Universität Frankfurt/Main
FB 20, Informatik
Wintersemester 1996/97

Seminar:

Parallelisierung und verteilte Ausführung funktionaler Programme
Prof. Dr. Manfred Schmidt-Schauß

Thema:

Worst-case-Scheduling paralleler, funktionaler Programme

von

Christian Hillebrecht

Nach einer Arbeit von

F. Warren Burton und V.J. Rayward-Smith
burton@cs.sfu.ca
vjrs@sys.uea.ac.uk

0. Überblick

1. Einleitung

Betrachtet man Implementierungen paralleler Programmiersprachen, so ist der die Laufzeit eines Programms entscheidend beeinflussende Faktor das sogenannte Scheduling der Prozesse. Die Aufgabe des Schedulings besteht in der Analyse des Programms, und - durch die damit gewonnenen Informationen - in der Zuweisung von Prozessen an Prozessoren, und insbesondere in der Festlegung der Ausführungsreihenfolge der einzelnen Prozesse pro Prozessor. Dabei dürfen keine Abhängigkeiten bezüglich der Ausführungsreihenfolge unter den Prozessen verletzt werden, und die zur Verfügung stehenden Hardwareressourcen sollen möglichst effizient eingesetzt werden. Da parallele, funktionale Programme im Vergleich zu parallelen, imperativen Programmen keine versteckten Abhängigkeiten enthalten können, sind für parallele, funktionale Programme eine größere Anzahl an Schedules möglich. Unter einer solchen versteckten Abhängigkeit könnte man sich beispielsweise eine globale Variable X vorstellen, die von einem Prozeß 1 verändert wird, und erst nach ihrer Veränderung von Prozeß 2 weiterverarbeitet werden darf, um einen sinnvollen Programmablauf zu gewähren. Diese globale Variable X verhindert damit alle Schedules, die Prozeß 2 vor oder simultan zu Prozeß 1 ausführen lassen. Daneben gibt es weitere Unterschiede zwischen parallelen, funktionalen und imperativen Programmen: Ein funktionales Programm ist per Definition immer deterministisch. Darüberhinaus zeigen insbesondere funktionale Programme ein starkes, dynamisches Verhalten, so daß eine Implementierung für parallele, funktionale Programme sowohl die dynamische Erzeugung von Prozessen als auch eine beliebige Synchronisation unter den Prozessen unterstützen muß. Da funktionale Sprachen auf einem hohen Abstraktionsniveau angesiedelt sind, wäre es natürlich wünschenswert, wenn die Implementierung der Sprache die Zuweisung der Prozesse an die Prozessoren, das Scheduling und eventuell auch die Aufteilung des Programms in Prozesse automatisch vornimmt, so daß sich der Programmierer auf sein Programm konzentrieren kann und sich nicht mit den Implementationsdetails beschäftigen muß.
Eine gängige, parallele Rechnerarchitektur besteht aus mehreren Prozessoren, die jeweils einen gewissen eigenen Speicher besitzen (sogenannte distributed-memory multiprocessors). Auf solchen Systemen ist es allgemein üblich, Prozesse nur auf

einem einzigen Prozessor auszuführen, d.h. nachdem die Ausführung eines Prozesses einmal begonnen hat, darf dieser Prozeß keinem anderen Prozessor zugewiesen werden. Programme, deren Prozesse diese Forderung erfüllen, sollen im folgenden als *verteilte Programme* (distributed programs) bezeichnet werden.
Im weiteren Verlauf dieser Ausarbeitung wird anhand einfacher Beispiele untersucht, ob ein vollautomatisches Scheduling für verteilte Programme prinzipiell durchführbar ist, das wesentliche Laufzeitverbesserungen gegenüber einem 1-Prozessor-System garantieren kann. Darüberhinaus wird geprüft, ob man die Prozesse optimal den Prozessoren zuweisen kann, wenn man nur die Informationen benutzt, die zum Zeitpunkt der Zuweisung zur Verfügung stehen. Anschließend wird untersucht, wie sich die Ergebnisse ändern, wenn die Einschränkung des verteilten Programms aufgehoben wird; wenn also Prozesse selbst nach ihrem Start noch verschoben, und somit auf mehreren Prozessoren ausgeführt werden dürfen.

