Parallele leichtgewichtige Prozesse zur Implementierung adaptiver numerischer Verfahren

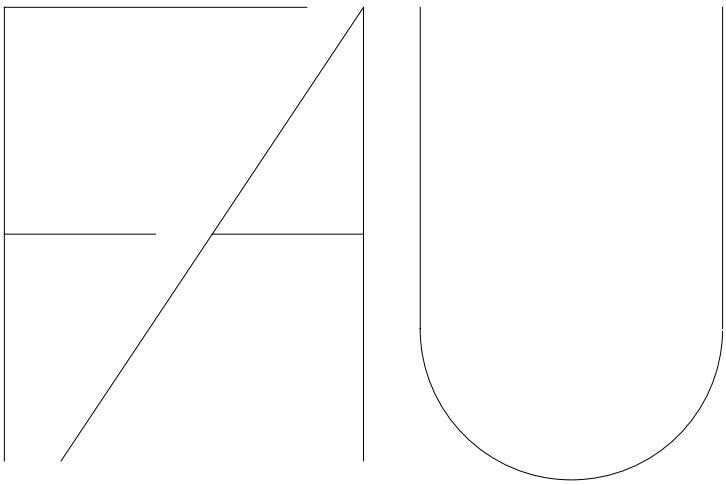
Frank Bellosa

Januar 1994 TR-I4-2-94

Interner Bericht

Institut für
Mathematische Maschinen
und Datenverarbeitung
der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl für Informatik IV
(Betriebssysteme)



Parallele leichtgewichtige Prozesse zur Implementierung adaptiver numerischer Verfahren

Frank Bellosa

email: bellosa@informatik.uni-erlangen.de

Universität Erlangen-Nürnberg
Institut für mathematische Maschinen und Datenverarbeitung IV
Lehrstuhl für Betriebssysteme

Überblick

'Zur Berechnung physikalischer Größen auf kontinuierlichen Problemgebieten wird zumeist das Gebiet diskretisiert und eine Näherungslösung für jeden diskreten Punkt errechnet. Bei adaptiven Berechnungsverfahren kann die Feinheit der Diskretisierung dynamisch verändert und die Bearbeitungsreihenfolge der Punkte vom Ausgang einzelner Rechenschritte abhängig gemacht werden. Da einfache reguläre Datentrukturen wie Vektoren und Matrizen für die Implementierung adaptiver Löser ungeeignet sind, entziehen sich diese Verfahren einer Vektorisierung oder einer auf Gebietspartitionierung basierenden Parallelisierung. Weil adaptive Verfahren jedoch unübertroffene numerische Eigenschaften aufweisen, müssen neue Parallelisierungskonzepte gefunden werden, um diese Verfahren auch auf parallelen Hochleistungsrechner einsetzen zu können.

Einen neuen Ansatz zur Parallelisierung adaptiver Verfahren auf diskretisierten Problemgebieten stellt der Einsatz extrem leichtgewichtiger Prozesse dar. Hierbei wird für jeden Punkt des Problemgebietes ein Prozeß erzeugt. Ausgehend von einer Startmenge an aktiven Prozessen, berechnet jeder aktive Prozeß den Wert des ihm zugewiesenen Gitterpunkts aus den Werten seiner Nachbarpunkte und aktiviert anschließend seine Nachbarprozesse, bevor er sich selbst suspendiert. Das Verfahren terminiert, wenn die Menge der aktiven Prozesse leer ist, d.h. alle Prozesse sich selbst suspendiert und keine anderen Prozesse mehr aktiviert haben. Die Aktivierung bzw. Suspendierung muß dabei nach bestimmten Regeln erfolgen, die eine konvergente Näherungslösung und Terminierung des Verfahrens garantieren. Der vorgestellte Ansatz stellt somit eine Parallelisierung der in [Rüde92] vorgestellten active set strategy dar.

