Seminar

Parallele Programmierung

Universität Marburg
Wintersemester 2005/06
Spaska Forteva

Inhaltverzeichnis:

1 Einführung

Die parallele Programmierung existiert seit mehr als 20 Jahren. Ursprünglich wurde von CDC6600 und IBM360/91 entwickelt. In den Jahren seitdem hat die parallele Computerwissenschaft in traditionellen Gebieten, wie Wissenschaft, Technik, Finanzen und neuen Anwendungsgebieten geschafft, komplizierte Probleme zu lösen, und Hochleistungsanwendungen zu erreichen. Trotz einigen Erfolge und dem versprechenden Anfang, hat paralleler Computer eine führende Stelle in der Informatik und Computertechnologie nicht gewonnen. Nur einen kleinen Prozentsatz wurde über Jahre verkauft.
Die parallele Computerwissenschaft gestaltet aber eine radikale Schicht der parallelen Programmierung in der Perspektive. Deswegen ist es vielleicht nicht überraschend, dass sie eine zentrale Rolle in den praktischen Anwendungen der Computerwissenschaft spielt. In den letzten 20 Jahren gab es Schwankungen zwischen wildem Optimismus ("für jede Frage, Parallelismus ist die Antwort"), und extrem Pessimismus ("Parallelismus ist ein Neigen Nische-Markt"). Auf diesem Grund ist es vielleicht der richtige Zeitpunkt für die Computerwissenschaft den Zustand der parallelen Programmierung zu untersuchen.
Wir beginnen die Diskussion in dem **Kapitel 1** mit der Frage, warum die parallele Computerwissenschaft eine gute Idee ist, und warum sie gescheitert hat, statt eine zentral Rolle zu spielen. Im **Kapitel 2** prüfen wir einige grundlegende Aspekte von parallelen Computern und Software. Im **Kapitel 3**wird das Konzept des parallelen Models und eine Liste mit einigen Eigenschaften, die Modelle der parallelen Programmierung haben sollten, diskutiert. Im **Kapitel 4** wird eine Übersicht über das breite Spektrum der vorhandenen parallelen Programmierungsmodelle.

2 Warum parallel

2.1 Vorteile

Hier einige Gründe, warum Parallelität ein Top-Thema geworden ist.

- Die reale Welt ist von Natur aus parallel, so ist das natürlich und richtig eine Berechnung über die echte Welt auf eine parallele Weise, oder mindestens auf einen Weg der Parallelität zu machen.

- Parallele Rechner besitzen eine höhere Leistung als einen sequentiellen Rechner
- Es gibt eine Begrenzung bei den sequenziellen Computermodellen, die aus

fundamentalen physischen Grenzen wie die Geschwindigkeit des Lichtes entstehen. Aus diesem Grund sind die Leistungen eines einzelnen Prozessors begrenzt.

- Selbst, wenn die Geschwindigkeit des Prozessors verbessert werden könnte, sind parallele Rechner für einige Anwendungen kosteneffektiver als einen Uni-Prozessor-Rechner.

2.2 Nachteile

Es gibt einige Schwierigkeiten und Nachteile der parallelen Programmierung:

- Als erstens ist unser menschliche bewusste Denken, das uns sequentiell erscheint. Das Ergebnis, wir verstehen Parallelität schwer.

- Zweitens, die Theorie der parallelen Computerindustrie ist noch unreifend und wurde weniger als denn die Theorie sequentielle Berechnung entwickelt. Wir wissen noch nicht viel über abstrakte Repräsentationen Logik und parallele Algorithmen, die effektiv genug für eine reale Architektur sind.

- Drittens, es wird lange Zeit dauernd, bis man die nötige Balance zwischen der unterschiedlichen Teile der parallelen Rechner erreicht, und den Beeinfluss dieser Balance der allgemeinen Performance untersucht. Sehr genau soll der Relationschip zwischen der Prozessorsgeschwindigkeit und innerer Kommunikationsleistung kontrolliert werden, um eine bessere Rechnerleistung zu erreichen.

- Viertens, die Parallelcomputerindustrie hat parallele Modelle mehr für ihren eigenen Markt entwickelt statt für den größten kommerziellen Markt. Dieser Markt war eigentlich klein und tendenzorientiert zu Military Anwendungen. Die Meinung über parallele Programmierung war, dass Parallele Computer teuer sind, denn sie kaum verkauft worden sind, und dass es einen Rick für den Hersteller und Benutzersein könnte. Das alles hat weiter den Enthusiasmus parallel zu Programmieren gedämpft.

- Die Ausführungszeit eines sequenziellen Programms ist ein konstanter Faktor, wenn es von einem schnellen Prozessor ausgeführt wird. Das ist nicht der Fall bei einem parallelen Programm, da es immer einen sequentiellen Anteil existiert, der nach Amdahls Gesetz die Effizienz beschränkt.

3 Basiskonzepten der Parallelität

Parallel Computer bestehen aus drei Basisblocken:

- Prozessoren
- Speichermodulen
- Systemskommunikationsnetz

3.1 Architekturen

MIMD – Architekturen

In der MIMD – Architektur hat jeder Prozessor seinen eigenen lokalen Speicher und Zugriff auf einen (gemeinsamen oder verteilten) Datenspeicher. Ein Verarbeitungsschritt besteht darin, dass jeder Prozessor eine separate Instruktion aus dem Datenspeicher lädt und ein eventuell errechnetes Ergebnis in den Datenspeicher zurückschreibt. So können die Prozessoren asynchron arbeiten, d.h. jeder Prozessor bearbeitet seine eigene Befehlsfolge unabhängig von den anderen Prozessoren ab. Erst beim Austausch von Zwischenergebnissen wird alles synchronisiert. MIMD-Modell hat hohe Flexibilität aber schwierige Programmierung.

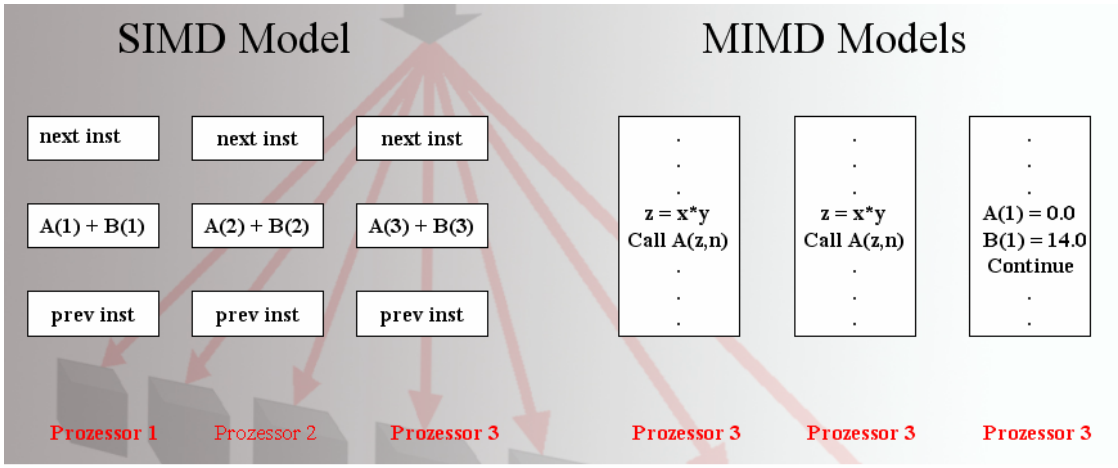


Abbildung 1.

Speicher-Architekturen

Im Falle eines verteilten Datenspeichers spricht man von einer **Distributed-Memory Architectur**. Jeder Prozessor hat eigenen Speicherbereich. Die Synchronisation der Daten wird durch Message Passing erreichen.