2. Definitionen

Im folgenden wird ein *deterministisches, verteiltes Programm auf m Prozessoren* durch ein geordnetes Tripel $(A, \subset, f)$ dargestellt, in dem A die Menge der elementaren Aktionen - kurz Aktionen - , $\subset$ eine partielle Ordnung auf diesen Aktionen und $f: A \to \{1, 2, \ldots, m\}$ eine totale Funktion bezeichnet. Angenommen es gelte für $a_1, a_2 \in A: a_1 \subset a_2$, dann bedeutet dies, daß a_1 vor a_2 ausgeführt werden muß. $\subset$ stellt also eine Präzedenzrelation dar. Die Funktion f gibt die Zuordnung der Aktionen zu den Prozessoren an. Der Einfachheit halber nimmt man an, daß jede Aktion in einer Zeiteinheit durch einen Prozessor ausgeführt werden kann. Was in einer höheren Programmiersprache üblicherweise als *Prozeß* oder *Task* bezeichnet wird, ist in diesem Zusammenhang nichts anderes als eine Menge von Aktionen. Unter einem *sequentiellen Prozeß* versteht man eine Kette von Aktionen, d.h. eine Menge von Aktionen, so daß für je zwei Aktionen a_1 und a_2 entweder $a_1 \subset a_2$ oder $a_2 \subset a_1$ gilt.
Ein *Schedule* für $(A, \subset, f)$ ist eine totale Funktion $s: A \to \{1, 2, \ldots\}$, für die gilt:
1. $f(a_1) = f(a_2) \wedge s(a_1) = s(a_2) \Rightarrow a_1 = a_2$.
2. $a_1 \subset a_2 \Rightarrow s(a_1) < s(a_2)$.

Ein Schedule ist eine Art "Zeitfunktion", die jeder Aktion ihren Ausführungsstartzeitpunkt zuordnet. Demnach bedeutet die erste Forderung, daß keine zwei Aktionen zur gleichen Zeit auf dem gleichen Prozessor ausgeführt werden dürfen. Die zweite Bedingung repräsentiert die Einhaltung der partiellen Ordnung $\subset$ durch den Schedule.
Die *Länge eines Schedules* ist definiert als $length(s, (A, \subset, f)) =_{def} \max\{s(a) \mid a \in A\}$ und ist somit ein Maß für die Ausführungsdauer eines Programms bei gegebenem Schedule s. Falls $(A, \subset, f)$ aus dem Kontext eindeutig hervorgeht, schreibt man auch kurz $length(s)$. Ein *optimaler Schedule* für $(A, \subset, f)$ ist ein Schedule s, so daß $length(s) \le length(t)$ für alle Schedules t für $(A, \subset, f)$ gilt. Ein Schedule heißt *arbeitserhaltend* (work conserving), wenn sich kein Prozessor im Leerlauf befindet, solange noch etwas zur Ausführung vorhanden ist.
Um die Güte eines Schedules für ein paralleles Programm zu bewerten, eignet sich der Wert des *Speedups*. Dabei versteht man bei m zur Verfügung stehenden

Prozessoren unter dem Speedup $S(m)$ den Quotienten aus der Laufzeit des Programms mit einem Prozessor T_1 und der Laufzeit bei m Prozessoren T_m, $S(m) =_{def} \frac{T_1}{T_m}$.
Für die folgenden Betrachtungen gelte zunächst die Annahme, daß jeder Prozeß genau einem Prozessor zugeordnet ist; es gibt also genauso viele Prozessoren wie Prozesse. Ein Prozeß bezeichnet in diesem Zusammenhang die Menge der Aktionen, die einem bestimmten Prozessor zugeordnet sind. Dabei werden an die Prozesse selbst keine weiteren Anforderungen gestellt.
Viele klassische Scheduling-Algorithmen benötigen zum Zeitpunkt, in dem der Schedule generiert wird, umfassende Informationen über die Anzahl der zu bearbeitenden Aktionen und über ihre Synchronisation. In der Praxis stehen einem Scheduler für verteilte Programme allerdings nur begrenzte Informationen zur Verfügung. Wäre der Scheduler beispielsweise in der Lage, die Anzahl der Aktionen in einem Prozeß und damit die Ausführungsdauer dieses Prozesses vor der Ausführung zu ermitteln, dann könnte man bestimmen, wie lange das gesamte Programm höchstens laufen dürfte. Dadurch könnte man schließlich das Terminierungsverhalten des Programms bestimmen. Dies würde aber bedeuten, daß man das unentscheidbare Halteproblem plötzlich doch entscheiden könnte. Dieser Widerspruch führt zu der Annahme, daß der Prozessor wie auch der Scheduler sehr wahrscheinlich nicht die Anzahl der Aktionen kennt, die ihm zur Ausführung zugewiesen wurden. Die Anzahl der momentan ausführbaren Aktionen ist ihm aber natürlich bekannt. Hinzu kommt, daß der Scheduler nicht vorausberechnen kann, wie die einzelnen Aktionen synchronisiert sind, wenn der Programmablauf - und damit auch die Synchronisation unter den Prozessen - durch die Existenz von Bedingungsausdrücken (if, case, ...) vor der Ausführung nicht eindeutig festgelegt werden kann. Aus beiden genannten Gründen gelte folgende Annahme:

Dem Scheduler erscheinen alle ausführbaren Aktionen ununterscheidbar.

In diesem Zusammenhang sollte ein Ergebnis nicht unerwähnt bleiben, das das obige Modell um verschiedene Typen von Aktionen und entsprechenden Prozessoren erweitert. Jaffe [2] zeigte, daß mit k Typen von Aktionen und m_i gleichen Prozessoren für Aktionen vom Typ i $(1 \le i \le k)$ die Länge ω jedes arbeitserhaltenden Schedules höchstens um den Faktor $k + 1 - 1/\max\{m_1, m_2, \ldots, m_k\}$ größer ist, als die Länge eines optimalen Schedules ω_{opt} (Beweis siehe Anhang). Das obige Modell ist demnach ein Spezialfall von Jaffes Modell, mit $k = m$ (m = Prozessorenanzahl) und $m_1 = m_2 = \cdots = m_m = 1$. Damit ergibt sich die triviale Aussage $\omega \le m \cdot \omega_{opt}$. Trivial deshalb, da sowohl ω als auch ω_{opt} als Laufzeiten interpretiert werden können, und da jeder beliebige, arbeitserhaltende Schedule bei m Prozessoren eine höchstens m-mal schlechtere Laufzeit als ein optimaler Schedule liefern kann.

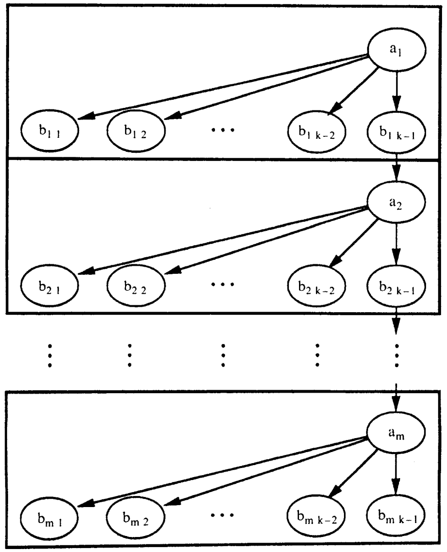


Abbildung 1(a). Beispiel 1.

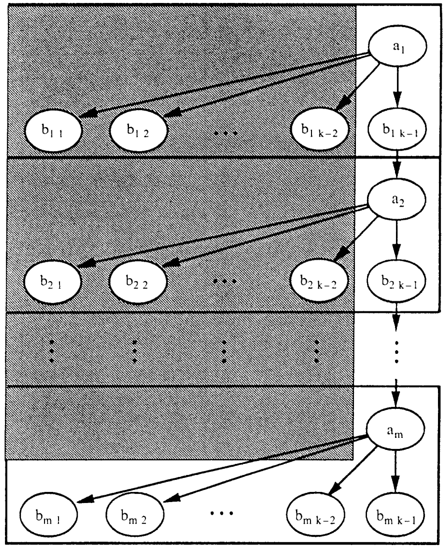


Abbildung 1(b). Beispiel 1 - Best-case.