1 Voll adaptives Rechnen mit leichtgewichtigen Prozessen

Zur Berechnung physikalischer Größen auf kontinuierlichen Problemgebieten wird zumeist das Gebiet diskretisiert und eine Näherungslösung für jeden diskreten Punkt errechnet. Die verwendeten Lösungsverfahren werden im folgenden in statische, adaptive und voll-adaptive Verfahren eingeteilt.

Bei *statischen Verfahren* wird nach einmal erfolgter Diskretisierung das Problemgebiet nach einem vorgegebenen Ablaufplan bearbeitet, bis eine hinreichend genaue Näherungslösung erzielt ist. Als Beispiel sind hier Varianten des Gauß-Seidel Lösers, die LR-Zerlegungen oder die CG-Verfahren genannt. Diese Verfahren weisen i.A. eine sehr hohe Datenparallelität auf und eignen sich daher sehr gut zum Einsatz auf Vektor- oder Feldrechnern. Durch eine Aufteilung des Problemgebietes auf die vorhandene Prozessoren kann eine effiziente Parallelisierung durchgeführt werden. Bei unregelmäßigen Problemgebieten wird jedoch eine Partitionierung immer schwieriger (NP-vollständiges Problem), weil die vorhandenen Punkte nicht nur gleichmäßig auf die Recheneinheiten verteilt werden müssen, sondern auch die Synchronisationspunkte zwischen den beteiligten Prozessoren minimiert werden müssen, um eine gute Effizienz zu erzielen.

Bei *adaptiven Verfahren* kann zwischen einzelnen Berechnungsphasen die Feinheit der Diskretisierung geändert werden. Diese bewirkt wesentlich bessere numerische Eigenschaften was Konvergenzgeschwindigkeit und Rechenaufwand betrifft.
Solange die Feinheit der Diskretisierung jeweils für das gesamte Problemgebiet verändert wird, gelten die gleichen Vektorisierungs- und Parallelisierungsmöglichkeiten wie bei den statischen Verfahren. So wird in der Strömungsmechanik [Schreck92] das Problemgebiet in einzelnen Blöcken auf die Prozessoren verteilt. Die Prozessoren führen dann Mehrgitterzyklen durch und lösen das lokale Teilproblem auf den einzelnen Blöcken mit einem ILU- oder CG-Löser. Zwischen den einzelnen Iterationen werden die Randwerte zwischen den Prozessoren ausgetauscht, wodurch nach mehreren Iterationen eine globale Näherungslösung erreicht wird. Hierbei lassen sich sehr gute Effizienzen auf eng- wie auch auf lose gekoppelten Parallelrechnern erzielen.
Werden jedoch im Laufe der Berechnung Teile des Problemgebietes unterschiedlich fein diskretisiert (um z.B. in der Nähe von Singularitäten eine höhe Punktedichte zu erreichen), ist eine Parallelisierung nur mit hohem Aufwand und mit Effizienzeinbußen möglich [Birken93]. Da die Diskretisierung jedoch nur zwischen festen Berechnungsphasen geändert wird, kann nach jeder Änderung der Diskretisierung in einer Reorganisationsphase eine neue Punkteverteilung sowie eine neue Berechnungs- bzw. Synchronisationsreihenfolge bestimmt werden. Die Einbußen an paralleler Effizienz werden jedoch zum Teil durch einen Gewinn in der numerischen Effizienz wieder aufgewogen.