Nachteile:

- Datenstrukturen zerlegen
- Overhead bei Kommunikation
- Benutzer muss Kommunikation programmieren

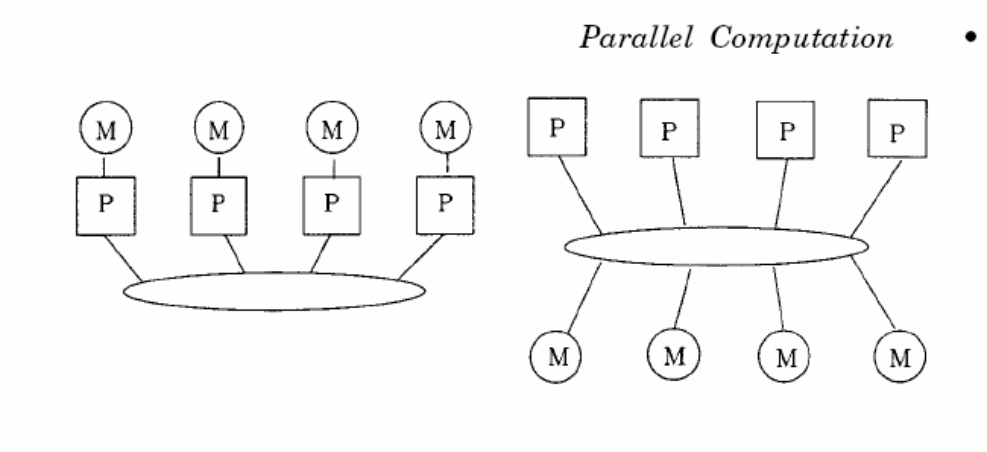


Abbildung 2. Distributed-memory MIMD und shared-memory MIMD

Im Falle eines gemeinsamen Datenspeichers spricht man von **Shared-Memory-Architectur**. Nur ein Prozessor kann in bestimmten Zeitpunkt auf gemeinsamen Speicher zugreifen.
Vorteil: schnelles Datensharing, einfache Nutzung.
Nachteil: Brandbreite ist begrenz, Bussystem aufwändig und teuer.

3.2 Parallele Software

Wir wenden uns jetzt zu der Terminologie der parallelen Software. Die Codeausführung von einem einzelnen Prozessor eines parallelen Computers ist in einer Umgebung, die ziemlich ähnlich zu dieser ist, die in einem Multiprogramm single-Prozessorssystem angewendet wird. So sprechen wir von Prozessen oder Tasks, um das Codeausführen in einer von Betriebssystem geschützter Speicherregion, zu beschreiben. Weil es die Kommunikation eines Prozessors mit anderen entfernten Prozessoren oder Speichers, Zeit nimmt, führen die meisten Prozessoren mehr als ein Prozess aus. So alle Standardtechniken der Multiprogrammierung anwenden: Prozesse bekommen descheduled, wenn sie tun Sie etwas durch eine entfernte Kommunikation, und werden für die Ausführung vorbereitet, wenn es eine passende Antwort erhalten wird. Eine nützliche Unterscheidung zwischen der virtuellen Parallelität eines Programms, das ist die Anzahl der logisch unabhängigen Prozesse, die es enthält, und der physikalischen Parallelität - die Anzahl von Prozessen, die gleichzeitig aktiv sein können (der, natürlich, gleich der Zahl der Prozessoren in des

parallelen Computers). Wegen der Kommunikationsaktionen, die in einem typischen parallelen Programm vorkommen, werden Prozesse öfter unterbrochen, als in einer sequenziellen Umgebung. Prozess-Manipulation ist teuer in einer multiprogrammierten Umgebung, deswegen immer mehr, parallele Computer verwenden lieber Threads statt Prozesse. Threads haben nicht eigenen von Betriebssystem geschützten Speicher.

3.3 Kommunikation

Die Prozesskommunikation hängt von der Architektur ab. Es gibt drei Haupttypen:

- **Message passing**

Der Sending-Prozess verpakt die Nachricht mit einem Kopf- header. Er zeigt, zu welchem Prozessor und Prozess die Daten übergeben werden müssen, und fügt sie in die Verbindungsnetz. Sobald die Nachricht abgeschickt worden ist, kann der Senden-Prozess weitermachen. Sender und Empfänger treten immer als Paar ein. Diese Art-Sendung wird **nonblocking send** genannt. Der Empfang-Prozess muss wissen, dass Daten zu ihm geschickt wurden. Wenn die erwarteten Daten noch nicht angekommen sind, suspendiert (blockiert) der Empfang-Prozess bis sie kommen. Message passing wird auf Distributet-Memory-Architectur angewendet.

- **Transfer through shared Memory**

Der sending-Prozess speichert die Werte in einen Speicherbereich, und der Empfang-Prozess kann sie dort lesen. Es ist schwierig, zu entdecken, wann einen Wert gelegt oder entfernt wurde. Standardbetriebssystemtechniken wie Semaphore könnten zu diesem Zweck verwendet werden. Jedoch ist das teuer und kompliziert die Programmierung.

- **Direct remote-memory access**

Immer mehr verteilte Speicher-Architekturen benutzen ein Paar von Prozessoren in jedem Prozess-Element. Ein Anwendungsprozessor macht die Ausführung des Programms und ein andere sog. Nachrichtenübermittlungsprozessor handelt den Griff-Verkehr zu und von dem Netz. Das hat der Vorteil, dass der Anwendungsprozessor ohne Unterbrechung weiter rechnen kann. Das ist eine Mischform der Kommunikation, in der verteilte Speicher Architektur angewendet wird.

3.4 Verbindungsnetz

Das Verbindungsnetz eines parallelen Rechners ist ein Mechanismus, der die Prozessoren mit einander und mit Speichermodulen zu kommunizieren erlaubt. Die Topologe dieses Netzes ist die komplette Einordnung von individuellen Verbindungen zwischen den prozessierten Elementen. Das ist natürlich repräsentiert als ein Graph. Die worst-case Wartezeit des Netzes ist die längste benötigte Zeit, um ein Paar von Prozessoren zu kommunizieren zu bringen. Im Allgemeinen, beobachtete Latenz ist größer als das, das durch die Topologie einbezogen wird. Das ist wegen der Verkehrsstauung im Netz selbst.
Die Leistung eines parallelen Programms ist gewöhnlich in Bezug mit seiner Ausführungszeit. Das hängt von der Geschwindigkeit des individuellen Prozessors ab, aber auch von der Einordnung der Kommunikation und der Fähigkeit des Verbindungsnetzes. Wenn man von den Kosten eines Programms sprechen würde, wäre gewöhnlich von seinen Ausführungszeitkosten. Jedoch, im Zusammenhang mit der Softwareentwicklung können notwendige Kosten entwickelt werden.

4 Modelle und ihre Eigenschaften

Ein Modell der parallelen Berechnung ist eine Schnittstelle zwischen der Programmierebene (high-level) und Rechnerarchitektur (low-level). Konkreter ist ein Modell eine abstrakte Maschine, die bestimmte Operationen dem Programmierniveau zur Verfügung stellt, und eine Implementierung dieser Operationen auf allen Architekturen nach unten ausführt.

