

**Paralleles Rechnen  
Parallelisierende Compiler  
Markus Weissmann  
WS 2004/05**

## **1. Einleitung**

Ein hoher Parallelitäts- oder doch zumindest Nebenläufigkeitsgrad ist heute in fast jeder Applikation, sei sie für einen eingebetteten Netzwerkcomputer, oder für einen SMP Computer mit 32 CPUs, erwünscht. Selbst einfachste CPUs besitzen mittlerweile SIMD Befehle die genutzt werden wollen, auf den Desktop-PCs kommen SMT und SMP Systeme immer stärker zum Einsatz und auf Servern und Mobilprozessoren verlangen VLIW Prozessoren nach Parallelität.

Die Erkennung von Nebenläufigkeit und Parallelität sind für den Menschen oft schwierig, die Implementierung von Interprozesskommunikation, Nutzung gemeinsamen Speichers und SIMD Assemblerinstruktionen aufwendig und fehleranfällig.

Es ist wünschenswert sich möglichst viel dieser Arbeit von der Maschine erleichtern zu lassen oder sie sogar komplett von ihr - ohne jegliches manuelles Zutun - erledigen zu lassen, da dies unmittelbar Zeit spart und Fehler vermeidet.

## **2. Parallelisierung**

### **2.1 Parallelisierung mit Hilfsmitteln**

#### **2.1.1 MPI Bibliotheken**

Die Message Passing Interface Frameworks PVM<sup>1</sup> und MPI<sup>2</sup> sind bekannte Bibliotheken um Nachrichten zwischen Nodes in einem Parallelrechner auszutauschen. Ihr Nachteil ist, dass sie trotz aller Abstraktion, den Programmierer nicht davor bewahren, den Austausch der Nachrichten selbst durchdenken und implementieren zu müssen. MPI führt lediglich eine weitere Abstraktionsebene ein, die die Netzwerkprogrammierung vor dem Programmierer verbirgt, nicht jedoch das Problem der Kommunikation an sich.

#### **2.1.2 Sprachunterstützung in Fortran**

Fortran hat seit der Fortran90 bzw. der Fortran95 Spezifikation eine Spracherweiterung, die es dem Programmierer erlaubt, Parallelität in einer Schleife explizit zu deklarieren.

Es existieren die "WHERE"<sup>3</sup> (Fortran90) und die allgemeinere "FORALL" (Fortran95) Schleife. "WHERE" führt eine Operation auf allen Elementen eines Arrays aus, die den gegebenen Bedingungen genügen. Die "FORALL"<sup>4</sup> Schleife verlangt als Parameter eine Menge, für deren Elemente sie ihre Befehle jeweils abarbeitet.

Beiden Schleifen ist gemein, dass die Ausführungsreihenfolge der einzelnen Schleifendurchläufe nicht deterministisch ist. Durch die vom Programmierer explizit gegebene Beliebigkeit der Ausführungsreihenfolge, kann der Fortran Compiler diese optimieren, und v. a. die parallele Ausführung der einzelnen Schleifendurchläufe - ohne weitere Analysen - ermöglichen.

#### **2.1.3 OpenMP Framework**

OpenMP<sup>5</sup> ist eine hybride Spracherweiterung und Bibliothek für die Sprachen C,

<sup>1</sup> <http://www.csm.ornl.gov/pvm/>

<sup>2</sup> <http://www-unix.mcs.anl.gov/mpi/>

<sup>3</sup> [http://www.unm.edu/cirt/introductions/aix\\_xlfortran/xlflrm54.htm](http://www.unm.edu/cirt/introductions/aix_xlfortran/xlflrm54.htm)

<sup>4</sup> [http://www.unm.edu/cirt/introductions/aix\\_xlfortran/xlflrm55.htm](http://www.unm.edu/cirt/introductions/aix_xlfortran/xlflrm55.htm)

<sup>5</sup> <http://www.openmp.org/>

C++ und Fortran. Um OpenMP zu verwenden wird deshalb nicht nur die Bibliothek benötigt, sondern ebenfalls ein spezieller C/Fortran Compiler, der die OpenMP-Syntax beherrscht.