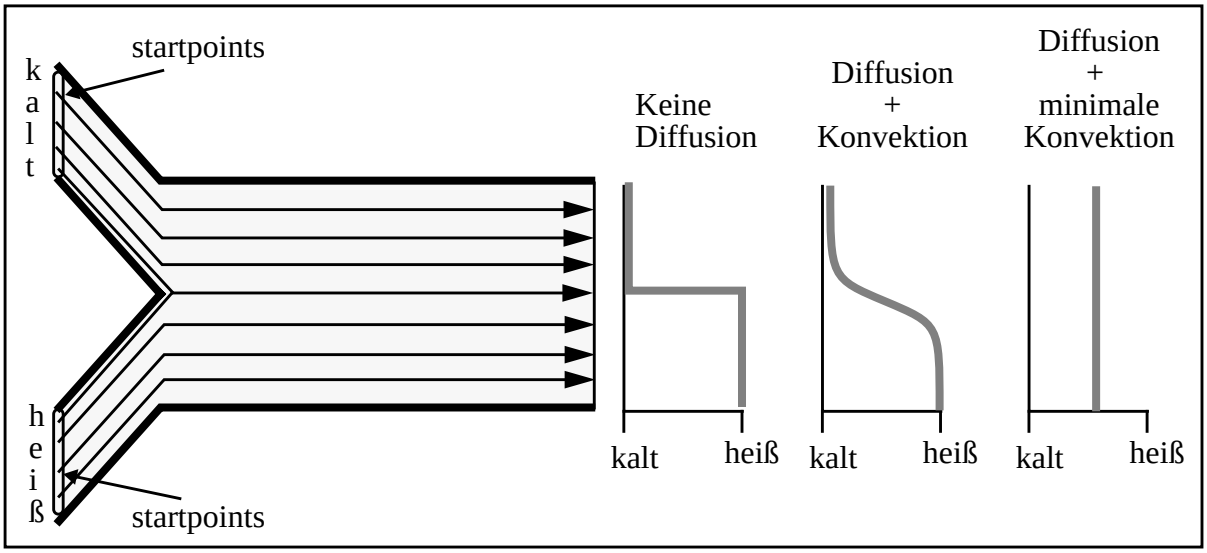
Voll-adaptive Verfahren steuern die Ablaufreihenfolge durch den Ausgang einzelner Berechnungschritte und benötigen dadurch eine minimale Anzahl an arithmetischen Operationen [Rüde92]. Durch die problemangepaßte dynamische Bestimmung der Abarbeitungsreihenfolge für die Gitterpunkte entziehen sich diese Verfahren allen Partitionierungsversuchen.

2 Die Active Set Strategie

Um adaptive Verfahren auf unregelmäßigen Gittern auf Monoprozessorarchitekturen zu implementieren, wurde die in [Rüde92] vorgestellte Active Set Strategie entworfen. Dabei werden aus der Menge aller Gitterpunkte durch ein Auswahlverfahren einige Punkte bestimmt, aus denen beim Start des Verfahrens die Menge der aktiven Punkte gewonnen wird.
Aus dieser Menge wird ein Punkt entfernt. Falls der lokale Fehler für diesen Punkt einen gewissen Schwellwert überschreitet, wird der Wert dieses Punktes an Hand der Werte seiner räumlichen Nachbarn neu berechnet. In diesem Fall werden auch alle Nachbarpunkte in die Menge der aktiven Punkte übernommen. Dieser Vorgang wiederholt sich so lange, bis es keinen aktiven Punkt mehr gibt.

```
func ActiveSetStrategy(grid,startpoints,threshold)
     aktiveset = startpoints;
     while(aktiveset ≠ Ø)
        select(point ∈ aktiveset ∈ grid)
        activeset = activeset \ point;
        if (error(point) > threshold)
            point = calculatepoint(point)
            aktiveset = aktiveset ∪ neighborpoints(point)
        end if
     end while
end func
```

Die Wirkungsweise dieses Verfahrens wird an einem Beispiel aus der Strömungmechanik deutlich.