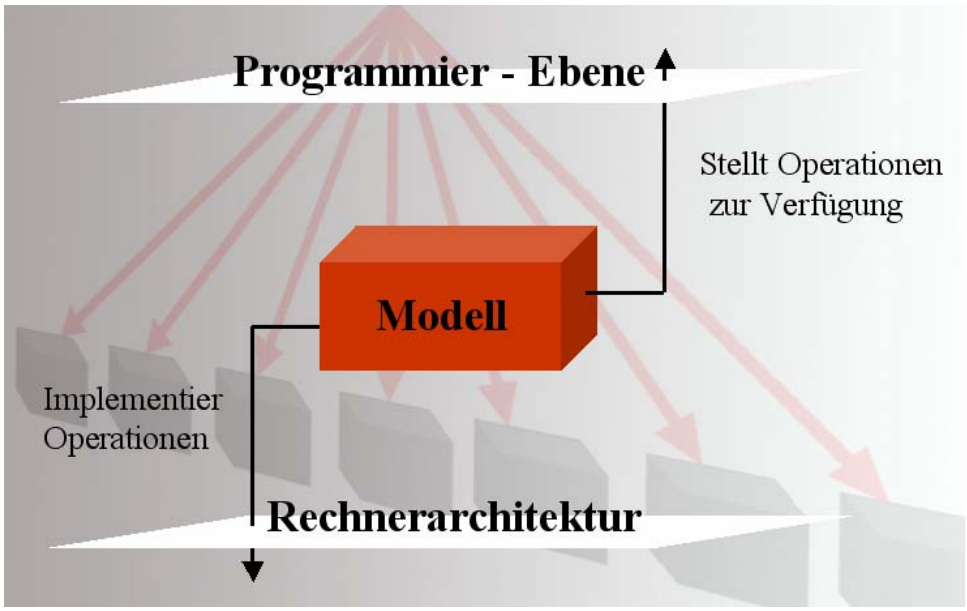


Abbildung 3.

Da ein Modell gerade eine abstrakte Maschine ist, gibt es verschieden Niveaus der Abstraktion. Zum Beispiel ist jede Programmiersprache ein Modell in dem Sinn, dass sie vereinfachte Ansicht von der zu Grunde liegenden Hardware zur Verfügung stellt. Das macht hart Modelle, abhängig von dem Niveau der Abstraktion ordentlich zu vergleichen. Es gibt sogar keine Isomorphe (one to one)-Verbindung zwischen Modellen.
Ein paralleles Programm ist ein extrem komplexes Objekt. Sei gegeben ein Programm, das auf einem 100-Processorsystem läuft, groß aber ziemlich üblich heute. D.h., dass es 100 aktive Threads in jedem Moment gibt. Um die Latenz der Kommunikation und Speicherzugang(zugriff) zu verbergen, ist jeder Prozessor in Verbindung gleichzeitig mit mehreren Threads, es folgt, die Anzahl der aktiven Threads ist mehrere mall größer (sagen wir 300). Jedes Thread kann mit einigen anderen Threads kommunizieren, und seine Kommunikation kann asynchron oder mit einem Synchronisationsverweis auf einem Thread sein. Es folgt, dass es 300^2 mögliche Interaktionen "im Gange" in jedem Moment geben konnte. Der Zustand solches Programms ist sehr groß. Das Programm das auf diese Ausführungsweise bewirkt, muss bedeutsam mehr abstrakt sein als die Beschreibungsweise selbst.
Das deutet, dass Modelle der parallelen Berechnung viel höher, als für die sequenzielle Programmierung, Niveaus der Abstraktion verlangen. Es ist noch (gerade) denkbar, große sequenzielle Programme im Assembly-Code zu implementieren, obwohl die neuesten sequenziellen Architekturen das immer schwieriger machen. Es ist wahrscheinlich unmöglich große MIMD Programm für 100 Prozessoren im Asembly-Code und kosteneffektiv zu schreiben. Außerdem, die detailirre Ausführung eines partikularen Programms auf die Architektur eines Stils wird wahrscheinlich sehr verschieden von der ausführlichen Ausführung auf einem anderen. So sind die Abstraktionen, die Unterschiede zwischen der Architektur Familien verbergen, notwendig.
Andererseits, ein abstraktes Modell ist nicht vom großen praktischen Interesse, wenn eine effiziente Methode, mit der Programme implementiert und durchzuführen konnten, nicht gefunden ist. So müssen Modelle nicht nur abstrakt, sondern auch intellektuell sein.

4.1 Anforderungen an den Modelle

Um ein Modell nützlich zu sein, sollte es beide Abstraktion und Effektivität ansprechen. Ein gutes paralleles Modell muss folgende Eigenschaften besitzen:

4.1.1 Leicht zu programmieren

Denn ein ausführbares Programm ein kompliziertes Objekt ist, muss ein Modell die meisten Details von den Programmierern verstecken. Soviel wie möglich sollte eine exakte Struktur

des Programms durch die Übersetzungseinheit (Compiler und Laufzeitsystem) anstatt durch den Programmierer eingesetzt werden. Das bedeutet für das Modell:

- **Dekomposition** eines Programms in parallelen Threads.

Das Programm wird in kleinen Stücken zerlegt, die auf verschiedenen Prozessoren ausführt werden. Das erfordert auch Separation der Code- und der Datenstruktur in eine möglicherweise große Anzahl von Stücken.

- **Mapping** der Threads zu den Prozessoren.

Sobald das Programm in Stücken zerlegt worden ist, muss eine Wahl getroffen werden, und zwar welche Stück auf welchem Prozessor eingesetzt wird. Die Platzierungsentscheidung wird häufig durch die Kommunikationsverbindung beeinfluss. Stücke die oft in Verbindung stehen, sollten auf Prozessoren ausgeführt werden, die näher zu einander in dem Kommunikationsnetz sind.

- **Kommunikation** der Threads.

Ihre genaue Form hängt viel von der aktuellen Architektur ab. Die Prozesse an beiden Enden müssen suspendiert werden, falls sie auf Daten warten, die nie kommen. Der Art der Kommunikation ist kompliziert. Das empfangende Ende und der Absender müssen gegen eine Blockierung vorbereitet sein. Das heiß, dass eine Kette von Kommunikationen unter n Prozessen durch eine Aktion eines einzelnen zusätzlichen Prozesses in Deadlock-Zyklus umgewandelt werden kann (eine empfangende Befehl des letztes Prozess vom dem ersten zu erhalten).

- **Synchronisation** der Threads.

Währen der Ausführung des Programms gibt es Momenten, wenn ein Paar Threads oder sogar eine große Gruppe wissen müssen, dass sie gemeinsam einen allgemeinen Zustand erreicht haben. Wieder ist die exakte Mechanismuseinheit ziel-abhängig, und es gibt enormes Potenzial für Deadlocks in der Interaktion zwischen Kommunikation und Synchronisierung.

Die Erfahrung hat gezeigt, dass Deadlocks in den parallelen Programmen in der Tat einfach zu verursachen und schwer zu entfernen sind. Deswegen sollten die Modelle soviel wie möglich abstrakt und einfach sein. Für viele Programme dies bedeutet, dass Parallelität nicht ausdrücklich im Programmiertext gebildet wird.

4.1.2 Software- Entwicklungsmethode.

Die vorgehende Anforderung deutet einen großen Abstand zwischen den Informationen, die vom Programmierer über die semantische Struktur des Programms bereitgestellt

werden, und der detaillierten Struktur der Ausführung. Das Modell sollte eine feste semantische Grundlage haben, auf dem die Transformationstechniken des Compilers angewendet werden können. Durch diese semantische Grundlage muss den großen Abstand zwischen Spezifikationen und Programmen adressiert werden.

