:

```
#include "mpi.h"                              /* include MPI definitions   */
 int main(int argc, char *argv[]) {
     int        numprocs, myid, value;
     MPI_Status status;

     MPI_Init(&argc,&argv);                   /* Initialize MPI            */
     MPI_Comm_size(MPI_COMM_WORLD, &numprocs); /* number of processes       */
     MPI_Comm_rank(MPI_COMM_WORLD, &myid);     /* rank of this process      */
     if (myid == 0) {                         /* Am I process 0?           */
       value = 1;
       printf("Process 0 sending %d to process 1.\n",value);
       MPI_Send(&value, 1, MPI_INT, 1, 99,     /* Yes, send value to proc 1 */
                MPI_COMM_WORLD);               /* with message tag 99       */

     } else {
       MPI_Recv(&value, 1, MPI_INT, 0, 99,     /* No, receive one int from 0 */
                MPI_COMM_WORLD, &status);      /* with message tag 99        */
       printf("Process 1 received %d.\n",value);
     }
     MPI_Finalize();                          /* Clean up                  */
 }   /* main */
```

Das gleiche MPI-Programm in Fortran:

```
      PROGRAM main
      INCLUDE 'mpif.h'                !* Include the MPI definitions
      INTEGER err, myid, numprocs, value, status(MPI_STATUS_SIZE)

      CALL MPI_Init( err )            !* Initialize MPI
C Determine number of processes
      CALL MPI_Comm_size( MPI_COMM_WORLD, numprocs, err )
C Determine rank of this process
      CALL MPI_Comm_rank( MPI_COMM_WORLD, myid, err )

      IF (myid .EQ. 0) THEN           !* Am I process 0?
          value=1                     !* Yes: send msg w. tag 99 to proc. 1
          PRINT *,'Process 0 sending ',value,' to process 1'
          CALL MPI_SSend(value, 1, MPI_INTEGER, 1, 99,
     +                   MPI_COMM_WORLD, err)
      ELSE                            !* No: receive msg w. tag 99 from proc. 0
          CALL MPI_Recv(value, 1, MPI_INTEGER, 0,
     +                  99, MPI_COMM_WORLD, status, err)
          PRINT *,'Process 1 received ',value
      ENDIF

      CALL MPI_Finalize(err)          !* Clean up MPI
      END
```

Die Realisierung des Programms mit PVM (C-Version).

```
#include "pvm3.h"                       /* include PVM-definitions  */

int main(int argc, char *argv[]) {
    int        myid, mytid, histid, value, atid, atag, alen, err;

    /*myid  = pvm_get_PE(pvm_mytid());*/  /* The CRAY way             */
    mytid  = pvm_mytid();                 /* The PVM way              */
    myid = pvm_joingroup("pelist");       /* The PVM way              */
    err = pvm_barrier("pelist", 2) ;

    if (myid == 0) {                      /* Am I process 0?          */
      histid = pvm_gettid("pelist", 1);
      value = 1;
      printf("Process 0 sending %d to process 1.\n",value);
      pvm_psend(histid, 99, &value, 1, PVM_INT);
    } else {
      histid = pvm_gettid("pelist", 0);
      pvm_precv(histid, 99, &value, 1, PVM_INT, &atid, &atag, &alen);
      printf("Process 1 received %d.\n",value);
    }
    err = pvm_barrier("pelist", 2) ;
}   /* main */
```

Die Realisierung des Programms mit PVM (Fortran-Version):

```
      PROGRAM PVMBsp

      INCLUDE "fpvm3.h"
      INTEGER*4 MyID, MyTID, value, info, iatid,
     &          iatag, ialen, istat, HisTID
      CHARACTER*6 PEList

      CALL pvmfmytid(MyTID)             ! Enroll in PVM
      PEList='AllPEs'
      CALL PVMFGETPE( PEList, MyID )
      call pvmfbarrier( PEList, 2, istat )
      IF (MyID .EQ. 0) THEN             ! Am I process 0?
      CALL pvmfgettid( PEList, 1, HisTID )
         value=1                        ! Yes: send msg w tag 99 to proc 1
         CALL pvmfpsend( HisTID, 99, value, 1, INTEGER4, info )
         print *, 'Process 0 sending ',value,' to process 1'
      ELSE                              ! No: receive msg w tag 99 from proc 0
      CALL pvmfgettid( PEList, 0, HisTID )
         CALL pvmfprecv( HisTID, 99, value, 1, INTEGER4, IATID,
     &                   IATAG, IALEN, ISTAT )
         print *, 'Process 1 received ',value
      ENDIF
      call pvmfbarrier( PEList, 2, istat )
      END

      subroutine PVMFGETPE( Group, MyPE )
      CHARACTER*6 Group
      INTEGER MyPE
      call pvmfjoingroup( Group, MyPE )
      END
```


Das folgende Beispiel (nur die Fortran-PVM-Version) tut im Prinzip das gleiche, jedoch in einer Master-Slave-Konfiguration. Es sollte ebenfalls selbsterklärend sein. Das Programm `hello` initiiert eine Instanz von `hello_other`, welches eine Nachricht an `hello` sendet. Diese wird empfangen und ausgegeben.