Über zwei Anschlüsse fließt eine kalte und eine heiße Flüssigkeit in ein Rohr. Gesucht ist die Wärmeverteilung am Ausfluß des Rohres in Abhängigkeit von der Strömungsgeschwindigkeit und dem Diffusionskoeffizienten. Wählt man als Startpunkte die Punkte im Anschlußbereich,

so wird sich die Menge der aktiven Punkte im Laufe der Berechnung entlang der Strömungslinien bis zum Ausfluß verschieben. Je stärker die Diffusion relativ zur Strömung (Konvektion) ist, desto mehr wird sich die Menge der aktiven Punkte aufweiten. Bei geringer Diffusion wird die Menge hingegen kaum mehr Punkte enthalten als die Startmenge.
Wesentlich an diesem Verfahren ist die Tatsache, daß sich die Bearbeitungsreihenfolge dem Problem anpaßt, vorausgesetzt eine geeignete Startmenge wird gefunden. Diese läßt sich bei N Gitterpunkten in O(N) finden, indem man zunächst für alle Gitterpunkte den lokalen Fehler bestimmt und alle Punkte in die Startmenge aufnimmt, die oberhalb des Schwellwertes liegen.

2.1 Parallelisierung der Active Set Strategie

Im folgenden werden für die Active Set Strategie Parallelisierungskonzepte vorgestellt, die vom Programmiermodell her auf einem gemeinsamen Speicher beruhen. Die Entwicklung in der Rechnerarchitektur bei Hochleistungsparallelrechnern geht von den Rechnern mit verteiltem Speicher (Message Passing Architekturen) weg und hin zu Architekturen mit verteiltem gemeinsamem Speicher (virtual shared memory). Hierbei wird durch spezielle Verbindungshardware die Konsistenz des verteilten Speichers sichergestellt, so daß der Programmierer einen globalen Adreßraum auf einem virtuell gemeinsamen Speicher zur Verfügung hat. Der Zugriff auf den Speicher dauert jedoch je nach Lokalität der angesprochenen Datenbereiche unterschiedlich lange. Man spricht daher auch von NUMA (Non Uniform Memory Access)-Architekturen. Typische Vertreter stellen die Rechner KSR1 und CONVEX SPP dar.

2.1.1 Partitionierung des Active Set

Der klassische auf Partitionierung beruhende Ansatz zur Parallelisierung der Active Set Strategie läge in einem zyklenhaften parallelen Abarbeiten der Menge der aktiven Punkte:

```
func ActiveSetStrategy(grid,startpoints,threshold)
     aktiveset = startpoints;
     while(aktiveset ≠ Ø)
        subsets = partition(aktiveset,nprocessors)

        aktiveset = Ø
        for each subset ∈subsets in parallel
            new_activeset = calculate(subset,threshold)
            aktiveset = aktiveset ∪ new_activeset
        end for
    end while
end func

func calculate(subset, threshold)
    new_activeset= Ø
    while(subset ≠ Ø)
        select(point ∈ subset)
```

```
        subset = subset \ point;
        if (error(point) > threshold)
         point = calculatepoint(neighborpoints)
         new_aktiveset = new_aktiveset ∪ neighborpoints(point)
        end if
    end while
    return(new_aktiveset)
end func
```

Bei diesem Ansatz ist entscheidend, durch welche Datenstruktur die jeweiligen Mengen repräsentiert werden. Folgende Gesichtspunkte müssen bei der Wahl der Datenstruktur berücksichtigt werden:

- Die Datenstruktur muß einfach partitionierbar sein.
- Die Vereinigungsmenge muß leicht gebildet werden können.

Insbesondere bei der Bildung der Vereinigungsmenge ist darauf zu achten, daß dieser Vorgang möglichst parallel erfolgen kann. Daher müssen parallele Algorithmen und Datenstrukturen gefunden werden, die effiziente Mengenoperationen ermöglichen. Diese Algorithmen sind auch für die Verwaltung von Warteschlangen auf Muliprozessorsystemen von wesentlicher Bedeutung.