4.1.3 Architektur-unabhängig

Das Modell sollte architekturen-unabhängig sein, damit man Programme von parallelem Computer zu parallelem Computer portieren kann, ohne in einer nicht trivialen Weise neu entwickeln zu müssen, oder geändert zu werden. Diese Anforderung ermöglicht eine weit verbreitete Softwarenindustrie für parallelen Rechner. Rechnerarchitektur hat wegen immer höherer Geschwindigkeit des Prozessors, die heutige Computertechnologie erreicht, verhältnismäßig kurze Leben. Benutzer des parallelen Rechners müssen vorbereitet sein, ihre Computer jedes fünftes Jahr zu ersetzen. Redentwicklung der Softwaren ist nicht kosteneffektiv, obwohl heute zu Tage das der Fall ist. Deswegen um die parallele Computertechnologie nützeffektiv zu sein, muss Software von den Änderungen des zugrundlegenden Rechners isoliert werden, selbst wenn diese Änderungen erheblich sind. Diese Anforderung bedeutet, dass das Modell genug abstrakt sein soll, um von konkreten Merkmalen des parallelen Rechners unabhängig zu sein

4.1.4 Einfach zu verstehen.

Damit es möglich ist das, vorhandene Software zu studieren und eventuell weiter zu entwickeln soll ein Modell verstehendlich sein, und sein Strukturgrundlage soll leicht von anderen Menschen akzeptiert werden. Wenn Parallelecomputermodelle in der Lage sind, die Komplexität zu verstecken und eine einfache Schnittstelle anzubitten, haben sie eine große Chance verwendet zu werden. Generales Ziel: bedienungsfreundlich Tools mit freier Zielwahl.

4.1.5 Garantierte Performance

Ein Modell sollte eine gute Leistung über Vielzahl von verschiedenen parallelen Architekturen garantieren. Das bedeutet nicht, dass die Implementierung die maximale Kapazität der Architektur ausnutzen muss. Die Implementierung soll, für die meisten Probleme nicht notwendig eine maximal höhere Leistung der Architektur erreichen. Ziel der Implementierung ist die offensichtliche Software-Kompliziertheit zu „konservieren“ und konstant zu halten. Es gibt Begrenzungen der parallelen Architektur, und zwar diese hängen von ihren Kommunikationseigenschaften ab. Eine leistungsfähige Architektur sollte eine willkürliche Berechnung mit der erwarteten Leistung durchführen können.

4.1.6 Geringe Kosten

Design eines Programms wird mehr oder weniger durch die Leistung ausgedrückt. Die Ausführungszeit ist sehr wichtig, aber auch wichtig ist Kostenentwicklung. Wir beschreiben beide als Kosten des Programms. Die Interaktion der Kostenmasse mit dem Designprozess im sequenziellen Softwarenkonstruktion ist verhältnismäßig einfach. Im Gegensatz zu sequentieller Software ist die Kostenmasse bei paralller Software schwer einzuschätzen, da schon kleine Änderungen im Programmtext und die Wahl des Rechners sich maßgeblich auf die Kosten eines Programms auswirken.
Der Aufbauprozess hat zwei Phasen:

- Entscheidungen werden über Algorithmen getroffen
- die asymptotischen Kosten des Programms werden beeinflusst

5 Übersicht über Modelle.(Baum-Repräsentaton)

Die Modelle werden abhängig von deren Grad der Abstraktion abgestuft. Siehe Abbildung4.

Stufen der Parallelisierung

- **Dekomposition Zerlegung des Programms in Stücke (Threads)**
- **Mapping Zuteilung der Threads auf die Prozessoren**
- **Kommunikation zwischen den Threads (Datenaustausch)**
- **Synchronisation: Möglich wenig Potenzial für Deadlocks**

5.1 Abstraktionsgrad 1 (Nichts explizit)

Das Model stellt alle Anforderungen für eine parallele Abarbeitung des Programms zur Verfügung, das heißt Zerlegung, Verteilung, Kommunikation und Synchronisation werden vom Model übernommen. Das sind Modelle, die sich von der Parallelität vollständig abstrahieren. Solche Modelle beschreiben nur den Zweck des Programms, nicht wie diesen Zweck erziehen soll. Die Softwarenentwickler interessieren sich nicht dafür, ob das Programm, das sie konstruieren, parallel oder sequentiell ausgeführt wird. Solche Modelle sind notwendigerweise abstrakt und verhältnismäßig einfach, denn das Programm nicht komplizierter als ein solche sequenzielles ist. Die besten Modelle der parallelen Rechner sind für die Programmierer diese, in denen sie Parallelität an allen nicht zu berücksichtigen brauchen. Eine Versteckung aller Aktionen, die eine parallele Berechnung erfordern, bedeutet, dass Software-Entwickler ihre vorhandenen Techniken und Fähigkeiten für eine sequenzielle Softwareentwicklung übertragen. Selbstverständlich sind solche Modelle

notwendigerweise abstrakt, was der Job der Entwickler schwierig macht. Weil die Transformation, Kompilation und Laufzeitsystem sich in der Struktur des eventuellen Programms schließen. Das bedeutet zu entscheiden, wie die spezifizierte Berechnung ausgeführt werden soll;

- Parallelisierende Compiler
- Funktionale Programmiersprachen

Auf einmal wurde es geglaubt, dass automatische Implementierung eines abstrakten Programms mit Hilfe einer gewöhnlichen imperativen Sprache gemacht werden könnte. Deswegen wurde viel Arbeit investiert, den Compiler zu parallelisieren, und festgestellt, dass ein hoch automatisierter Übersetzungsprozess nur dann praktisch ist, wenn er von einem vorsichtig gewählten Modell startet, das abstrakt und aussagekräftig ist.
Innerhalb dieser Kategorien präsentieren wir Modelle abhängig ihrer Grad der Steuerung über Kommunikation und Struktur.

5.1.1 Dynamische Struktur

Eine populäre Methode (Weg) die Computer-Berechnung in einem deklarativen Weg zu beschreiben, in dem das gewünschte Resultat spezifiziert wird, ohne zu sagen, wie das Resultat gerechnet werden soll, ist eine Menge von Funktionen und Gleichungen anzuwenden.
Funktionen können an andere Funktionen übergeben werden und Rückgabe sein. Bei den funktionalen Programmiersprachen gibt es keine Seiteneffekte. Es folgt, dass das Ergebnis nur von den eingegebenen Werten abhängt. Der Vorteil für die parallele Verarbeitung besteht darin, dass die Argumente der Funktionen parallel ausgewertet werden. Sie können ohne Änderung des Ergebnisses parallel zueinander ausgewertet werden.
Higher-order fuktional programming behandelt Funktionen als λ-Terms und berechnet ihre Werten mit einer Benutzung der Reduktion in λ-Kalküls. Ein Beispiel solcher Sprache ist Haskell [Hudak und Fasel 1992]. Haskell schließt auch einige typische Merkmale der funktionalen Programmierung, wie Benutzer definierte Typen, „faule“ Auswertung, Modellanpassung. Außerdem hat Haskell eine parallele Funktion -I/O System und liefert einen Modul-Service. Die aktuelle Technik, die von funktionalen Sprachen für das Berechnen der Funktionswerten Benutzt wird, heißt **hight-reduktion** [Peyton-Jones and Lester 1992]. Funktionen werden als Bäumen, mit allgemeinen Unterbäumen für geteilte Unterfunktionen repräsentiert. Der Berechnungsregel selektiert die Graphsunterstrukturen, reduziert diese auf einfachen Formen und wider ersetzt sie in der großen Graphstruktur. Wenn es keine weitere Berechnungsregel angewendet werden könnten, ist der resultierte Graph das Resultat der Berechnung. Es ist leicht zu sehen, wie die Graphreduktionsmethode in prinzipiellen Regeln parallelisiert werden kann, und auf nicht überlappende Sektionen

von dem Graphen selbständig und folglich gleichzeitig angewendet werden.
So können Multiprozessoren nach reduzierbaren Teilen des Diagrams in einer unabhängigen Weise suchen, die abhängig nur von der Struktur des Graphen sind. Z.B. ,wenn der Ausdruck (exp1*exp2), wo exp1 und exp2 willkürliche Ausdrücke sind, ausgewertet werden soll, können zwei Threads unabhängig exp1 und exp2 auswerten, damit man beide Werte gleichzeitig bekommt.
Leider gibt es Nachteile bei dieser einfachen Idee. Die Abhängigkeit von Eingabewerten führt dazu, dass bestimmte Argumente gar nicht auswertet werden müssen oder nicht terminieren.
Z.B. haben die meisten funktionalen Sprachen irgendeine Form:

```
if b(x) then
            f(x)
    else
            g(x)
```

Offenbar ist genau einer der Werte von f(x) oder von g(x) erforderlich, aber man weißt nicht welches, bis der Wert von b(x) unbekannt ist. Die Auswertung von b(x) verhindert eine überflüssige Arbeit, aber verlängert den Pfad der Berechnung. Wertet man f(x) und g(x) spekulativ aus, könnte ein Ausdruck unter Umständen nicht terminieren. Beispiel, wen f(x) nur für Werte von x terminiert, für die b(x) true ist. Es ist kompliziert unabhängigen Programmteilen zu finden, mit denen eine erwartete Berechnung bis zum finalen Ergebnis ausgeführt werden kann, ohne eine anspruchsvolle Analysis des Programms als ganz. Parallelegraphenreduktion war ein begrenzter Erfolg auf einen Rechner mit share-memory-Architektur. Ob sie auch für Verteiltenspeicher (distributed-memory) Rechner geeignet ist, ist es noch nicht klar.
Solche Modelle sind einfach und erlauben Software-Entwicklung mit Transformation, aber sie garantieren keine Performance.