Dank der Architektur- und Plattformunabhängigkeit von OpenMP ist es auf den verschiedensten Systemen einsetzbar, von Sparc/Solaris über PPC/Linux bis IA32/Windows.

OpenMP erlaubt auf Shared-Memory-Architekturen eine elegante Thread-Nutzung, vom Grundkonzept nicht unähnlich dem Prozessmodell unter Unix.

#### **2.1.4 Unified Parallel C (UPC)**

Ein im Ansatz mit OpenMP verwandtes Konzept wird von UPC<sup>6</sup>, dem Unified Parallel C, verfolgt, welches von einigen U.S.-amerikanischen Universitäten zusammen entwickelt wird. Für UPC wird ebenfalls ein eigener Compiler bzw. - je nach Implementierung - ein spezieller Präprozessor zusammen mit einem nativen Compiler benötigt.

UPC ist ein Dialekt der Sprache C und zielt sowohl auf Shared-Memory- als auch auf Distributed-Memory-Architekturen. Ähnlich wie die Spracherweiterungen in Fortran muss sich der Programmierer prinzipiell nur um die explizite Deklaration des potentiell parallel ausführbaren Codes kümmern. Dies geschieht in Form eines neuen Schleifenkonstrukts, der "upc\_forall"-Schleife<sup>7</sup>. Die ihr übergebenen Parameter beschreiben eine Menge, deren Elemente in beliebiger Reihenfolge abgearbeitet werden dürfen. Auf Parallelrechnern mit verteiltem Speicher muss der Programmierer des weiteren noch angeben, welche Datenbestände zwischen den Nodes geteilt werden müssen.

#### **2.2 Automatische Parallelisierung**

Die vollautomatische Parallelisierung ist die logische Weiterentwicklung der Parallelisierungsideen zur nächsten Stufe. Mit einem solchen Übersetzer kann, schlicht durch seine Benutzung, die Effizienz eines auf einem Parallelrechner ausgeführten seriellen Programmes gesteigert werden.

In diesem Szenario kann sich der Softwareingenieur voll auf die Korrektheit und Funktionalität seiner Anwendung konzentrieren und tappt nicht in die mannigfaltigen Fallstricke von Race-Conditions, Deadlocks etc., die die Nebenläufigkeit mit sich bringt. Ein Korrektheitsbeweis für seriellen Code ist ebenfalls einfacher zu bewerkstelligen. Auch wird die Produktivität erhöht, da das aufwendige Fehlersuchen in parallelem Code entfällt.

Ebenfalls lässt sich mit solch einem Compiler alter, bereits existierender Code durch die Ausnutzung eines Parallelcomputers beschleunigen, ohne Zeit in ein Neuschreiben des Programms investieren zu müssen.

##### **2.2.1 Compiler Grundlagen**

Der Programmcode durchläuft bei der Übersetzung folgende Stufen:

1. Lexikalische Analyse
2. Grammatikalische Analyse (Parsen)
3. Semantische Analyse
4. Compiler Analyse (Kontrollfluss und Datenfluss)
5. Optimierung
6. Codegenerierung

Wobei die ersten drei Stufen als Compiler-Frontend, die Stufen vier bis sechs als Compiler-Backend bezeichnet werden.

Die lexikalische Analyse identifiziert die Tokens der Sprache im Quellcode, das Parsen generiert eine Zwischencoderepräsentation des Programms. Die semantische Analyse prüft die (semantische) Korrektheit des erzeugten Zwischencodes.

Die für die Parallelisierung interessanten Phasen sind die Compiler Analyse und die

---

<sup>6</sup> <http://upc.gwu.edu/>

<sup>7</sup> <http://www.gwu.edu/~upc/faq.html>

darauf fußende Optimierung, auf die im folgenden näher eingegangen wird. Abschliessend obliegt es der Codegenerierung die dem jeweiligen System inwohnende Parallelität durch entsprechenden Maschinencode auszunutzen (z. B. SSE Befehle auf IA32, Threads auf SMP und SMT Systemen etc.).