2.1.2 Repräsentation der aktiven Punkte durch aktive Threads

Der in dieser Arbeit vorgestellte Ansatz beruht auf dem Einsatz extrem leichtgewichtiger Prozesse (Threads) in Verbindung mit einer auf die Gitterpunkte verteilten Aktivitätsinformation. Hierbei wird für jeden Gitterpunkt ein leichtgewichtiger Prozeß (Thread) erzeugt, der zunächst einmal suspendiert wird. Eine Startmenge von Threads wird aktiviert. Jeder Thread führt den folgenden Zyklus aus:

```
func working_thread(myself, threshold)
    while (1)
        if (error(myself) > threshold)
         localpoint = calculatepoint(neighborpoints)
         activate(neighbors)
        end if
        suspend(myself)
    end while
```

Jeder aktive Prozeß berechnet den Wert des ihm zugewiesenen Gitterpunkts aus den Werten seiner Nachbarpunkte und aktiviert anschließend seine Nachbarprozesse, bevor er sich selbst suspendiert. Das Verfahren terminiert, wenn die Menge der aktiven Prozesse leer ist, d.h. alle Prozesse sich selbst suspendiert und keine anderen Prozesse mehr aktiviert haben.
Jeder aktive Prozeß kann auch neue Gitterpunkte und damit neue Threads erzeugen. Er muß je-

doch zuvor exklusiven Zugriff auf die Datenstruktur aller Nachbarn der neu generierten Gitterpunkte haben, um dort die neuen Threads in die Liste der zu aktivierenden Threads eintragen zu können.

Dieses Vorgehen bringt gegenüber den bisherigen Parallelisierungsansätzen eine Reihe von Vorteilen mit sich:

- Der Programmcode wird portabel. Er enthält keine rechnerabhängigen Parallelisierungskonzepte.
- Der Wissenschaftler muß sich nur um die Rechenroutinen und Aktivierungsregeln kümmern.
- Die Lastverteilung wird in ein Laufzeit-/Betriebssystem verlagert, das optimal an die zugrunde liegende Rechnerarchitektur angepaßt ist.

Die in diesem Abschnitt vorgestellte Implementierung adaptiver numerischer Verfahren mit leichtgewichtigen Prozessen, stellt hohe Anforderungen an das verwendete Laufzeit-/Betriebssystem, das für eine effiziente Verwaltung der Threads sorgen muß.

3 Systemanforderungen

Die Anforderungen, die an leichtgewichtige Prozesse zur Implementierung numerischer Verfahren gestellt werden, unterscheiden sich zum Teil erheblich von denen, die an Threads zur Implementierung von Systemprogrammen gestellt werden.
Folgende Threadoperationen sollten besonders effizient ablaufen:

- Thread Generierung und Terminierung
- Threadwechsel durch Suspendierung und Aktivierung
- Abfrage der gerade aktivierten Threads
- Synchronisationsoperationen (Mutex, Conditional-Mutex, Barriersync)

Auf eine Reihe von Mechanismen kann verzichtet werden:

- Verdrängendes Scheduling
  Wenn ein Prozessor Arbeit hat, sollte er diese auch zu Ende führen können. Da alle Threads zur Lösung desselben Problems beitragen, ist eine gleichmäßige Verteilung der Rechenleistung über alle aktiven Threads nicht nötig. Ein Threadwechsel sollte daher nur dann erfolgen, wenn ein Thread terminiert bzw. suspendiert, blockiert oder aktiviert wird.
- Prioritäten
  Alle Threads sind gleichberechtigt. Deshalb kann auf die Vergabe von Prioritäten verzichtet werden, da dies eine Lastverteilung auf die Prozessoren nur erschwert.

4 Zusammenfassung und Ausblick

Leichtgewichtige Prozesse eignen sich hervorragend, um adaptive Verfahren auf Rechnern mit gemeinsamem Speicher zu implementieren. Da die Lastverteilung durch ein Laufzeit/Betriebssystem erfolgt, muß der Wissenschaftler nicht mehr Lastverteilungsaspekte in seine Implementierung einbeziehen. Er muß nur noch eine geeignete Datenstruktur für jeden Gitterpunkt sowie Aktivierungsregeln für die leichtgewichtigen Prozesse bereitstellen. Das Laufzeitsystem übernimmt die Verwaltung dieser Threads. Wichtig ist hierbei, daß alle benötigten Operationen mit Threads sehr effizient ablaufen.