Concurrent rewriting ist eine Methode, mit der die Regeln für Überschreiben den Programmstücken werden durch eine andere Weise gewählt. Noch einmal, Programme sind Termen, die ein gewünschtes Ergebnis beschreiben. Sie werden durch Wiederholung einer Menge von Regeln überschrieben, bis es keine weiteren Regeln angewendet werden können. Der resultierte Term ist das Ergebnis der Berechnung. Einige Beispiele solcher Modelle sind OBJ [Goguen and Winkler 1988; Goguen et al. 1993, 1994]. Diese sind funktionale Sprache, deren Semantik auf gleiche Logik basiert. Ein Beispiel, gibt die Eigenschaft dieser Methode. Das folgende Modul ist funktional und polynominal differenzierbar. Die Polynome repräsentieren die gewöhnlichen Aktionen. Die Zeilen, die mit eq beginnen, sind Regeln, die durch eine elementare Rechnerart übergeschrieben werden. Die Zeile, die mit ceq beginnen, sind bedingungsübergeschriebene Regeln.

```
Fmod POLYY-DER is
```

```
Protecting POLYNOMIAL.
op der : Var Poly -> Poly.
op der : Var Mon -> Poly.
var A : Int
var N : NzNat .
vars P Q : Poly .
vars U V : Mon .
eq der(P + Q) = der(P) + der(Q).
eq der(U . V) = (der(U) . V) + (U. der(V)).
eq der(A * U) = A* der(U) .
ceq der(X^N) = N * (X^(N – 1)) if N > 1 .
eq der(X^1= = 1 .
eq der(A) = 0.
Endfm
```

Ein Ausdruck

```
Der(X  ^ 5 + 3 * X ^ 4 + 7 * X ^2)
```

kann parallel berechnen, denn es gerade mehrere Plätze gibt, wo man Überschreiben anwenden kann. Diese einfache Idee kann benutzt werden, um mehrere andere parallele Computermodelle nachzubilden.
Modelle dieser Art sind einfach und abstrakt und erlauben Software-Entwicklung durch Transformation, aber genau wie vorige sie können nicht Performance garantieren, und diese Modele sind zu dynamisch, um Benutzungskostenmassen zu garantieren.

5.1.2 Statische Struktur

Abstrakten Programmen können aus vorbereiteten (Blocks) Bausteine, deren Implementierung vordefiniert war, eingebaut werden. Dieses Verfahren hat folgende Vorteile:

- Die fertigen Bausteine antreiben das Niveau der Abstraktion, denn sie die grundlegenden Einheiten sind, mit den die Programmierer arbeiten. Sie können eine willkürliche Menge von interner Komplexität verstecken.
- Die Bausteine können intern parallel sein, aber sie können sequenziell zusammengestellt werden. In dem Fall brauchen die Programmierer nicht zu beachten, dass sie intern parallel eigentlich sind.
- Die Implementierung jedes Bausteines braucht nur einmal für eine Architektur realisiert zu werden.

In dem Kontext von paralleler Programmierung werden solche Bausteine Skeletten genannt [Cole 1989] und sie unterstreichen eine Anzahl von wichtigen Modellen. Eine Eigenschaft

der Skelette ist, dass sie Kontrollstrukturen entkapseln.
Wir einschränken unsere Aufmerksamkeit auf algorithmischen Skeletten, die eingekapselte Kontrollstrukturen sind. Die Idee ist, dass jedes Skelett mit einem Standartalgorithmus oder Fragment korrespondiert und, dass diese Skelette man sequentiell ordnen(zusammensetzen) kann. Die Software-Entwickler selektieren die Skelette, die sie benutzen möchten, und setzen sie zusammen.

Algorithmische Skelette (Higher Order Functions)
Dem Benutzer wird ein Programmrahmen zur Verfügung gestellt. Diese Metafunktionen bekommen als Parameter Benutzerdefinierte Funktionen sowie Datenstrukturen für die ein - und Ausgabe. Die Skelette verkörpern Programmierparadigmen.
Beispiel : Divide&Conquer

```
: d&c  ::        (a-> Bool) -> (a->b) -> (a -> [a]) -> ([b] -> b) -> a -> b
  d&c             trivia        solve     divide     conquer        p
```

if Problem klein genug
then löse das Problem direkr
else
DIVIDE: zerlege das Problem, CONQUER :löse jedes Teilproblem rekursiv
SOLVE :berechne aus der Teillösungen die Gesamtlösung
Ablauf :
Schritt 1: übergebe das Problem dem Wurzel
Schritt 2: Falls das Problem zerlegbaràteilen
Schritt 3: wiederhole Schritt 2 rekursiv so lange bis die Teilprobleme direkt gelöscht werden können, oder bis der gewünschte Parallelitätsgrad.
Schritt 4: berechnen die Blätter und reiche die Ergebnisse an die Eltern.
Schritt 5: alle inneren Knoten, die Teillösungen von ihren Söhnen empfangen haben, vereinigen diese zu einer neuen Teillösung und geben diese dann an ihren Vater weiter.
Schritt 6: wiederhole Schritt 5 solange, bis die Gesamtlösung die Wurzel erreicht.

Wichtig:Es gibt verschieden Techniken das Berechnungsbaum zu gestallten, aber die Anzahl der Prozessoren hängt von der Anzahl der generierten Subprobleme ab.
(#Prozessoren=#Teillösungen)

Homomorphische Skelette
Eine Betrachtung der Berechnung und Kommunikation von homomorphischen Skeletten ist auf Abb.5 gezeigt.

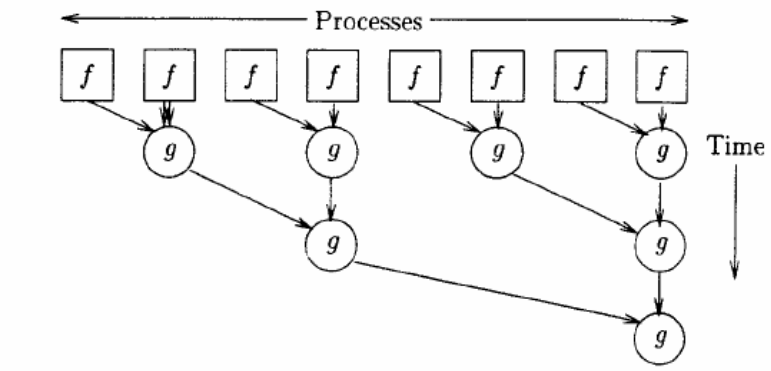


Abbildung 5.

Jede mögliche Liste Homomorphie kann berechnet werden, indem man passenden Parametern für f und g einsetzt.

