AUTOMATISCHE VEKTORISIERUNG UND PARALLELISIERUNG VON FORTRAN PROGRAMMEN

Adrian Spalka

h.o.-Computer Software GmbH, Köln

1. Deutscher Fortran 90 Kongreß

Köln, 16./17. November 1992

INHALTSVERZEICHNIS

1. EINLEITUNG

Das Verlangen nach mehr Rechenkraft wächst in vielen wissenschaftlichen Bereichen ununterbrochen. Die experimentelle Forschung wird nicht selten von hohen Kosten und vielen, auch ökologischen Risiken begleitet. Daher werden Experimente immer häufiger durch rechnergestützte Simulationen ersetzt oder ergänzt. Die mathematischen Modelle dieser Simulationen sind sehr rechen- und speicherintensiv. Konkrete Anwendungen aus der Meteorologie, Atom- und Plasmaphysik erfordern die Berechnung von 10^6 bis 10^9 Unbekannten, die, als klassische Algorithmen formuliert, die Leistungsgrenze vorhandener Computer um Größenordnungen gesprengt haben.

Alle diese verschiedenen Probleme weisen dabei die gleiche mathematische Struktur auf: partielle Differentialgleichungen (PDGL). Als ein Maßstab für die Leistung eines Superrechners kann die Diskretisierung elliptischer PDGL gelten, die ihrerseits zur algebraischen Gleichungen auf Gitterstrukturen führt, den *Gitterproblemen.*

Unter den Algorithmen, die zur Lösung von Gitterproblemen herangezogen werden, nehmen die in den achtziger Jahren entwickelten und gefestigten Mehrgitterverfahren (MGV) eine zentrale Stellung ein. MGV sind inhärent parallel und haben bei N Gitterpunkten die Komplexität O(N) mit einem konstanten Faktor, der durch eine passende Wahl der Komponenten des Mehrgitters klein gehalten werden kann. Den Vergleich verschiedener Methoden und das Verhältniss von Rechenzeit und Komplexität verdeutlicht die nachstehende Tabelle, die Werte für ein Gitterproblem, die inkompressible Navier-Stokes Gleichungen (2D), von einem sequentiellen Rechner (IBM 3090) zeigt.

Gitterpunkte	Linien Relaxation	Newton-Gauß	Mehrgitterverfahren
1024	20 s	69 s	4,7 s
4096	280 s	1080 s	16 s
16384	60 min	300 min	61 s
Komplexität	$O(N^2)$	$O(N^2)$	$O(N)$

Tabelle 1.

Die Vorteile von MGV bei elliptischen PDGL sind so überzeugend, daß ihre Idee auch auf andere Bereiche übertragen wurde; hierzu gehören u.a.:

1. Monte Carlo Methoden in der Physik: Renormalisierung
2. Große, schwach besetzte Systeme: algebraische MG
3. Finite Elemente: MG auf hierarchischen Basen
4. Mustererkennung: lokale Verfeinerung mit Filtern
5. Bildverarbeitung: Methoden der Auflösung

Die primäre Zielgruppe bildeten alle Probleme, die sich auf MGV zurückführen lassen. Als nächstes sollten auch alle Standardalgorithmen aus der numerischen Mathematik effizient unterstützt werden.

2. ARCHITEKTUR VON SUPERRECHNERN

2.1 KLASSIFIKATION VON ARCHITEKTUREN

Bei der Bewertung von Rechensystemen im Hinblick auf rechenintensive Probleme hat sich die Klassifikation nach Flynn bewährt, die auf der Struktur der verarbeitenden Komponenten, d.h. der Kontroll- und der arithmetischen Einheiten (ALU, *Arithmetic and Logical Unit*), basiert. Es ergeben sich hiernach drei Klassen:

1. SISD: Single Instruction Single Data

   Hier gibt es nur einen Befehls- und einen Datenstrom, der von einer Kontroll- und einer arithmetischem Einheit verarbeitet wird. In diese Klasse fallen alle herkömmlichen Einprozessorsysteme.

2. SIMD: Single Instruction Multiple Data

   Diese Systeme sind mit einer Kontroll- und vielen arithmetischen Einheiten ausgestattet, wobei jede ALU individuellen Speicherzugriff hat. Alle arithmetischen Einheiten führen im Gleichschritt den gleichen Befehl auf ihren eigenen Daten aus. SIMD Rechner sind auch unter der Bezeichnung Feldrechner (*array computers*) bekannt. Zu Vertretern dieser Klasse gehören u.a.:

   i. ILLIAC IV mit 64 ALUs

   ii. Burroughs B6500 und BSP mit 16 ALUs

   iii. Connection Machine CM1 mit 64K 1-Bit ALUs

Vektorcomputer, die über Instruktionen verfügen, die mit Feldern, anstatt mit Skalaren arbeiten zählen zu den *schwachen* Vertretern der SIMD Rechner.