## **2.2.2 Verfahren zur Programmanalyse**

Die beiden vom Übersetzer durchgeführten Flussanalysen sind die Kontrollflussanalyse und die Datenflussanalyse.

### **2.2.2.1 Kontrollflussanalyse**

Der Kontrollfluss wird als Graph der zusammenhängenden Codeblöcke erkannt. Dafür wird zunächst jede Zeile des vom Compiler-Frontend erzeugten Zwischencodes als Block definiert. Diese werden dann zu grösseren Blöcken zusammengefasst, sofern die Ausführungsreihenfolge der beiden Blöcke unmittelbar und unbedingt aufeinander folgt. Die Kanten des entstandenen Graphen sind i. A. an den Stellen, an denen der Zwischencode Sprungbefehle und Sprungmarken besitzt.

Der Kontrollflussgraph kann nun auf Dominatorknoten durchsucht werden. Diese Knoten haben die Eigenschaft, dass sie vor allen anderen Knoten im Graphen, die von ihnen dominiert werden, durchlaufen werden. Zum Finden der Dominatoren sind effiziente Algorithmen<sup>8</sup> bekannt. Diejenigen Dominatoren, die im Kontrollflussgraphen Rückwärtskanten besitzen, sind Schleifenköpfe. Dies ist insoweit von Interesse, als die einzelnen Durchläufe von Schleifen potentiell parallel ausführbar sind, bzw. Parallelisierung von Schleifencode zu einem merklicheren Speedup führt, da davon auszugehen ist, dass er mehrfach durchlaufen wird.

Da die Dominatorsuche auf dem Zwischencode arbeitet, findet sie auch in Quellcode der z. B. "goto"<sup>9</sup> verwendet zuverlässig die Schleifenköpfe.

### **2.2.2.2 Datenflussanalyse**

Die Datenflussanalyse ermöglicht es dem Compiler zu entscheiden, ob die Ausführungsreihenfolge von zwei Zwischencodebefehlen das Ergebnis des Programms beeinflusst.

Es gibt vier Arten von Abhängigkeiten, die Flow-, Anti-, Output- und Input-Abhängigkeit<sup>10</sup>. Sie beziehen sich auf den Inhalt einer Variable, und ob die beiden untersuchten Operationen diese lesen oder schreiben wollen.

Falls Operation A die Variable schreibt und Operation B sie danach liest, spricht man von Flow-Abhängigkeit. Die Anti-Abhängigkeit liegt im umgekehrten Fall vor. Falls beide Operationen die Variable schreiben wollen, handelt es sich um die Output-Abhängigkeit. Wollen hingegen beide lesen, spricht man von einer Input-Abhängigkeit, die jedoch im folgenden vernachlässigt werden soll, da sie nur bei feingranularer Parallelität auf Systemen mit exklusivem Lesezugriff auf den Speicher zum Tragen kommt.

Sollen nun zwei Operationen parallel ausgeführt werden, muss der Compiler sicherstellen, dass sich zwischen ihnen keine Datenflussabhängigkeit ergeben hat. Die Abhängigkeiten können u. U. durch das Einfügen temporärer Variablen verschoben werden.

Der diesbezügliche Königsweg ist das sog. Static Single Assignment (SSA)<sup>11</sup>. Bei diesem Verfahren wird durch das Einführen von weiteren Variablen und unter Benutzung der nichtdeterministischen phi-Funktion sichergestellt, dass jede Variable im gesamten Zwischencode nur ein einziges mal geschrieben wird. Auf diese Weise reduzieren sich die möglichen Datenflussabhängigkeiten lediglich auf die Flow-Abhängigkeit. Mit dem Static Single Assignment erkaufte man sich durch das später nötige Auflösen der phi-Funktionen und die Konsolidierung der zusätzlichen

<sup>8</sup> [BuKRW] A new, Simpler Linear-Time Dominators Algorithm, 1998

<sup>9</sup> <http://www.acm.org/classics/oct95/>

<sup>10</sup> [PiBJMS] Dependence Flow Graphs: An Algebraic Approach to Program Dependencies, 1991

<sup>11</sup> Modern Compiler Implementation in Java, Cambridge University Press, 1997

Variablen einen drastisch reduzierten Datenflussgraphen.

