Ein adaptives Lastverteilungsverfahren für Devide-and-Conquer-Algorithmen

H. Unger[*], B. Däne[†]

Abstract

In many cases the execution of a parallel program will be strongly influenced by data dependencies and processes running at the same time on the machine or network. These factors can not be predicted. Therefore powerful load balancing procedures are necessary in order to improve the run time behaviour. The disadvantage of this more efficient dynamic approach arises from a large communication or management overhead and delayed activities. In our contribution we intend to present an improved dynamic, decentral method for hierachical automation appliances using idle times of the processor and the communication system for doing so.

1 Einführung

Technische und technologische Prozesse erfordern neben einer entsprechenden Verarbeitungsleistung auch einen umfassenden Austausch von Informationen zwischen räumlich oder funktional getrennten Prozessen in einem engen zeitlichen Rahmen. Die entsprechende Software weist dabei folgende Merkmale auf:

- es existieren eine Reihe von parallel abarbeitbaren Aktivitäten
- diese Aktivitäten weisen eine Reihe von Abhängigkeiten auf und erfordern daher eine Anzahl von Kommunikationsaufrufen
- die Prozesse besitzen eine hierarchisch geordnete Struktur, wobei in der Abarbeitung in der Regel Wartezeiten auftreten

[*]University of Rostock, Germany; hunger@informatik.uni-rostock.de
[†]Technical University of Ilmenau, Germany; bdaene@theoinf.tu-ilmenau.de

- nur in wenigen Faellen ist die Bearbeitung der Prozesse an eine spezielle Maschine gebunden, lediglich Prozesse mit E/A-Operationen sind an bestimmte Maschinen gebunden.

Die genannten Merkmale haben in der Vergangenheit dazu geführt, daß in der Automatisierungstechnik parallele bzw. verteilte Architekturen zunehmend Verbreitung fanden bzw. historisch gewachsen sind. Analysiert man die Abarbeitung der Software auf diesen Maschine genauer, stellt man fest, daß die Rechenlast der einzelnen Maschine direkt vom Fluß der Daten, Materialien oder Informationen des Prozesses selbst abhängig ist und daher Lastspitzen sich oftmals von Verarbeitungsknoten zu Verarbeitungsknoten verlagern. Der Kontrollfluß wird zeitlich von folgenden Faktoren signifikant beeinflußt:

- der aktuellen Auslastung des technischen oder technologischen Prozesses bzw. einzelner seiner Stufen
- der Verfügbarkeit von E/A-Daten von den entsprechenden Geräten
- Laufzeitdatenabhängigkeiten, wie z.B. Laufzeit von Iterationsschleifen, Anzahl der erzeugten parallel arbeitenden Clients oder aber der Entscheidung über alternativ auszuführende Instruktionsfolgen.