3. MIMD: Multiple Instruction Multiple Data

Dies sind asynchrone parallele Maschinen, auch Mehrprozeßsysteme genannt, die über zwei oder mehr Prozessoren verfügen und von denen jeder unabhängig von allen anderen eine Befehlsfolge ausführen kann. In Anlehnung an die Kommunikationsart zwischen den Prozessoren findet eine Aufspaltung in zwei weitere Klassen statt:

i. Eng gekoppelte Systeme mit gemeinsamem Speicher

ii. Lose gekoppelte Systeme mit privatem lokalem Speicher

Im ersten Fall findet die Kommunikation im Speicher statt, d.h. daß alle Prozessoren auf den gleichen Code und Daten zugreifen können. Diese Organisation ermöglicht einen vielseitigen Einsatz, jedoch ist die maximale Anzahl der Prozessoren in der Praxis gegenwärtig auf etwa 32 beschränkt. Dies rührt von der hohen Komplexität des Systeme her, die erforderlich ist, um eine kollisionsfreie Arbeit der Prozessoren zu gewährleisten.

Im Fall (ii) findet Kommunikation ausschließlich über den Versand von Nachrichten statt. Systeme dieser Art sind einfach beschaffen und setzen bzgl. der Anzahl der Prozessoren zur Zeit keine Obergrenze fest. Der Zugriff eines Prozessors auf fremden Speicher ist jedoch um mehrere Größenordnungen langsamer, als auf seinen lokalen. Die Effizienz solcher Systeme hängt daher direkt vom Verhältniss Berechnung/Kommunikation ab, wodurch Algorithmen mit stark lokalem Verhalten eindeutig bevorzugt werden.

2.2 BEISPIEL: DIE SUPRENUM ARCHITEKTUR

Im Rahmen obiger Klassifikation gehört Suprenum den lose gekoppelten MIMD Systemen an.

Entscheidend für diese Wahl war die Beschaffenheit moderner numerischer Algorithmen, mit den MGV im Mittelpunkt und etlichen weiteren Vorgaben:

1. Grobkörnige Parallelität entlang des MIMD Prinzips, das eine gleichzeitige Bearbeitung verschiedener Teile einer Aufgabe erlaubt.

2. Die Verbindungsstruktur innerhalb des Rechners soll der natürlichen Struktur der Aufgaben angepaßt sein.

3. Das System soll einfach programmierbar und bereits vorhandene Software einfach übertragbar sein.

4. Der Rechner soll je nach Leistungsanforderung konfigurierbar sein.

5. Vektorverarbeitung, insbesondere Erweiterungen von Fortran 90, sollen effektiv unterstützt werden.

Entlang dieser Richtlinien ist der Aufbau des Suprenum Rechners hierarchisch in mehrere Ebenen unterteilt. Die Komponenten einer jeden Ebene zeichnen sich durch ein eigenes Verbindungssystem und eine eigene Funktionalität aus. Nach aufsteigender Größe unterscheidet man:

1. Knotenrechner

   Dies ist ein hochentwickelter Vektorrechner, kurz Knoten, der auf einer Platine (223 mm x 480 mm) untergebracht ist. Hierauf befinden sich drei Modulgruppen:

   i. Prozessorkern bestehend aus:
   - MC 68020 20 MHz CPU
   - MC 68851 PMMU mit 80 MB/s DMA
   - MC 68882 Skalarkoprozessor
   - 8 MB DRAM

   ii. Pipelinevektorprozessorsatz WTL 2264/2265 von Weitek

   iii. Kommunikationskoprozessoren, die in Eigenregie, d.h. ohne die Arbeit anderer Chips zu unterbrechen, den Nachrichtenaustausch mit anderen Knotenrechnern organisieren

   Ein Knotenrechner ist die kleinste produktive Arbeitseinheit.

2. Cluster

   Cluster sind Verbunde von bis zu 16 Knotenrechnern zusammen mit einem Diagnose-, einem Platten- und ein oder zwei Kommunikationsknoten. Alle Komponenten eines Clusters sind über einen doppelten, parallelen Hochgeschwindigkeitsbus (320 MB/s) verbunden. Der Plattenknoten verfügt über eine Datentransferrate von 10 MB/s und unterstützt zwei Platteneinheiten mit je etwa 1,2 GB Kapazität; er dient vorrangig der Speicherung temporärer Daten.