Definition: Function h ist homomorphism falls es ein binar Operator + gibt, und für alle Listen xs und vs gilt:

$$h\ (xs\ ++\ ys)\ =\ h\ xs + h\ ys$$

Beispiel unterschiedlicher f und g:

sum $f = id$, $g = +$

Maximum $f = id$, g =binary max

length $f = 1$, $g = +$

sort $f = id$, g = sorted_merge

Der Compiler- oder Bibliothek- Schreiber wählt, wie jeder eingekapselte Algorithmus implementiert sollte, und wie intra und inter Parallelität der Skelette für jede mögliche Zielarchitektur organisiert ist. Jeder Prozess führt das gleiche Programm auf unterschiedlichen Daten aus.

Algorithmische Skelette sind einfach und abstrakt. Aber, weil das Programm als Komposition von Skeletten ausgeführt werden muss, ist die Ausdrucksfähigkeit der Programmiersprache eine offene Frage. Eine garantierte Performance ist möglich, falls die Skelette mit Vorsicht gewählt worden sind.

5.2 Abstraktionsgrad 2

Das sind Modelle, in denen Parallelität explizit gebildet wird, aber Dekomposition des Programms in Threads implizit ist. D.h., dass die Software-Entwickler sich darum nicht kümmern brauchen, wie das Programm in Stücken geteilt werden soll. Der Vorteil für den Compiler liegt darin, dass keine komplizierte Datenabhängigkeitsanalyse gemacht wird. In solchen Modellen beachten Softwarenentwickler wie viel Parallelität verwendet wird, um ein Potential in den Programmen ausgedrückt zu haben.

5.2.1 Dynamische Strukturen

Vertreter sind explizite logische Sprachen. Sie werden auch Konkurrentenlogischensprachen genannt. Beispiele von Sprachen in dieser Kategorie sind PARLOG [Gregory 1987], Dreieck-Einleitung [Pereira und Nasr 1984], gleichzeitige Einleitung [Shapiro 1986], GHC [Ueda 1985] und Faser [fördern Sie und Schneider 1990]. Prolog gilt als eine Programmiersprache der vierten Generation. Prolog kennt nur einen Grundtyp als Datenstruktur: den Term. Terme können dabei atomar, komplex oder variabel sein. Abhängig von dieser Interpretation kann man einen atomar Klausel <-C als einen Prozess, einen Konjunktive - Klausel <- C1, …, C als eine Prozess-Network, und eine logische Variable zwischen zwei Untertermen als Kommunikationskanal zwischen zwei

Beispiel PARLOG

H <- G1, G2, …, Gn | B1, B2, …, Bm. N, m>=0,

Wo H Klauselnkopf ist, die Menge Gi der Guard ist, und Bi der Körper der Klauseln ist. Un das Symbol „|“Konjunktion zwischen Gi und Bi ist.

H ist true, wenn die beide Gi und Bi true sind. Um H zu lösen, abhängig von der Prozessinterpretation, ist es nötig den Guard Gi zu lösen. Falls die Lösung erfolgreich war, erst dann kann man B1, B2, …, Bm parallel lösen.

Diese Sprachen fordern, dass die Parallelität von Programmierern explizit mit Hilfe von Anmerkungen welche Klausen parallel ausgeführt können, spezifiziert wird.

5.2.2 Statische Strukturen

Für Modelle mit statischer Struktur, erscheint das Skelettkonzept noch einmal, aber dieses mal, wird das um einzelne Datenstrukturen gegründet

Wir betrachten Fortran mit der Erweiterung einer ForAll Schleife, in der die Iterationen der Schleife unabhängig sind und gleichzeitig von verschiedenen Prozessoren ausgeführt werden können. Beispielweise besitzt Fortran die Erweiterung einer ForAll-Schleife, die zu einer Vektoranweisung äquivalent ist und somit parallel berechnet werden kann.

```
ForALL ( I = 1:N, J = 1:M)
            A(I,J) = I * B(J)
```

Falls die Schleife mehrere Anweisungen enthält, können diese nicht parallel zu einander ausgewertet werden. Jede Anweisung muss dann vollständig abgearbeitet werden, bevor man mit der nächsten Anweisung beginnt.
Man muss aufpassen, die Schleifen nicht die gleiche Lokation zu referenzirren. Das kann nicht im allgemeine automatisch gewählt werden. Die meisten Fortran-Dialekte dieses Typs übergeben das Problem dem Programmierer.
Viele Fortran-Dialekte, wie FORTRAN-D [Tseng 1993] und (HPF) [High Performance FORTRAN Language Specifikation 1993; Steele 1993] starten von dieser Art der Parallelität und fügen mehr direktere Datenparallelität in die Aktion, wie die Datenstrukturen den Prozessoren zugeteilt werden sollen.

5.3 Abstraktionsgrad 3

Das sind Modelle, in den Parallelität und Dekomposition explizit sind, aber Mapping, Kommunikation und Synchronisation implizit sind. Solche Modelle anfordern Entscheidung über breaking up der vorhandenen Arbeit in Stücken verteilen zu werden. Sie entlasen den Software-Entwickler von den Implikationen solcher Entscheidungen.
Das abstrakte Modell übernimmt die Abbildung der Stücke auf den Prozessoren, sowie die Kommunikation und Synchronisation, aber der Programmierer sollte explizit die Dekomposition angeben. Das Modell übernimmt die Abbildung der Programm-Stücken auf den Prozessoren, sowie die Kommunikation und Synchronisation, aber der Programmierer soll explizit die Dekomposition angeben. Dabei sollte vom Modell sichergestellt werden, dass die Zuordnung auf die Prozessoren keinen Einfluss auf die Leistung des Programms hat.

Statische Strukturen

Ein Beispiel dieser Klasse ist das BSP- Modell (Bulk Synchronous Parallelism Model).
Eine abstrakte BSP-Maschine besteht aus: Prozessoren mit lokalem Speicher, einem Netzwerk mit Punkt-zu-Punkt Verbindungen (jede Komponente kann jeder anderen Komponente Nachrichten schicken), sowie aus einem Mechanismus zur Synchronisation der Komponenten. Ein Algorithmus im BSP- Modell besteht aus zeitlichen Abschnitten, Supersteps genannt, die durch Synchronisation getrennt sind. Innerhalb eines Supersteps kann jeder Prozessor mit seinen eigenen Daten rechnen oder Nachrichten an andere Prozessoren senden. Danach ruft er eine Synchronisationsfunktion auf. Wenn alle Prozessoren ihre Synchronisation gestartet haben, wird gewartet, bis die Nachrichten ihr Ziel erreicht haben. Anschließend erfolgt ein neuer Superstep. Erst in diesem stehen die, im vorigen Schritt gesendeten Nachrichten, zur Verfügung.