3. Worst-case-Speedup verteilter Programme

In diesem Kapitel wird der Worst-case-Speedup zweier deterministischer, verteilter Programme ermittelt. Die Prozessorenanzahl beträgt jeweils m, d.h. zur Ausführung des Programms sind maximal m Prozessoren verfügbar. Jedes Programm wird durch ein Diagramm dargestellt, wobei einzelne Aktionen durch Ovale repräsentiert werden. Durch Pfeile zwischen den Ovalen wird eine Relation auf den Aktionen definiert. Die partielle Ordnung $\subset$ (Präzedenzrelation) läßt sich dann als transitive Hülle dieser Relation ablesen. Die Aktionen, die in einem gemeinsamen Kasten zusammengefaßt sind, sind einem bestimmten Prozessor zugeordnet, und stellen somit einen Prozeß dar.

Beispiel 1 (siehe Abbildungen 1(a) und 1(b)): Der erreichbare Speedup dieses Beispielprogramms hängt entscheidend von der Reihenfolge ab, in der die ausführbaren Aktionen gescheduled werden.
<u>Worst-case</u>: Werden die Aktionen von links nach rechts gescheduled, so führt dies zu einer sequentiellen Abarbeitung des Programms: Zunächst wird a_1, dann $b_{11}, b_{12}, \ldots, b_{1\,k-1}$, danach $a_2, b_{21}, \ldots, b_{2\,k-1}$, usw. ausgeführt. Die Laufzeit beträgt mit k Aktionen pro Prozessor $m \cdot k$ Zeiteinheiten, da m Prozessoren und damit auch m Prozesse vorhanden sind. Dies ist offenbar die schlechtest-mögliche Laufzeit, wenn man von solchen Schedules absieht, die Prozessoren unnötig unbeschäftigt lassen. Damit ergibt sich für den Speedup $S(m) = \frac{mk}{mk} = 1$.
<u>Best-case</u>: Werden die ausführbaren Aktionen aber von rechts nach links gescheduled, dann müssen nur noch die Aktionen in der rechten Spalte und in der untersten Zeile sequentiell ausgeführt werden (siehe Abbildung 1(b)). Nun werden die Aktionen in folgender Reihenfolge ausgeführt: $a_1, b_{1\,k-1}, b_{1\,k-2}$ und $a_2, b_{1\,k-3}$ und $b_{2\,k-1}$,

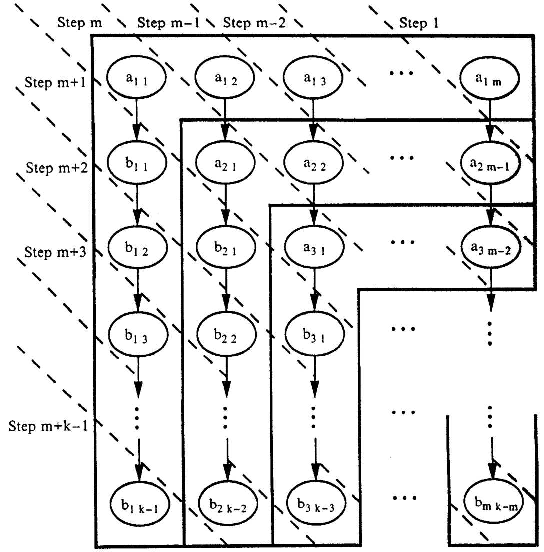


Abbildung 2(a). Beispiel 2 - Best-case.

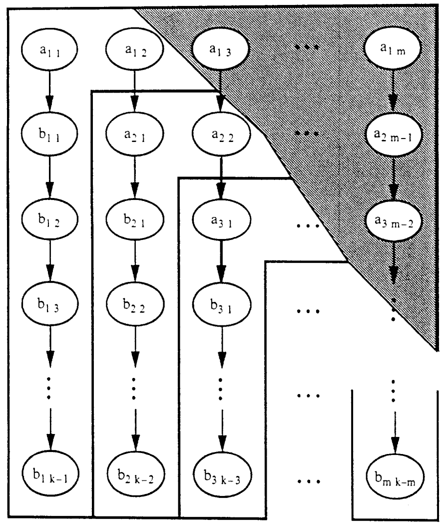


Abbildung 2(b). Beispiel 2 - Worst-case.