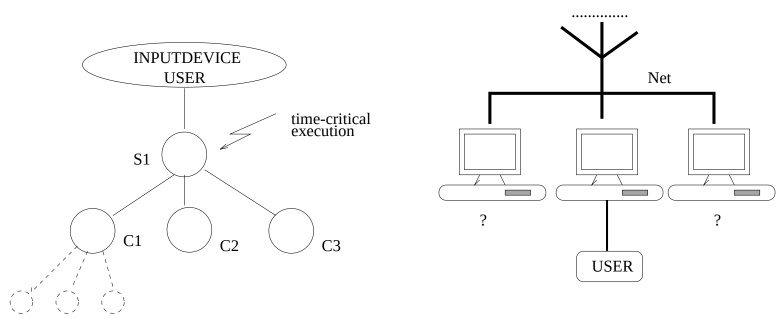

Abbildung 1: Ein Devide and Conquer System mit der verwendeten Hardware

Um den Einfluß der genannten Faktoren zur Laufzeit noch kompensieren zu können, wurden Load Balancing Verfahren entwickelt. Diese können die Abarbeitung des Programms zur Laufzeit an die jeweiligen Gegebenheiten des Computersystems anpassen. Vor- und Nachteile solcher Methoden sollen im flogenden Abschnitt unter besonderer Berücksichtigung der genannten Anwendungen (s. Abb. 1) diskutiert werden. Dabei ist zu beachten, daß in dem dargestellten Teile - und - Herrsche - Verfahren insbesondere die Abarbeitung der Serverprozesse zeitkritisch ist und das Laufzeitverhalten des gesamten Programms beeinflußt. Die damit verbundene Forderung, diese Prozesse auf einem möglichst leistungsfähigen Rechner mit möglichst wenig Last von anderen Prozessen zu bearbeiten, stellt dabei einen Ansatzpunkt für die Entwicklung unseres neuen Ansatzes dar.

2 Bekannte Load Balancing Verfahren

Die in der Literatur existierenden Load Balancing Verfahren, (s. [1] für einen Überblick), können grob in zwei Klassen eingeteilt werden: sogenannte 'non-preemptive'- bzw. 'preemptive'-Verfahren. Da erstere lediglich eine Plazierung von Prozessen bei ihrem Start im System erlauben (z.B. [7]), sind sie für die oben beschriebene Aufgabe ungeeignet. Preemptive Systeme dagegen erlauben die Suspendierung eines Prozesses nach seinem Start verbunden mit einer Verlagerung und Fortsetzung auf einem anderen Prozessor des Systems (z.B. [8]). Um Zeitpunkt und Zielprozesse für eine derartige Verlagerung zu ermitteln, werden entweder Vorinformationen über Prozeßstrukturen oder Laufzeitdaten wie z.B. Lebens- und CPU-Zeit der entsprechenden Prozesse herangezogen. Obwohl durch die genannten Verfahren Verbesserungen in der Laufzeit erzielt werden können, bleiben einige Probleme ungelöst. Diese betreffen:

- den hohen Verwaltungs- und Kommunikationsaufwand in einem i.a. recht langsamen Netzwerk (vgl. [4]),
- eine verspätete Reaktion, die erst durch existierende Disproportionen verurscaht wird ,
- die an einem möglicherweise unpassenden Punkt durchgeführte Prozeßmigration,
- Probleme in der Bestimmung von Prozessen, deren Verschiebung zu den gewünschten Effekten führt und
- die hohen Kosten einer Verlagerung eines Prozesses.

Aus der Sicht der Autoren sind viele der genannten Probleme vermeidbar, wenn von der festen Struktur des Parallelprogramms mit fixierten Prozessen und Kommunikationsstellen abgegangen wird (s.u.). Ferner erfordert ein wirksames Load Balancing die Nutzung sowohl von Laufzeit- als auch Vorinformationen. Im folgenden soll der von uns genutzte Ansatz einer flexibleren Strukturinterpretation näher erläutert werden, wobei wir auf eine Implementierung unter C und PVM [1] (s. auch [6]) Bezug nehmen.

3 Eine lastadaptive Programmiermethodik

3.1 Grundlagen

Die Erfahrungen der Autoren resultieren insbesondere aus der Implementierung von Petri-Netz-Modellen ([5]). Für unsere Untersuchungen soll zunächst folgende typische Struktur eines Parallelprogramms untersucht werden, die sich sowohl als Zeitdiagramm als auch Ausschnitt eines Petri-Netzes darstellen läßt (s. Abb. 2).