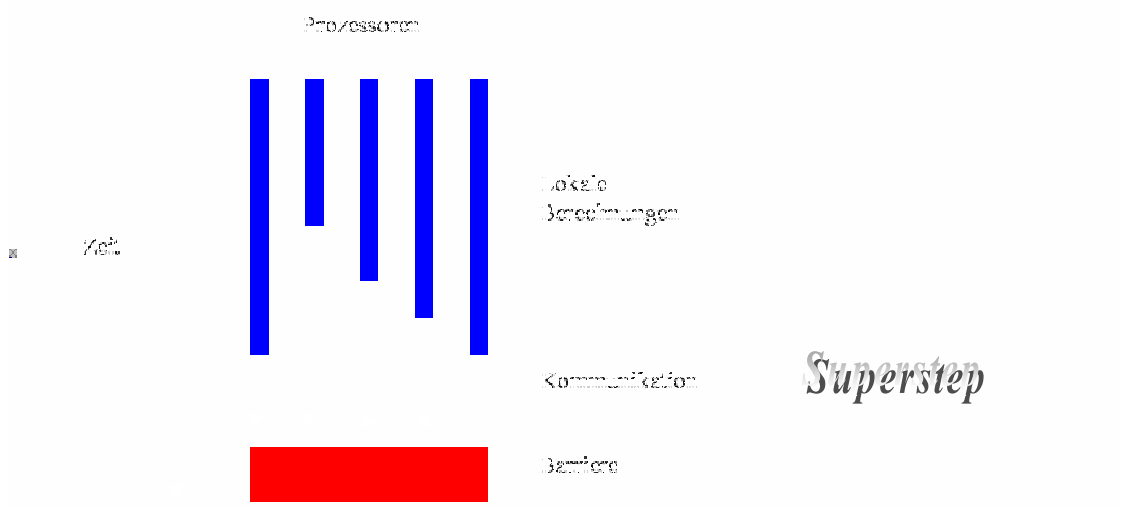


Abbildung 6

Die Länge eines Supersteps lässt sich abschätzen durch:

- Die lokale Arbeit eines Prozessors w
- Kommunikationskosten: Wenn jeder Prozessor maximal h Nachrichten sendet und h Nachrichten empfängt, so ist h*g eine obere Schranke für die Kommunikationskosten, wobei g die Bandbreite des Netzwerkes ist.
- Die Kosten für die Synchronisation L

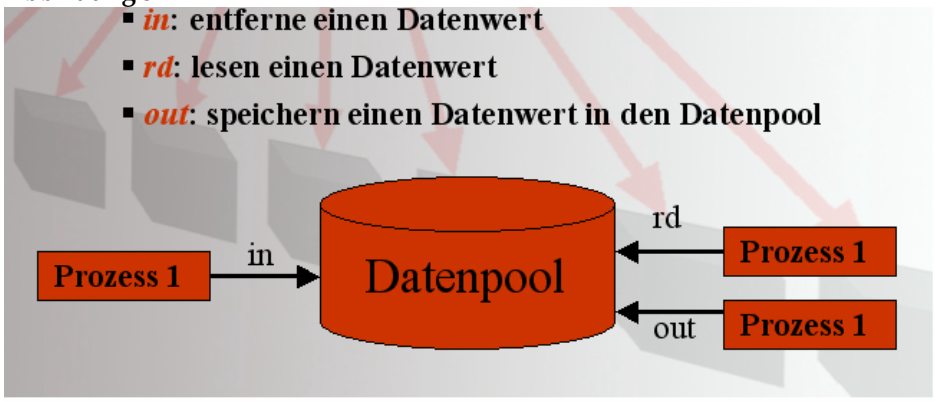
Somit ist die Länge eines Supersteps die Summe w+ h*g + L, die allerdings nicht berücksichtigt, dass das Versenden der Nachrichten zeitgleich mit der lokalen Berechnung eines Prozessors stattfinden kann, vorausgesetzt der Computer erlaubt das asynchrone Senden von Nachrichten. Es handelt sich also um eine worst-case-Abschätzung. [2]

5.4 Abstraktionsgrad 4

Modelle in den Parallelismus, Dekomposition, und Mapping explizit sind, aber Kommposition und Synchronisation implizit sind. Hier muss der Softwaren-Entwickler die Arbeit nicht nur in Stücke brechen. Er muss darauf sich kümmern, wie man am bestens die Stücke auf den gekennzeichneten Prozessor setzt. Weil Location oft einen markierten Effekt hat auf die Kommunikationsleistung hat erfordert das fast unvermeidlich eine Kenntnis des gekennzeichneten Verbindungsnetzes des Prozessors. Es ist schwer solche Software über unterschiedliche Architekturen zu bilden.

Dynamische Strukturen
Die Koordinationssprache Linda ist ein Vertreter dieser Klasse, sie wurde Anfang der 80er Jahre auf der Yale-Universität entwickelt. Sie stellt einen Datenpool (Tuple Space) zur Verfügung, in den man Daten ablegen oder entnehmen kann Dies ersetzt die Punkt-zu-Punkt-Kommunikation zwischen den Prozessoren.
Linda aus drei Operationen:

Abbildung 5



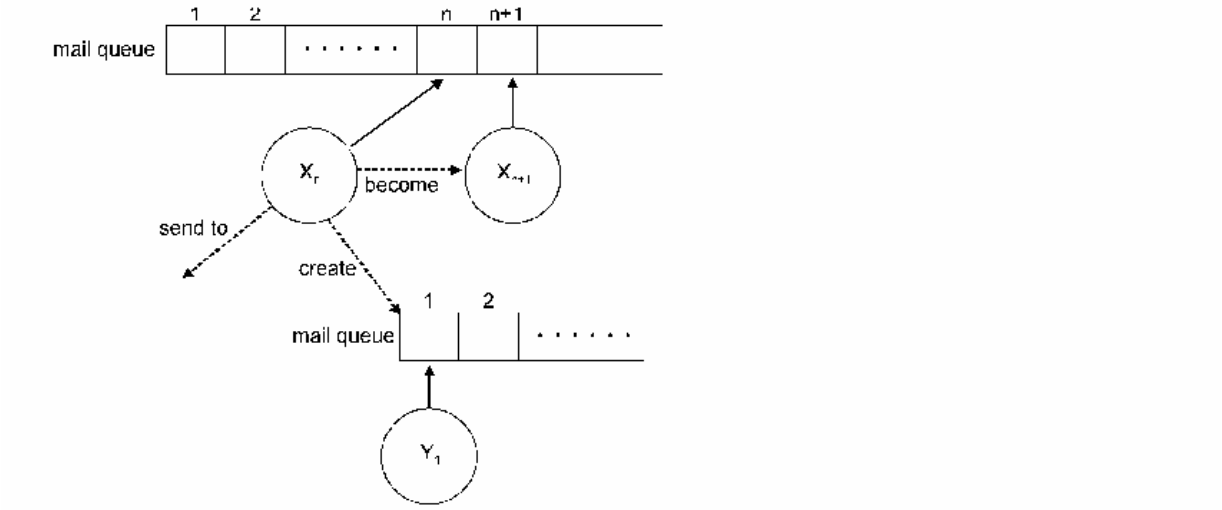
Mit diesen Operationen kann man zu einer Programmiersprache hinfügen und somit zur Koordination von Prozessen verwendet. Ein Teil der Datenfelder ist zur Identifikation der Daten reserviert und wird als Schlüssel interpretiert. Somit kann eindeutig auf die Daten zugegriffen werden. Für Rechner mit verteiltem Adressraum müssen die Zugriffsoperationen auf dem Pool durch Kommunikationsoperationen ersetzt werden.

5.5 Abstraktionsgrad 5

Dynamische Strukturen Vertreter :Actor-Modell

ist eine abstrakte Darstellung eines *Aktors*. Dieser besteht aus einer beliebig großen Mail-Queue, deren Slots mit natürlichen Zahlen numeriert werden. Der *Aktor* X hat seine ersten n-1 Nachrichten schon abgearbeitet und bearbeitet momentan die Nachricht n. Die Operationen werden von einer sogenannten **actor machine** Xn ausgeführt. Eine solche *actor machine* besitzt einen inneren Zustand in Form eines Verhaltens, das die auszuführenden Primitiven bestimmt. Im Beispiel versendet Xn eine Nachricht an ein nicht eingezeichnetes Objekt, erzeugt einen neuen *Aktor* Y und legt damit gleichzeitig

dessen initiale *actor machine* Y1 fest. Weiterhin führt die *become*-Operation zum Festlegen des *replacement behaviour* und damit zur Erzeugung der actor machine Xn+1.



Wichtig ist, dass sehr viele dieser Vorgänge parallel ablaufen können. Und zwar kann man drei verschiedene Arten der Parallelität unterschieden:

Parallelität zwischen Sender und Empfänger:
Aufgrund der asynchronen Kommunikation kann der Sender einer Nachricht sofort weitere Operationen ausführen, auch wenn der Empfänger die Nachricht noch nicht vollständig bearbeitet hat oder sogar noch mit anderen Nachrichten beschäftigt ist.