3. Hochleistungskern (HL-Kern)

   Er wird aus maximal 16×16=256 Clustern gebildet, die über doppelt ausgeführte bitserielle Ringbusse in der Topologie eines Torus verbunden sind. Diese Anordnung ermöglicht eine fehlertolerante Kommunikation mit einer 50 MB/s Bandbreite, die beim Durchreichen einer Nachricht ohne Unterbrechung der Rechenprozesse erledigt wird.

Für den Benutzer ist der HL-Kern über den Front-End (FE) Rechner erreichbar, der ein gewöhnlicher sequentieller Rechner ist.

Eine Ausbaustufe von 4×4=16 Clustern kann folgende Leistungsdaten aufweisen:

Rechenleistung:	5 GFlops
Hauptspeicher:	2 GB
Kommunikation	
Intracluster:	320 MB/s
Intercluster:	50 MB/s
Plattenspeicher:	35 GB

3. SOFTWAREKONZEPT

Ein Knotenrechner kann nur auf seinen lokalen Speicher und den *Briefkasten* seines Kommunikationskoprozessors zugreifen. Nachrichtenaustausch ist der einzige Weg, um Zugang zu Daten eines anderen Knotenrechners zu erlangen. Der Front-End Rechner wird hier wie ein Knoten behandelt.

Auf dem FE Rechner findet der Benutzer Unix System V als Betriebssystem. Der HL-Kern erscheint als eine Komponente in der Unix Welt. In Wirklichkeit besitzt der HL-Kern das eigens für ihn entwickelte verteilte Betriebssystem PEACE (Process Execution and Communication Environment), das für eine effiziente Auslastung seiner Betriebsmittel zuständig ist. Die Wechselwirkungen zwischen Unix und PEACE werden für den Benutzer transparent gehalten, d.h. er entwickelt, testet und sammelt Ergebnisse unter Unix.

PEACE realisiert auf dem HL-Kern einen Merhprozeßbetrieb, den dessen Hardwarestruktur nahegelegt hatte:

1. Dynamische Prozeßstruktur, d.h. Prozesse können an jeder Deklarationsstelle im Quelltext eines Programmes definiert und überall explizit erzeugt werden. Die Anzahl vorhandener Prozesse kann nur zur Laufzeit ermittelt werden.

2. Jeder Prozeß ist zu einem Zeitpunkt nur einem Knoten zugeordnet, d.h. er besitzt einen eigenen, von allen anderen Prozessen getrennten Adreßraum.

3. Der Ablauf der Prozesse ist völlig asynchron.

4. Synchronisation und Kommunikation werden durch expliziten Nachrichtenversand erreicht.

5. In der Kommunikation werden Prozesse direkt benannt[1] und die Namensgebung ist wahlweise symmetrisch oder asymmetrisch, d.h. der Empfänger hat sowohl die Möglichkeit einen bestimmten Sender zu nennen, wie auch alle ankommenden Nachrichten zu empfangen.

6. Das Synchronisationsschema zwischen Sender und Empfänger ist fehlertolerant asynchron, d.h. der Sendeprozeß wird so lange blockiert, bis PEACE die Nachricht im Briefkasten des Empfängerprozesses abzulegen versucht, danach erhält er einen Statuswert über den Ausgang dieses Versuchs und wird entblockiert. Damit ist nur sichergestellt, Störungen ausgenommen, daß die Nachricht beim Empfänger angekommen ist, nicht aber, daß sie vom ihm gelesen wird.

Dem Empfänger bietet PEACE Dienste an, mit denen er den Inhalt seines Briefkastens abfragen kann und eine `receive` Funktion, um eine Nachricht zu lesen. Sollte der Briefkasten leer oder die angegebene Nachricht nicht vorhanden sein, so wird der Empfänger bis zu ihrem Eintreffen blockiert.

Der Start eines Programmes beginnt stets mit einem Initialprozeß (IP) auf dem Unix FE Rechner, von wo aus weitere Prozesse auf dem HL-Kern inkarniert werden. Die Prozesse an sich unterscheiden sich von nichts voneinander; es liegt in der Verantwortung des Benutzers dafür zu sorgen, daß die eigentlichen Berechnungen nicht vom IP ausgeführt werden und daß alle direkten Zugriffe der HL-Kernprozesse auf externe Datenbestände durch `send/receive` Anweisungen zum FE Rechner ersetzt wurden. Der IP terminiert erst dann, wenn alle seine Prozesse auf dem HL-Kern terminiert sind.