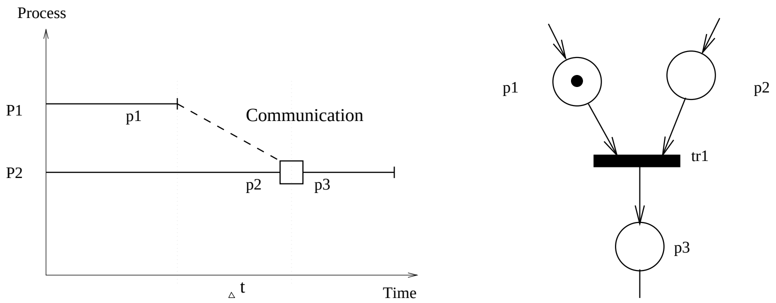

Abbildung 2: Ein System von 2 Prozessen

Es ist ersichtlich, daß die dargestellten 2 Prozesse $P1$ und $P2$ zu folgender Abarbeitungsfolge führen: Nach der Abarbeitung einiger Anweisungen erwartet der Prozeß $P2$ den Empfang von Daten, die durch den Prozeß $P1$ am Ende seiner Abarbeitung bereitgestellt werden. Nach Erhalt dieser Informationen kann $P2$ seine Arbeit fortsetzen, während $P1$ terminiert. Es ist auch sofort klar, daß derselbe Algorithmus umgesetzt wird, falls $P1$ die Fortsetzung der Arbeit übernimmt, während $P2$ beendet wird. Lediglich die

[1] **P**arallel **V**irtual **M**achine für UNIX Cluster, entwickelt vom Oak Ridge National Laboratory (siehe: Sunderam, V.S.: PVM.- in: Concurrency: Praktice and Experience, Dec. 1990, pp. 315-339, Atlanta 1990)

starren Strukturen der vorhandenen Programmiersprachen lassen derartige Optionen nicht zu, obwohl diese Flexibilität gleichfalls zu einer Lastbalancierung führen kann. Dies tritt genau in dem Fall ein, wenn jeweils derjenige Prozeß die Weiterbearbeitung übernimmt, der auf der Maschine mit der jeweils geringeren Last bearbeitet wird. Die dazu benötigten Informationen können zur Laufzeit mit geringem Aufwand ermittelt werden. Es soll gezeigt werden, daß eine derartige Anpassung möglich ist, falls folgende Voraussetzungen erfüllt werden:

- jede Maschine kann jeden Teil der Aufgabe bearbeiten;
- jede Ausführungsalternative ist im Rahmen der Struktur eines einheitlichen, universellen Prozesses auf jeder Maschine vorhanden;
- dieser Prozeß kann mit verschiedenen Eintrittspunkten im System mehrfach gestartet werden.

3.2 Technische Umsetzung

Unter diesen Voraussetzungen kann das neue Verfahren wie folgt beschrieben werden: Aus den Ergebnissen der Analyse des vorliegenden Algorithmus können durch statistisch bewertete Simulations- und Analyseergebnisse diejenigen Stellen ermittelt werden, an denen die oben beschreibenen Wartezeiten mit hinreichender Größe auftreten. Diese zu erwartende Zeit, die einer der Prozesse die Kommunikationsstelle eher erreicht als der/die anderen, sei in der Folge mit ${}_{\triangle}t$ bezeichnet.

Zur Herbeiführung einer Lastausgleichsreaktion werden an den ermittelten Stellen die Kommunikationsprotokolle gemäß Abbildung 3 abgeändert. Die Funktionalität dieser Änderung wird im folgenden unter der Annahme erklärt, daß $P1$ ohne Beschränkung der Allgemeinheit derjenige Prozeß ist, der die Kommunikationsstelle zuerst erreicht (alle Strukturen existieren symmetrisch für den anderen Fall). Dabei soll die Zeit, bis $P2$ ebenfalls den nötigen Funktionsaufruf ausführt, genutzt werden, um Operationen zur Ermittlung der günstigsten Weiterarbeit auszuführen. Das System wird dadurch zunächst nicht belastet [2].

[2] im reinen Multiprozessing oder aber im Multitasking, wenn die Anzahl der Prozesse wesentlich grösser als 1 ist