$b_{1\,k-4}$ und $b_{2\,k-2}$ und a_3, usw. (das "und" zwischen Aktionen soll andeuten, daß die betreffenden Aktionen gleichzeitig ausgeführt werden). In der Zeit, in der diese $2(m-1)+k = 2m+k-2$ Aktionen (rechte Spalte, unterste Zeile) abgearbeitet werden, sind auch die übrigen Aktionen, d.h. die grau-unterlegten, vollständig bearbeitet. Allein in der Zeit, in der der letzte Prozessor aktiv ist (er ist der letzte Prozessor, der gestartet, und der letzte, der beendet wird), könnten alle grau-unterlegten Aktionen ausgeführt werden. Damit beträgt der Speedup $S(m) = \frac{mk}{2m+k-2}$ und es gilt:

$$\lim_{k\to\infty} S(m) = \lim_{k\to\infty} \frac{mk}{2m+k-2} = \lim_{k\to\infty} \frac{m}{\frac{2m}{k}+1-\frac{2}{k}} = \frac{m}{0+1-0} = m.$$

D.h. mit zunehmender Anzahl an Aktionen pro Prozessor strebt der Speedup gegen die Prozessorenanzahl m, also gegen den bestmöglichen Speedup.
Der schlechte Worst-case-Speedup liegt in der Tatsache begründet, daß jeder Prozessor eine einzige kritische Aktion $b_{i\,k-1}$ $(1 \le i \le k)$ besitzt, die alle folgenden Prozessoren blockiert. Da im 2. Abschnitt angenommen wurde, daß dem Scheduler alle ausführbaren Aktionen gleich erscheinen, kann dieser nicht die eine kritische unter den $k-1$ Aktionen herausfinden. Dies bewirkt jedoch, daß man keine zufriedenstellende Laufzeit und damit keinen anständigen Speedup garantieren kann.
Beispiel 1 läßt sich darüberhinaus als eine Art Musterbeispiel für eine Vielzahl von Programmstrukturen ansehen, die alle am prinzipiell gleichen Problem scheitern, daß eine kritische Aktion das Blockieren anderer Prozessoren verursacht. Allerdings zeigt die Behandlung von Beispiel 1, daß - unter obiger Annahme - keine vernünftigen Einschränkungen existieren, dieses Blockieren zu verhindern.

Beispiel 2: Das folgende Beispiel (siehe Abbildungen 2(a) und 2(b)) repräsentiert Programmstrukturen mit einem höheren Maß an Synchronisation unter den Prozessen.

Best-case: Im bestmöglichen Fall werden die Aktionen von rechts nach links gescheduled. Gemäß Abbildung 2(a) können alle Aktionen auf einer Diagonalen in einem Schritt bearbeitet werden. Um das gesamte Programm auszuführen, sind demnach $m+k-1$ Schritte erforderlich. Da es sich insgesamt um $m \cdot k$ Aktionen handelt, beträgt die sequentielle Laufzeit $m \cdot k$. Daraus ergibt sich ein Speedup von $\frac{mk}{m+k-1}$, der für wachsendes k gegen die Prozessorenanzahl m strebt:

$$\lim_{k\to\infty} S(m) = \lim_{k\to\infty} \frac{mk}{m+k-1} = \lim_{k\to\infty} \frac{m}{\frac{m}{k}+1-\frac{1}{k}} = \frac{m}{0+1-0} = m.$$

Worst-case: Werden die ausführbaren Aktionen allerdings von links nach rechts gescheduled, so ändert sich der Speedup deutlich. In diesem Fall werden die Aktionen außerhalb des grau-unterlegten Dreiecks sequentiell ausgeführt (siehe Abbildung 2(b)). Die Reihenfolge lautet nämlich: $a_{11}, b_{11}, \ldots, b_{1\,k-1}, a_{12}, a_{13}$ und a_{21}, a_{14} und $b_{21}, \ldots, a_{1\,m}$ und $b_{2\,k-2}, a_{22}, a_{23}$ und a_{31}, a_{24} und $b_{31}, \ldots, a_{2\,m-1}$ und $b_{3\,k-3}$, usw. . Die Anzahl der Aktionen im grauen Dreieck läßt sich aus Abbildung 2(b) abzählen: $(m-2)+(m-3)+\cdots+1 = \Sigma_{i=1}^{m-2} i = \frac{1}{2}(m-2)(m-1)$. Daraus folgt, daß sich außerhalb des grauen Dreiecks $mk-\frac{1}{2}(m-2)(m-1)$ Aktionen befinden müssen. Die Ausführungsdauer beträgt demzufolge $mk-\frac{1}{2}(m-2)(m-1)$ Zeiteinheiten. Für den Speedup ergibt sich nun $S(m) = \frac{mk}{mk-\frac{1}{2}(m-2)(m-1)}$, der für k gegen Unendlich seinerseits gegen 1, also den schlechtest-möglichen Speedup, konvergiert:

$$\lim_{k\to\infty} S(m) = \lim_{k\to\infty} \frac{mk}{mk-\frac{1}{2}(m-2)(m-1)} = \lim_{k\to\infty} \frac{m}{m-\frac{1}{2k}(m-2)(m-1)} = \frac{m}{m-0} = 1.$$

Auch das zweite Beispiel zeigt ein sehr schlechtes Worst-case-Verhalten bezüglich Laufzeit und damit auch Speedup, ohne daß man irgendwelche vernünftigen Lösungsansätze erkennen kann, soetwas zu verhindern. Zwei Hoffnungen bleiben aber noch: Möglicherweise befindet sich der Average-case näher am Best- als am Worst-case, und das Problem des schlechten Speedups tritt somit nicht allzu häufig auf. Auch könnte sich der Worst-case-Speedup verbessern, wenn man die Prozesse intelligenter den Prozessoren zuweist. Diese zwei Aspekte werden in den beiden folgenden Abschnitten näher betrachtet.

4. Durchschnittlicher Speedup verteilter Programme

Da es kaum möglich ist, die durchschnittliche Struktur eines verteilten Programms zu bestimmen, kann man keine Wahrscheinlichkeitsfunktion definieren, die jedem realen Programm bzw. dessen zugrundeliegender Struktur eine Wahrscheinlichkeit sinnvoll zuordnet. Stattdessen könnte man höchstens eine Annahme über die Wahrscheinlichkeit treffen, mit der der Scheduler unter allen ausführbaren Aktionen eine bestimmte auswählt. So kann man in naheliegender Weise annehmen, daß der Scheduler für ihn ununterscheidbare Aktionen mit der gleichen Wahrscheinlichkeit auswählt, und diese Aktionen in zufälliger Reihenfolge ausführen läßt. Für das erste Beispiel aus dem vorigen Abschnitt bedeutet dies, daß jeder Prozessor im Durchschnitt die Hälfte der $k-2$ unkritischen Aktionen ausführt, bevor er die einzige kritische Aktion bearbeitet. Diese Methode würde

$$\underbrace{k}_{m\text{–ter Prozessor}} + \underbrace{2(m-1)}_{\text{rechte Spalte}} + \underbrace{\tfrac{1}{2}(k-2)(m-1)}_{\text{erwartete Anzahl unkritischer Aktionen}} = \tfrac{1}{2}k + m - 1 + \tfrac{1}{2}mk$$

Schritte für die Bearbeitung des gesamten Programms erfordern. Damit hätte man einen Speedup $S(m)$ von $\frac{mk}{\frac{1}{2}k+m-1+\frac{1}{2}mk}$ erreicht, der für k gegen Unendlich unter 2 liegt:

$$\lim_{k\to\infty} S(m) = \lim_{k\to\infty} \frac{mk}{\frac{1}{2}k+m-1+\frac{1}{2}mk} = \lim_{k\to\infty} \frac{m}{\frac{1}{2}+\frac{m}{k}-\frac{1}{k}+\frac{1}{2}m} = \frac{m}{\frac{1}{2}+0-0+\frac{1}{2}m} = 2\frac{m}{1+m} < 2.$$

Dieses Ergebnis bedeutet aber, daß sich der Speedup im Mittel nicht wesentlich besser verhält, als im Worst-case.