Für einen schnellen Threadwechsel wird vor allem ein minimaler Threadkontext benötigt. Zu diesem Kontext gehört neben einem eigenen Stack auch die Belegung der Prozessorregister. Durch Compilerdirektiven könnte erreicht werden, daß in der Nähe von besonderen Funktionsaufrufen, die Zahl der verwendeten Register minimal ist, so daß bei einem Wechsel nur wenige Register auf den Stack kopiert werden müssen.
Der Verwaltung der Stacks muß besonderes Augenmerk geschenkt werden, insbesondere der Anforderung und Freigabe von Stackspeicherbereichen bei der Generierung und Terminierung von Threads. Hier ist an eine Wiederverwertung von Speicherbereichen zu denken, bei der die beteiligen Prozessoren freigegebene Speicherbereiche nicht wieder als gemeinsamen freien Speicher an das Laufzeitsystem zurückgeben, sondern für neue lokale Speicheranforderungen zurückhalten.

Die Hauptproblematik liegt aber in der verteilten Verwaltung der Threadwarteschlangen. Hierbei handelt es sich im wesentlichen um das Einfügen und Entfernen von Elementen in disjunkten Mengen. Dabei muß darauf geachtet werden, daß unterschiedliche Prozessoren möglichst nicht gleichzeitig auf denselben Speicherbereich zugreifen und durch geschickt eingefügte “prefetch”-Operationen Speicherbereiche schon in den lokalen Speicher geholt werden, bevor diese vom Prozessor benötigt werden. Dieses Prefetching ist bei NUMA-Architekturen mit ihrem nicht uniformen Speicherzugriff sehr wesentlich, um den Prozessor nicht durch das Warten auf Daten all zu sehr abzubremsen.

Bislang wurden geeignete Laufzeitsysteme für numerische Anwendungen nur für symmetrischen Multiprozessoren entwickelt [Ande 89][Ande 92][Marsh 91][McCa 93]. Für NUMA-Architekturen gibt es bislang kaum Untersuchungen [LeBl 89][Ghos 93]. Der Entwicklung eines Laufzeitsystems mit einer leistungsfähigen Threadverwaltung für diese Rechnerklasse soll die weitere Forschung dienen.

Literaturverzeichnis

[Ande 89] T. Anderson, E. Lazowska, H. Levy, “The Performance Implication of Thread Management Alternatives for Shared-Memory Multiprocessors”, ACM Trans. on Comp. Vol. 38 No. 12, Dec. 1989

[Ande 92] T. Anderson, B. Bershad, E. Lazowska, H. Levy, “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”, ACM Trans. on Comp. Sys. Vol. 10 No. 1, Feb. 1992

[Birken93] K. Birken, “Ein Parallelisierungskonzept für adaptive, numerische Berechnungen”, Diplomarbeit IMMD III, March 1993

[Ghos 93] K. Ghosh, “Experimentation with Configurable, Lightweight Threads on a KSR Multiprocessor”, Georgia Institue of Technology: Technical report GIT-CC-93/ 37

[LeBl 89] T. LeBlanc , “Memory management for large-scale numa multiprocessors”, Department of Computer Science: Technical report*311

[Marsh 91] B.D. Marsch, “First-Class User-level Threads”, Proc. 13th Symp. on Operating Systems Principles, ACM, pp 110-121, 1991

[McCa 93] C. McCann, R. Vaswani, J. Zahorjan, “A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors”, ACM Trans on Comp. Sys. Vol. 11, May 1993

[Rüde92] Ulrich Rüde, “On the multilevel adaptive iterativ Method”, SIAM journal on scientific and statistical computing, Vol. 15, 1994

[Schreck92] E. Schreck, “Numerical Simulation of Complex Fluid Flows on MIMD Computers”, Report LSTM, Universität Erlangen-Nürnberg, 1992