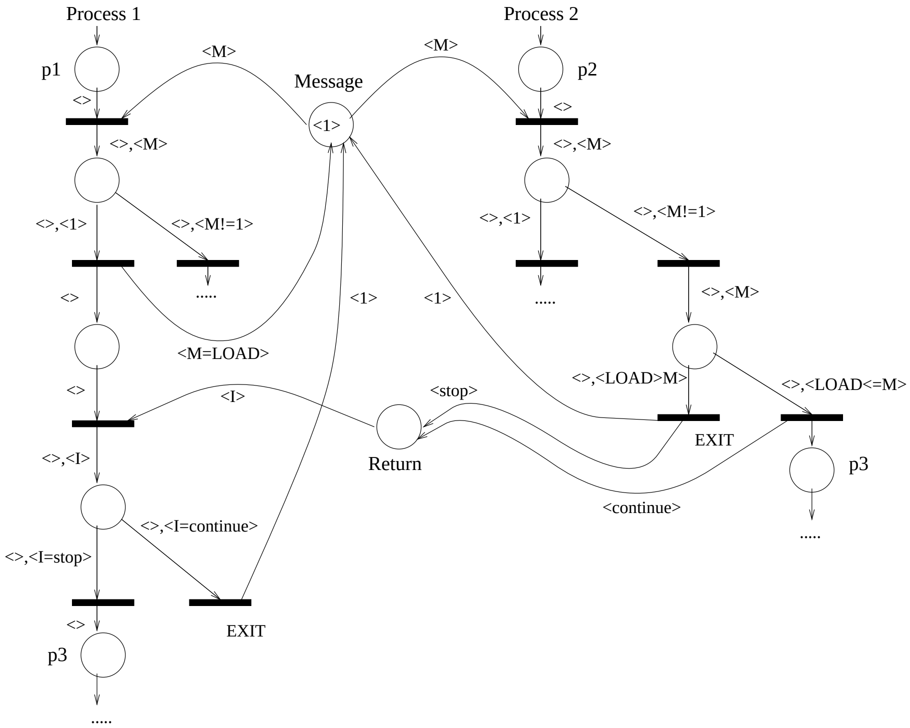

Abbildung 3: Beschreibung des Load Balancings

Zunächst liest $P1$ den Wert $Message$, der in diesem fall noch den Initialwert 1 enthält. Dieser Wert wird durch den aktuellen Lastfaktor der Maschine ersetzt, auf der $P1$ abgearbeitet wird. Diese Operation wird in der Zeit ${}_{\triangle}t$ ausgeführt, die dazu groß genug sein muß, um keine zus'atzliche Last zu erzeugen. Wenn $P2$ den Kommunikationspunkt erreicht, kann er die für ihn bereitliegende Information in $Message$ von $P1$ lesen und den Lastfaktor von dessen Maschine mit seinem eigenen vergleichen. Auf der Basis dieses Vergleichsergebnisses entscheidet $P2$, welcher Prozeß die Abarbeitung fortsetzen soll und teilt diese Entscheidung $P1$ im Rahmen der ohnehin notwendigen Nachricht mit. Während einer der Prozesse nun die Abarbeitung fortsetzen kann, stellt der andere den Initialwert von $Message$ wieder her und beendet seine Ausführung.

Betrachtet man die Kosten der beschriebenen Methode, so muß man zum einen in Rechnung stellen, daß die Nachrichtenlänge von $P2$ nach $P1$ sich vergrößert hat. Als zweiten Faktor muß man die Zeit angeben, die $P2$

```
// PVM - IMPLEMENTATION

....
load=get_load();            //return the load of the node
pvm_recv(-1, MESSAGE);
pvm_upkint($message,1,1);
if (message==1)             //if process passes at first
  { pvm_pkint(&load,1,1);
    pvm_send(-1, MESSAGE);
    pvm_recv(-1, RETURN);
    pvm_upkint(&reti,1,1);
    if (reti==CONTINUE)
      { message=1;
        pvm_pkint(&message,1,1);
        pvm_send(-1, MESSAGE);
        pvm_exit();
      }
  }
else                        //process is second
  { if (load>message)       //decides who continues
      { reti=STOP;
        pvm_pkint(&reti,1,1);
        pvm_send(-1, RETURN);
        message=1;
        pvm_pkint(&message,1,1);
        pvm_send(-1, MESSAGE);
        pvm_exit();
      }
    else
      { reti=CONTINUE;
        pvm_pkint(&reti,1,1);
        pvm_send(-1, RETURN);
      }
  }
p3:
.....
```

Abbildung 4: Umsetzung mit C und PVM

benötigt, um den Lastfaktor seiner Maschine zu ermitteln und die notwendige Vergleichsoperation auszuführen. Gegebenenfalls müssen beide Nachrichten den Inhalt von Laufzeitdaten zusätzlich enthalten, die der weiterarbeitende Prozeß von dem anderen Prozeß benötigt (falls das parallele Programm nicht aus sogenannten atomaren Prozessen besteht). Die Erfahrung zeigt jedoch, daß diese Kosten i.a. durch den erzielten Zeitgewinn kompensiert werden, insbesondere wenn Störprozesse dieselbe Maschine nutzen und $\triangle t$ groß genug ist.