3.1 EIN-/AUSGABEMODELLE

Falls in einem lose gekoppelten System nicht jeder Knoten E/A betreiben kann, müssen auch hierfür spezielle Vorkehrungen getroffen werden. Für die Organisation der E/A existieren zwei Modelle der Zusammenarbeit zwischen Knoten, die E/A betreiben können (E) und solchen, die es nicht können (R):

1. Kontrollflußgesteuert:

   Die E-Prozesse werden individuell auf die R-Knoten zugeschnitten. Auf

[1] Im Gegenteil zur *indirekten* Bennenung, die z.B. einen Kanal oder eine 'pipe' adressiert.

E werden E/A-Operationen von Kommunikationsbefehlen von etwa gleicher Gestalt unmittelbar begleitet. Die Anzahl und Reihenfolge der `send/receive` Aufrufe zur Laufzeit in E und in R entsprechen sich im 1:1 Verhältnis. Neben den eigentlichen E/A-Daten müssen hier auch Daten des Kontrollflusses übergeben werden. Die Anwendung koordiniert explizit die Aktivitäten der E und der R. Bild 4. zeigt die erforderlichen Modifikationen anhand eines kleines Programmstückes.

```
...
REAL X, Y
...
READ(..., X)
READ(..., Y)
IF (X*X .GT. Y) THEN
   READ(...)
ELSE
   PRINT *, 'error: ...'
END IF
...
```

Bild 1a

E-Prozeß:

```
...
REAL X, Y
LOGICAL B01
...
READ(..., X)
SEND(..., X)
READ(..., Y)
SEND(..., Y)

RECEIVE(..., B01)
IF (B01) THEN
   READ(...)
   SEND(...)
ELSE
   RECEIVE(...)
   PRINT *, 'error: ...'
END IF
...
```

R-Prozeß:

```
...
REAL X, Y
LOGICAL B01
...

RECEIVE(..., X)

RECEIVE(..., Y)
B01 = X*X > Y
SEND (B01)
IF (B01) THEN

   RECEIVE(...)
ELSE
   SEND(...)
END IF
...
```

Bild 1b

2. Dienstleistungsorientiert:

   E-Prozesse haben einen universellen Charakter und ist stets gleich. Einen Teil davon bildet ein E/A-Kern, der einen festen Satz von Dienstleistungen zur Verfügung stellt. Die Dienstleistungen sind, je nach Art der E/A-Anforderung, über einen Prozeß- oder eine Unterprogrammschnittstelle ansprechbar. Eine Anwendung hat nur auf die E-Prozesse direkten Einfluß; mit den steht sie in einem Auftraggeber/Auftragnehmer (AG/AN) Verhältnis, d.h sie kann alle auf R-Knoten nicht verfügbare Dienste (hier speziell E/A) vom E-Knoten anfordern.

Dem Konzept nach sind Superrechner für einen Mehrbenutzerbetrieb nicht geeignet. Sie sind gezielt für die Bearbeitung eines großen, parallelen Problems ausgelegt.

3.2 VEKTORBEFEHLE IN FORTRAN

Nachdem ein Problem grobkörnig parallelisiert, d.h. in Tasks aufgeteilt wurde, gilt es, Operationen, die auf Vektoren als Ganzes ausgeführt werden können, auf den dafür speziell vorgesehenen Vektorprozessoren ablaufen zu lassen. Standardmäßig bietet Fortran 77 hier nur wenig Unterstützung an. Der Numeriker fühlt sich an dieser Stelle im Stich gelassen. Die wichtigsten Funktionen aus seinem Bereich, wie:

1. komponentenweise Verknüpfung zweier Vektoren
2. Verknüpfung eines Vektors mit einem Skalar
3. Bildung des Skalarprodukts
4. Berechnung der Norm eines Vektors (Matrix)

wurden nicht bedacht.