Parallelität der einzelnen Operationen innerhalb einer actor machine:
Da die Operationen, die eine *actor machine* ausführt, im allgemeinen unabhängig voneinander sind, können sie gleichzeitig ausgeführt werden. In wären dies das Verschicken einer Nachricht, das Erzeugen von Y und die Bestimmung des Folgeverhaltens.

Parallelität der einzelnen *actor machines*:
Mehrere *actor machines* des gleichen Objekts können parallel arbeiten. Dies ist möglich, da sie eigene, unabhängige innere Zustände besitzen. D.h. im Beispiel kann Xn+1 arbeiten, sobald Xn den Folgezustand festgelegt hat und die Kommunikation n+1 eingetroffen ist (diese Nachricht kann natürlich schon längere Zeit in der Mail-Queue stehen), auch wenn Xn noch beschäftigt ist.

5.6 Abstraktionsgrad 6(Alles Explizit)

Der Programmierer muss sich bei diesem Modell um alle Details der parallelen Abarbeitung kümmern, dass heißt er muss Zerlegung, Verteilung, Kommunikation und Synchronisation explizit angeben.
Die Vorteile der Modelle sind folgende:

- Man kann ein Standart-Compiler verwenden
- Der Programmierer explizit die parallele Abarbeitung steuert und somit auch ein effizientes paralleles Programm erhalten kann

Dynamische Strukturen

Vertreter dieser Klasse sind die Thread-Programmiermodelle und die Message- Passing-Programmiermodelle wie MPI (Message Passing Interface) und PVI (Parallel Virtual Machine).
PVM und MPI sind Varianten eines Message Passing Protokolls, d.h. Programme schicken sich gegenseitig Messages zu, wenn sie kommunizieren müssen. Message Passing kann zwar schnell implementiert werden, ist jedoch der Grund, weshalb paralleles Programmieren so schwierig ist. Schon zwei Prozesse können sich gegenseitig in einem Deadlock lahmlegen, da sie endlos darauf warten, dass der andere eine Ressource freigibt. Wir werden uns in diesem Artikel auf MPI und BSP beschränken. Beide sind "Single Instruction, Multiple Data"-Protokolle (SIMD): alle Prozessoren führen die gleichen Anweisungen aus, jedoch mit verschiedenen Daten.

- MPI ist Variante eines Message Passing Protokolls, d.h. Programme schicken sich gegenseitig Nachrichten zu, wenn sie kommunizieren müssen.

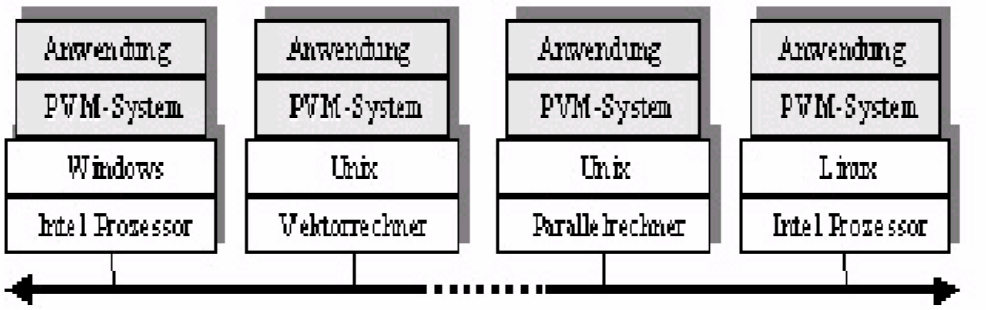
MPI stellt Routinen für häufig benötigte Funktionen zur Verfügung, darunter broadcast und reduce

MPI_InitMPI Initialisieren
MPI_Comm_size wie viele Prozessoren vorhanden sind MPI_Comm_rank Welcher Prozessor bin ich?
MPI_Send eine Message schicken
MPI_Recv eine Message empfangen
MPI_FinalizeMPI terminieren.

- PVM (Parallel Virtual Machine)

Das ist ein Softwarepaket. Es ermöglicht: heterogenes Netz von Computern als einen virtuellen Parallelrechner zu verwenden. Das PVM-Paket besteht aus zwei Bestandteilen: dem Dämon (pvmd) und der Benutzerbibliothek (libpvm).
Das PVM-Paket ist für alle Unix- und Windows-Systeme erhältlich.

PVM unterstützt alle TCP-basierten Rechnernetze wie FDDI, ATM, Ethernet, etc. und enthält C, C++ und Fortran Bibliotheken.
Die Graphik zeigt, dass es möglich ist eine Anwendung auf einem heterogenen System parallel auszuführen.



Statische Strukturen PRAM(Parallel Random Access maschine)
ist eine Registermaschine, die parallel Befehlbearbeitung ausführt.
Beispiel (1)
for <Bedingung> pardo <Anweisungen>
for i = 1 to 100 pardo xi := 0
Wert 0 -> 100 Speicherplätze gleichzeitig
initialisieren
Beispiel (2)
gegeben eine Liste, die unsortiert in
den Speicherzellen x1 bis xn gespeichert
Sind. Gesucht MAX-Element

```
for i, j = 0 to n pardo
      if xi >= xj
            then bij := 1
      else  bij := 0
   fi
for i = 0 to n pardo
      mi := bi1 & bi2 &…
      6 bin
```

6 Zusammenfassung

In der Seminararbeit wurde ein Überblick über parallele Programmiermodelle und Sprachen auf Grund folgende sechs wichtiger Kriterien repräsentiert.

- Leicht zu programmieren
- Architektur nicht abhängig
- Model muss einfach und verständlich
- Abstrakt
- Performance
- Kosen

Die Modelle wurden geschätzt abhängig davon, wie sie diese Kriterien erfüllen. Modelle, Sprachen und Werkzeuge stellen einen Vermittler zwischen Benutzern und paralleler Architektur dar und erlauben die einfache und wirkungsvolle Anwendung der parallelen Berechnung in vielen Verwendungsgebieten. Die Verwendbarkeit der Modelle und der Sprachen, die von der Architekturkompliziertheit entziehen, hat eine bedeutende Auswirkung auf den parallelen Software-Entwicklung Prozess und folglich auf den weit verbreiteten Gebrauch von parallelen Rechnersystemen.

So können wir hoffen, dass innerhalb einiger Jahre, es werden Modelle gegeben, die einfach zu programmieren sind und mindestens gemäßigte Abstraktion besitzen. Und man das diese Modelle in einer breiten Spektrum der parallelen Computer verwendet werden kann. Sie sollen einfach zu verstehen und eine vorhersagbare gute Leistung durchführen. Man kann hoffen, dass in nächsten Jahren Modelle gegeben werden, die leicht zu programmieren sind.

Es wird lange dauernd bis die Software-Entwicklungsmethoden in allgemeinen Gebrauch kommen, aber es sollte keine Überraschung sein, weil wir noch mit Software-Entwicklung für die sequenzielle Programmierung kämpfen. Die Kosten für Programme zu berechnen ist für jedes mögliches Modell mit vorhersagbarer Leistung möglich, aber, solche Kosten in Software-Entwicklung in einer nützlichen Weise zu integrieren ist viel schwieriger.

Quälen

- **David B.Skilicom & Domenico Talia:**

Models and Languages for parallel Computation“

- **Prof. Dr. Herbert Kuchen**

„Programmierung mit Algorithmischen Skeleten“

- **Prof. Dr. R. Loogen**

„Parallele Programmierung Vorlesung S/2005“

- **Marco Gilbert, Universität Trier**

„Parallel Virtual Machine

- **Andreas Justen : Proseminar**

„Paralleles Rechnen“