Das Master-Programm:

```
      program hello
      implicit none
      include 'fpvm3.h'
c -------------------------------------------------------------
c Example fortran program illustrating the use of PVM 3
c -------------------------------------------------------------
      integer mytid, childtid, nproc, numt, info
      integer from, msgtag
      character*100 buffer
      character*8 arch
c
c --- Enroll task in PVM / Who am I ?
      call pvmfmytid( mytid )
      print 100, 'I am T', mytid
c
c --- One instance of hello_other program
      nproc = 1
c
c --- ANY configured machine is acceptable
      arch = '*'
c
c --- Start hello_other on SOME host
      call pvmfspawn( 'hello_other',PVMDEFAULT,arch,nproc,childtid,numt)
c
c --- Successful ?
      if ( numt .eq. 1 ) then
c     --- Messages from ANY task and with ANY tag are accepted
         from = -1
         msgtag = -1
c     --- Receive, unpack and print message
         call pvmfrecv( from, msgtag, info )
         call pvmfunpack( STRING, buffer, 100, 1, info )
         print 100, 'From T', childtid, ' I got: ', buffer
      else
c     --- Error occured
         print *, 'cannot start hello_other'
      endif
c
 100  format ( A, I8, A, A)
c
c --- Program finished; leave PVM before exiting
      call pvmfexit( info )
      stop
      end
```

Das Slave-Programm:

```
      program hello_other
      implicit none
      include 'fpvm3.h'
c -------------------------------------------------------------
c Example fortran program illustrating the use of PVM 3
c -------------------------------------------------------------
      integer mytid, parenttid, msgtag, bufid, info
      character*100 buffer
      character*8 arch
c
c --- Enroll task in PVM / Who am I ?
      call pvmfmytid( mytid )
c
c --- Who is my daddy ?
      call pvmfparent( parenttid )
c
c --- Prepare and pack buffer
      buffer =  'Hello world !!!'
      call pvmfinitsend( PVMDEFAULT, bufid )
      call pvmfpack( STRING, buffer, 100, 1, bufid )
c
c --- Set msgtag to something
      msgtag = 1
c --- Send message to parent
      call pvmfsend( parenttid, msgtag, bufid )
c
c --- Program finished; leave PVM before exiting
      call pvmfexit( info )
      stop
      end
```


8 Abschließende Betrachtung und Ausblick

Es kann gesagt werden, daß die in (1.2) angeführten Vorteile der Parallel-Computer sich in der Tat realisieren lassen. Allerdings gibt es das Preis-/Leistungsverhältnis betreffend zwei sehr unterschiedliche Trends. Einerseits erfreut sich die Anwendung von Systemen, die ohne Mehrkosten verfügbar sind (wie PVM und MPI) einer immer größeren Beliebtheit, andererseits kommen (wie z.B. am Konrad-Zuse-Zentrum, Berlin) vermehrt auch Massiv-Parallel-Rechner auf, die zwar viel Geld kosten, sich aber auch auf dem neuesten technischen Stand befinden und eine enorm hohe Rechen- und Gesamt-System-Leistung bieten. Die besten Möglichkeiten ergeben sich aber durch das Zusammenspiel von beiden: Programme können auf Netzwerken entwickelt werden und müssen dann nur für die wirkliche Arbeit die Rechenleistung der Superrechner beanspruchen. Dadurch bleibt mehr Rechenzeit für weitere Anwender zur Verfügung. Die Voraussetzung dafür ist die Portierbarkeit, die bei PVM und MPI heute gegeben ist.

Das größte Problem der Parallelisierung ist aber immer noch die erwähnte Komplexität in der Programmentwicklung. Selbständig parallelisierende Betriebssysteme, die es schaffen ohne größere Verluste bei der Rechenleistung Programme (oder Teile davon) automatisch zu parallelisieren, gibt es leider noch nicht. Dazu kommt noch, daß für praktisch alle Anwendungen auf sequentiellen Computern bereits sehr umfangreiche Programmbibliotheken existieren.

Es kommen jedoch auch immer mehr parallele Anwendungen auf den Markt, die oft große Vorteile gegenüber ihren sequentiellen Konkurrenten haben, und auch die fehlende Standardisierung bei Parallel-Computern, die noch vor wenigen Jahren eine Verbreitung des Parallel-Konzeptes verhinderte, ist erfolgreich in Angriff genommen worden. Daher ist zu erwarten, daß der Zuspruch für die Parallelität in Zukunft noch sehr viel größer wird und sich schließlich auch auf der Anwenderschicht durchsetzt.