### **2.2.3 Optimierungsmöglichkeiten**

Als Voraussetzung für eine effektive Beschleunigung durch die Parallelisierung existieren mehrere Bedingungen. Zum einen muss die Operation frei von Seiteneffekten sein, sie muss z. B. für gegebene Parameter stets reproduzierbar das exakt selbe Ergebnis liefern. Um eine sinnvolle Parallelisierung zu erreichen, muss die zu optimierende Operation ausserdem teuer im Sinne von Rechenaufwand sein.

Die Reproduzierbarkeit kann vom Übersetzer nur dann gewährleistet werden, wenn er die Funktion kennt, sie also Bestandteil des zu compilierenden Programmcodes oder einer Standardbibliothek ist. Hierbei ist zu beachten, dass die meisten Compiler lediglich den Code einer einzelnen Quellcodedatei betrachten und so keine Sicherheit bzgl. der Seiteneffektfreiheit von bereits compiliertem Objektcode besitzen.

Ein spezielles Augenmerk bei der Optimierung ist auf Code in Schleifen zu legen. Dies geschieht aus zweierlei Gründen. Zum Einen werden die Operationen in einer Schleife potentiell sehr oft durchlaufen, was im Falle einer erfolgreichen Optimierung der einzelnen Funktion zu einem deutlichen Speedup führt, zum Anderen können die Funktionen untereinander parallel ausgeführt werden, was für die Schleife im Idealfall eine Beschleunigung um den Faktor der im System verfügbaren Prozessoren Recheneinheiten nach sich zieht (vgl. Amdahls Gesetz).

Dank dem Dominatorsuchalgorithmus ist es dem Compiler möglich Schleifen im Code zu identifizieren die nun als Ziel einer eingehenderen Analyse in Frage kommen. Als Voraussetzung, ob verschiedene Schleifendurchläufe nebenläufig ausführbar sind, muss der Compiler nun die Datenabhängigkeiten zwischen den einzelnen Durchläufen feststellen. Bei Verwendung des Static Single Assignment sind hierbei natürlich wieder nur die Flow-Abhängigkeiten von Belang.

Falls nun eine Schleife eine Flow-Abhängigkeit vom  $n+1$ -ten zum  $n$ -ten Durchlauf trägt, also die jeweilige Iteration die Daten der vorhergehenden liest, ist eine Parallelisierung i. A. nicht möglich (siehe z. B. Gauss-Seidel Verfahren zur Bestimmung linearer Gleichungssysteme).

Sofern es jedoch innerhalb einer Schleife keinerlei Datenabhängigkeiten gibt, ist die Ausführungsreihenfolge der einzelnen Schleifendurchläufe beliebig, es können also, bei Vorhandensein von genügend Prozessoren, prinzipiell alle Durchläufe parallel ausgeführt werden.

### **2.2.4 Tricks zur Optimierung**

Im allgemeinen Fall sind Schleifen durch Datenabhängigkeiten in ihrer Parallelisierbarkeit eingeschränkt. Um diese Abhängigkeiten zu umgehen gibt es, neben dem bereits genannten Static Single Assignment, noch weitere Transformationen.

Die Skalare Transformation behebt eine Output-Abhängigkeit, welche dadurch entsteht, dass ein und dieselbe Variable in jedem Schleifendurchlauf geschrieben wird. Unter Einführung eines temporären Vektors, dessen  $n$ -tes Element beim jeweils  $n$ -ten Schleifendurchlauf geschrieben wird, kann der finale Schreibvorgang auf die problembeladene Variable hinter die Schleife verschoben werden. Da nun nach der Schleife nur einmal der Wert einer Komponente des Vektors in die Variable geschrieben wird, innerhalb der Schleife jedoch immer nur in einen anderen Teil des Vektors, kann die Schleife nun problemlos parallel ausgeführt werden. Es ist anzumerken, dass das SSA-Verfahren die Output-Abhängigkeiten von vornherein eliminiert. Deshalb tritt dieses Problem bei einem Übersetzer der das SSA verwendet nicht auf.