Das beschriebene Verfahren zeichnet sich ferner dadurch aus, daß es für mehr als 2 Prozesse und Rendezvous-Kommunikationen verallgemeinert werden kann. Die notwendigen Änderungen können sowohl als Teil des Quellkodes

in ein fertiges paralleles Programm hineincompiliert werden oder durch den Programmierer berücksichtigt werden. Eine Nutzung als Teil einer neuen Funktion in der PVM-Kommunikationsbibliothek ist ebenso denkbar. Dabei ersetzt eine neue Funktion die alten pvm_send und pvm_recv Funktionen. Diese besitzt jedoch einen weiteren Aufrufparameter, der die ggf. zu übertragenden Laufzeitdaten mit berücksichtigt. Die alten Kommunikationsfunktionen können jedoch auch genutzt werden, falls Kommunikationsstellen existieren, an denen ${}_{\triangle}t$ nicht groß genug ist und daher von einer Lastverteilungabgesehen werden soll.
Gleichzeitig kann mit der beschriebenen Funktion auch ein Load Balancing an Nicht-Kommunikationsstellen durch den Nutzer erzwungen werden; dies stellt einen weiteren Vorteil unseres beschriebenen Verfahrens dar.

4 Zusammenfassung

In dem vorliegenden Beitrag wurde eine Variante eines durch den Programmierer zu beeinflussenden Laod Balancings vorgestellt, wobei sowohl Vor- als auch Laufzeitinformationen genutzt werden. Dazu werden neue Kommunikationfunktionen vorgestellt, die an bestimmten Stellen Kommunikation und Lastverteilung miteinander verbinden. Für den nötigen Overhead werden dabei im wesentlichen Leerlaufzeiten der CPU bzw. des Kommunikationsnetzwerkes effektiv genutzt. Auf diese Art und Weise werden zahlreiche Nachteile herkömmlicher Lastverteilungsverfahren vermieden und dynamische Laufzeitadaptionen selbst auf Systemen ohne entsprechende zusätzliche Software möglich.

Literatur

[1] M. Harchol-Balter; A.B. Downey: Exploiting Process Lifetime Distributions for Dynamic Load Balancing.-
Technical Report TR-95-021, ICSI, Berkeley, (1995)

[2] F. Douglas; J. Ousterhout: Transparent Process Migration: Design alternatives and the sprite implementation.- Software-Practice and Experience, 21(8), (1991)

[3] H.-U. Heiss; M. Schmitz: Distributed Load Balancing Using a Physical Analogy.-
Technical Report 5/92, University of Karlsruhe, Dept. of Informatics, (1992)

[4] G. Hipper, M. Klein, D. Tavangarian: Parallel Computing in Workstation Clusters using a Concurrent Network Architecture.-
ISSM94, Washington, D.C., (1994)

[5] H. Unger: Untersuchungen zur Implementierung von Petri-Netz-Modellen auf Mehrprozessorsystemen.-
Dissertation A, TU Ilmenau, (1994)

[6] S. White, A. Alund, V.S. Sunderam: Performance of the NAS Parallel Benchmarks on PVM based Networks.-
NASA, RNR 94-008, (1994)

[7] S. Zhou; X. Zheng; J. Wang; P. Delisle: Utopia: a load-sharing facility for large heterogeneous distributed computing systems.-
Software-Practice and Experience, 23(2), (1993)

[8] A. Barak; G. Shai; R.G. Wheeler: The MOSIX Distributed Operating System: Load Balancing for UNIX.-
Springer Verlag, Berlin, (1993)