Es ist also klar, daß diese Operationen in Fortran 77 nur in einer externen Bibliothek implementiert werden können. Das herkömmliche Bild einer Prozedur oder Funktion kann jedoch vermieden werden, falls man auf Fortran 90 zugreifen kann. Eine fast kanonische Erweiterung der arithmetischen Operationen auf Vektoren ist entweder bereits vorgesehen oder durch Überladen möglich. Der Programmierer kann sich hier bei der Kodierung einer

mathematisch vertrauten Notation bedienen, z.B. mit den Vektoren U, V und W, und einer reellen Zahl A:

```
U = V + W
U = V * A
U = V * W
A = U * V
```

Die Vorteile dieser Lösung sind deutlich, da zum einen der Programmierer nicht gezwungen wird, sich mühselige Namen zu merken, wie z.B.

```
DVCAL(a, u, v) (Name aus IMSL) für
U = A * V;
```

zum anderen braucht die Syntax und Semantik, d.h. der Sprachumfang, nicht ausgedehnt werden, um der gewohnten Notation nahe zu kommen. Die Erstellung der obigen Bibliothek muß natürlich hardwarenah geschehen.

Abschließend sollte aber auch erwähnt werden, daß es auf diesem Weg nicht möglich ist, Operationen bereitzustellen, die etwa nur auf jedem zweiten oder dritten Element eines Vektors, oder allgemein auf nichtzusammenhängenden Teilbereichen arbeiten.

4. AUTOMATISCHE KONVERTIERUNG

Bei dem Entwurf neuer Algorithmen und deren Kodierung können MIMD, d.h Parallelisierung, und Vektorisierung direkt bedacht werden. Es gibt aber schon heute unzählige Programme, die bei ihrer Entstehung diese Punkte völlig unberücksichtigt ließen. Eine rein manuelle Konvertierung dieser Programme kommt wegen des zu erwartenden Zeitaufwands und der Fehleranfälligkeit nur sehr selten in Frage. Angesprochen werden sollen die Möglichkeiten einer

1. vollautomatischen und
2. wissensbasierten

Transformation von Programmen.

Bei einem vollautomatischen Vorgehen wird erwartet, daß der Compiler das Eingabeprogramm genau untersucht, für eine Transformation in Frage kommenden Passagen erkennt und anschließend parallelisiert und vektorisiert. Solche sogenannte Supercompiler existieren bereits. Sie sind jedoch nicht in der Lage, jedes Eingabeprogramm effizient anzugehen. Die Qualität der Ergebnisse hängt von der Struktur des Algorithmus ab; genauer gesagt, können Supercompiler versteckte Parallelität in sequentiellen Algorithmen entdecken, jedoch können sie im gegenwärtigen Stadium daraus keinen neuen Algorithmus entwickeln, der der Natur der Hardware besser entspricht.

Im allgemeinen verhindert die Komplexität der Beziehungen zwischen der Struktur des Algorithmus, der Hardwarearchitektur und den Transformationsstrategien global optimale Lösungen. Viele Faktoren lassen durchblicken, daß eine Neuordnung am besten in einem interaktiven, wissensbasierten System angegangen werden kann. Hier kann der Benutzer selbst strategische Entscheidungen treffen und das System mit Informationen versorgen, die auf automatischen Weg nicht ermittelt werden können (Unentscheidbarkeit oder *intractability*).

4.1 VEKTORISIERUNG

Die besten Kandidaten für die Umschreibung eines sequentiellen Abschnitts in eine Folge von Vektoranweisungen bieten Schleifen, die auf Feldern arbeiten.

Ein einfaches Beispiel ist:

```
DO I = 1, M
    DO J = 1, N
        A(I,J) = B(I,J) * C(I,J)
    END DO
DO
```

das vektorisiert als

```
A = B * C
```

notiert werden kann. Alle elementorientierten Operationen können geradlinig von einer auf mehrere Dimensionen erweitert werden. Daß eine Transformation nicht immer möglich ist, zeigt folgende Schleife:

```
DO I = 1, N
    A(I+1) = A(I) * B(I)
END DO
```

Eine schrittweise Nachvollziehung der Iterationen liefert folgende Reihenfolge:

```
A(2) = A(1) * B(1)
A(3) = A(2) * B(2)
...
A(N+1) =A(N) * B(N)
```

in der jede Zeile unmittelbar von der vorhergehenden abhängt. Diese, auf die Schleife bezogen, zyklische Abhängigkeit verhindert eine Vektorisierung. Allgemein gesprochen ist eine Vektorisierung nur dann sinnvoll, wenn unter Beachtung aller Datenabhängigkeiten einer Schleife die gleiche Operation auf mehreren Elementen eines Vektors ausgeführt werden kann.