Im Falle zweier verschachtelter Schleifen, kann es sinnvoll sein, die Schleifenvertauschung vorzunehmen. Sofern die Datenflussabhängigkeiten dies erlauben, ist eine Vertauschung von Vorteil, um die Granularität des Parallelitätsgrades zu verändern. Sollte sich z. B. die äussere Schleife parallel ausführen lassen, die innere jedoch nicht, kann es auf einer Vektorcomputerarchitektur

von Vorteil sein die Schleifen zu tauschen. In diesem Fall kann dann die innere Schleife mit Hilfe der Vektoreinheit parallel ausgeführt werden. Auf einer Architektur mit dazu relativ hohen Kommunikationskosten, z. B. einem Cluster, gilt das genaue Gegenteil: Eine parallelisierbare äussere Schleife ist besser, da dann alle inneren Schleifen von jeweils einem Node bearbeitet werden können, wobei während einer Schleifenberechnung dann keine weitere Kommunikation nötig ist.

### **2.2.5 Codegenerierung**

Bei der Codegenerierung ist der nun gewonnene Zwischencode auf den Maschinencode des Zielcomputers abzubilden. Auf einem System mit SIMD bzw. Vektorbefehlen sind die entsprechenden Assemblerbefehle (z. B. SSE, AltiVec) zu benutzen, um beispielsweise eine Schleife mit  $n$  Durchläufen durch eine Schleife mit  $n/4$  Iterationen zu ersetzen, in denen jeweils 4 der ursprünglichen Durchläufe durch die Vektoreinheit der CPU parallel berechnet werden. Auf Prozessoren mit VLIW Architektur (z. B. IA64, Crusoe) verhält es sich ähnlich.

Auf Systemen mit grobgranularerer Parallelrecheneinheit wie z. B. SMP oder SMT Systemen gilt es die erkannte Parallelität auf die einzelnen CPUs zu verteilen. Auf eingebetteten Systemen wie digitalen Signalprozessoren (DSP) muss der Compiler das Scheduling selbst übernehmen und zur Übersetzungszeit festlegen. Auf komplexeren Systemen mit vollwertigem Betriebssystem und eigenem Scheduler sind die Thread-Bibliotheken (z. B. pthread<sup>12</sup>) des Betriebssystems die entscheidenden Komponenten, die der Compiler kennen und benutzen muss. So wissen auf SMT Computern die Scheduler des Betriebssystems um die asymmetrische Leistungsfähigkeit der virtuellen CPUs, was der Compiler berücksichtigen muss.

Cluster oder gar Grids sind als Zielplattform für einen parallelisierenden Compiler eine ernstzunehmende Herausforderung, da das Scheduling hier ungemein komplexer als auf SMP Systemen ist. In diesen Bereichen verlässt man sich derzeit noch nicht auf die Fähigkeiten des Compilers die Parallelität optimal auszunutzen.

In den Bereich der Codegenerierung fällt zu guter letzt noch die sogenannte Peephole- oder Schlüssellochoptimierung. Hierbei wird der Zwischencode - oder u. U. auch der Maschinencode - noch nach Optimierungsmöglichkeiten durchsucht. Die hierbei erreichbaren Leistungssteigerungen sind gering, die noch findbare Parallelität (z. B. mehrer aufeinander folgende Additionen durch SIMD Befehle ersetzen) ist äusserst spärlich und führt zu keinem nennenswerten Speedup mehr.

### **2.2.6 Existierende Parallelisierende Übersetzer**

Nicht nur kommerzielle Compiler wie der Intel C/C++/Fortran-Compiler oder IBMs Compiler für die Power Architektur besitzen Vektorisierungseinheiten. Die freie und weit verbreitete GNU Compiler Collection, kurz gcc<sup>13</sup>, bietet ebenfalls Parallelisierungsalgorithmen.