Die Abhängigkeiten können sehr vielfältig sein und müssen einer Vektorisierung nicht immer im Weg stehen. In den letzte Jahren ist eine Reihe von Methoden entwickelt worden, die unter gewissen Umständen eine Abhängigkeit auflösen können. Nur kurz dem Namen nach seien einige von ihnen erwähnt:

1. Umstellung der Reihenfolge der Anweisungen:
   kann auf zwei Anweisungen angewandt werden ,wenn zwischen ihnen keine schleifenunabhängige Abhängigkeit besteht.

2. Schleifenaufspaltung:
   spaltet eine Schleife mit mehreren Anweisungen in mehrere Schleifen auf.

3. Schleifenaustausch:
   vertauscht die Köpfe zweier benachbarter, ineinander verschachtelter Schleifen.

4. Skalarausdehnung:
   überführt in einer Schleife eine skalare Größe in einen Vektor.

Ziel der Umformungen ist eine Aufspaltung der Schleifen in solche, die nur eine zyklisch unabhängige Anweisung enthalten. Als Beispiel für die Anwendung von (3) sei folgende Situation gezeigt:

```
DO I = 1, 100
    DO J = 1 TO 100
        A(I,J+1) = A(I,J) * B(I,J)
    END DO
END DO
```

nach (3):

```
DO J = 1, 100
    DO I = 1, 100
        A(I,J+1) = A(I,J) * B(I,J)
    END DO
END DO
```

nach Vektorisierung:

```
DO J = 1, 100
    A(1:100,J+1) = A(1:100,J) * B(1:100,J)
END DO
```

Auf eine bestimmte Architektur festgelegt gelte die Fundamente der Vektorisierung als gut verstanden. Eine automatische Transformation liefert in den meisten Fällen auch effiziente Ergebnisse.

4.2 PARALLELISIERUNG

Bei der Parallelisierung gibt es zwei völlig unterschiedliche Ansätze, die entweder von eng gekoppelten Mehrprozeßsystemen mit gemeinsamen Speicher oder von verteilten Systemen ausgehen.

Im ersten Fall ähnelt die Vorgehensweise der Vektorisierung bei der Suche nach Schleifenparallelität. Die meisten Anwendungen aus diesem Bereich lassen sich als eine Anzahl von Programmteilen charakterisieren, die mit gemeinsamen Daten unterschiedliche Aufgaben erfüllen, wie z.B.: Datenbanken und Prozeßüberwachung.

Lose gekoppelte Systeme eignen sich dagegen nur für Anwendungen mit einer datenlokalen Struktur. Der Ansatz zur Parallelisierung wird von der Tatsache beherrscht, daß Kommunikation teuer ist und folglich minimiert werden muß. Die Datenparallelität einer Anwendung führt zu einer Aufteilung der Daten in disjunkte Teilmengen, die in Adreßräumen verschiedener Prozesse angesiedelt werden. Jeder Prozeß führt dann das gleiche Programm auf verschiedenen Teilen des Datenbereiches aus.

Von einem sequentiellen Programm ausgehend, kann zur Zeit nur eine interaktive Parallelisierung erfolgversprechend durchgeführt werden. Dem Benutzer kommt hier die Aufgabe zu, vom System gefundene Partitionen zu korrigieren, selbst Partitionen vorzuschlagen um dann ihre endgültige Form festzulegen. Abhängig von der getroffenen Wahl und dem Programm fügt das System die nötigen Kommunikationsbefehle ein und versucht anschließend ihre Anzahl zu minimieren und den Umfang der Nachrichten zu vergrößern.

Obwohl diese Methode auf beliebige Programme anwendbar ist, für die eine Datenaufteilung festgelegt wurde, kann die Effizienz des parallelisierten Programmes nur dann zufriedenstellend sein, wenn es, an Hand numerischer Algorithmen erläutert, einige Kriterien erfüllt:

1. Es arbeitet auf einem Gitter
2. Die Berechnungen an den Gitterpunkten sind lokal, d.h. benötigen nur wenig Daten aus der angrenzenden Nachbarschaft.

Die typische Größe eines Gitters kann zur Zeit im Bereich von 10^9 Punkten liegen.

In diesem Bereich ist die Forschung und Entwicklung noch im vollen Gange.

LITERATURHINWEIS

1. Thomas, B.
   Suprenum - Meilenstein im Supercomputing.
   Hard and Soft, 4 (1989).

2. Trottenberg, U.
   On the SUPRENUM Conception (version 2).
   Suprenum Report 1.1 (1987).

3. Zima, H. P.
   Supercompilers for Supercomputers.
   ACM Press (1990).