Die derzeit existierenden Übersetzer leiden unter dem Manko, dass sie meist nur eine einzige Datei des Quellcodes auf einmal untersuchen. Dies führt dazu, dass die Granularität der Parallelität, die diese Compiler finden können, meist eher gering ist. Sie sind deshalb für die auf heutigen Personal Computern weitverbreiteten SIMD Recheneinheiten gut geeignet, z. T. auch für kleine SMP Systeme oder die derzeit an Bedeutung gewinnenden SMT Prozessoren. Für Cluster oder gar Grids sind sie jedoch eher ungeeignet.

## **2.3 Alternative Ansätze**

Als Alternative zur sprachgestützten bzw. zur vollautomatischen Parallelisierung bietet sich ein weiterer Ansatz an, der die traditionellen Ansätze der Programmierung über Bord wirft.

Der klassische Ansatz um ein algorithmisches Problem auf einen (Parallel-) Rechner zu bringen ist, ihn in einer Programmiersprache zu schreiben. Hernach wird versucht,

<sup>12</sup> <http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html>

<sup>13</sup> <http://gcc.gnu.org/>

aus dem seriellen Quellcode ein parallel ausführbares Programm zu erzeugen. Der Zwischenschritt der Serialisierung des Problems als Programmcode zerstört evtl. die dem Problem ursprünglich innewohnende Parallelität. Eine einfache Summe über eine Menge von Integerwerten, ist im Grunde genommen mit logarithmischem Aufwand zu bewältigen (bei genügend Prozessoren). Wird dies jedoch z. B. in der Programmiersprache C als for-Schleife realisiert, so ist die entstandene Semantik eine andere, da Addition von Integerzahlen nicht assoziativ ist, was eine spätere Parallelisierung verbietet.

Um nun dieser Problematik zu begegnen, wird versucht den Zwischenschritt der Benutzung einer (klassischen) Programmiersprache komplett zu umgehen, indem man das ursprüngliche Problem als mathematische Gleichung definiert. Diese Gleichung wird dann von einem speziellen Compiler in ein Programm konvertiert, der mathematische Ausdruck wird praktisch selbst zur Programmiersprache.

Dieses Feld der parallelisierenden Compiler ist das jüngste der hier vorgestellten und ist derzeit noch ein Forschungsfeld. Nichtsdestotrotz findet diese Technik derzeit schon beim Entwurf von DSPs Verwendung.

### **3. Zusammenfassung**

Parallelisierung ist eine der wenigen Möglichkeiten, die Leistung heutiger Computersysteme noch signifikant zu steigern. Dem gegenüber steht der z. T. hohe Aufwand der parallelen Programmierung, bzw. des Findens von parallel ausführbaren Algorithmen.

Die Möglichkeiten zur parallelen Programmierung, die von Programmiersprachen wie Fortran und Frameworks wie OpenMP unterstützt werden, sind für den Softwareentwickler ein essentielles Hilfsmittel beim effizienten Implementieren paralleler Algorithmen.

Vollautomatische parallelisierende Übersetzer versuchen den Programmierer noch stärker zu unterstützen, indem sie völlig ohne dessen Mithilfe versuchen die Parallelität in gegebenem seriellen Programmcode zu finden. Dies befreit den Entwickler zwar nicht davon die geeigneten Algorithmen auszuwählen, verhilft ihm aber, sofern er um die Arbeitsweise des Compilers weiß, zu einem zuverlässigen Werkzeug, um die Parallelität seines Computers auszunutzen.

Der alternative Ansatz, die klassische Programmierung komplett zu umgehen, und ein Programm direkt aus einem mathematischen Ausdruck zu gewinnen, hat zwar noch keinen kommerziell verfügbaren Übersetzer hervorgebracht, wird aber in Zukunft v. A. für den Bereich des wissenschaftlichen Rechnens von grossem Interesse sein.