5. Prozeß-Verteilung

In den vorangegangenen Abschnitten wurde davon ausgegangen, daß alle Prozesse, und mit ihnen auch alle Aktionen, vor Ausführung des Programms den Prozessoren zugewiesen werden. Nun wird nur noch gefordert, daß die Menge aller Aktionen in gewisse Prozesse aufgeteilt werden, und daß alle Aktionen eines bestimmten Prozesses auf dem gleichen Prozessor ausgeführt werden müssen. Die einzelnen Prozesse dürfen hingegen beliebig und unabhängig voneinander den Prozessoren zugeteilt werden. Zusätzlich soll in diesem Abschnitt die Einschränkung fallen, daß jede Aktion genau eine Zeiteinheit beansprucht. So wird es im folgenden Beispiel Aktionen geben, die nur eine Zeiteinheit zu ihrer Ausführung in Anspruch nehmen, und solche, die k Zeiteinheiten benötigen. Möchte man weiter bei der einen Zeiteinheit pro Aktion bleiben, so stelle man sich die k Zeiteinheiten kostenden Aktionen als eine Kette aus k Aktionen vor, die jeweils nur eine Einheit brauchen.

Beispiel 3:

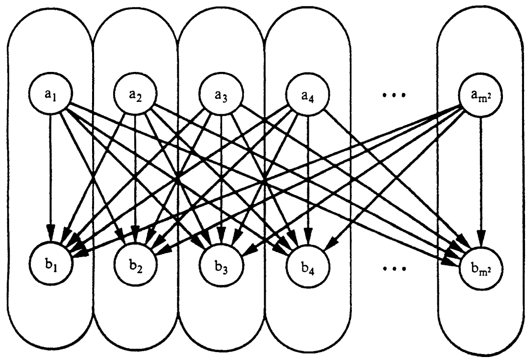


Abbildung 3. Beispiel 3.

Abbildung 3 zeigt die Struktur eines Programms, das auf m Prozessoren ausgeführt werden soll, und aus insgesamt m^2 Prozessen besteht. Jeder der m^2 Prozesse setzt sich aus einer a-Aktion und einer b-Aktion zusammen. Aufgrund der durch die Pfeile definierten Präzedenzrelation unter den Aktionen, darf keine b-Aktion begonnen und ausgeführt werden, bevor nicht alle a-Aktionen erledigt worden sind. Jede a-Aktion erfordere genau eine Zeiteinheit, wohingegen jede b-Aktion

entweder vom Typ *gut* oder vom Typ *schlecht* ist. Jede gute b-Aktion braucht zur Ausführung wie jede a-Aktion nur eine Zeiteinheit. Eine schlechte b-Aktion beansprucht allerdings k Zeiteinheiten, mit einem beliebigen, großen k. Ein Prozeß, dessen b-Aktion gut bzw. schlecht ist, wird im folgenden als guter bzw. schlechter Prozeß bezeichnet. Unter den m^2 Prozessen seien m schlecht und $m(m-1)$ gut.
Zur weiteren Betrachtung dieses Beispiels nimmt man folgendes an: Bevor die Ausführung eines Prozesses beginnt, muß dieser einem Prozessor zugewiesen worden sein. Zusätzlich wird angenommen, daß der Scheduler nicht entscheiden kann, ob ein bestimmter Prozeß gut oder schlecht ist, bevor nicht die Ausführung der b-Aktion begonnen hat. Die letzte Annahme liegt wieder in der Nichtentscheidbarkeit des Halteproblems begründet. Denn wäre es möglich, vor Ausführung einer b-Aktion deren Typ zu ermitteln, könnte man die Laufzeit eines jeden Prozesses errechnen, und damit schließlich entscheiden, ob die Ausführung des Programms terminiert oder nicht. Da gemäß Programmstruktur alle a-Aktionen vor jeder b-Aktion ausgeführt sein müssen, müssen alle Prozesse ihrem entsprechenden Prozessor zugeordnet sein, bevor der Typ eines jeden Prozesses bekannt ist.
<u>Best-case</u>: Im obigen Beispiel erhält jeder Prozessor im bestmöglichen Fall einen schlechten und $m-1$ gute Prozesse, so daß zur Ausführung des Programms insgesamt $k+1+2(m-1)=2m+k-1$ Zeiteinheiten erforderlich werden. Da die sequentielle Bearbeitung $m^2\cdot 1+m(m-1)\cdot 1+m\cdot k = m(2m+k-1)$ Zeiteinheiten kostet, führt dies zu einem Speedup von $S(m)=\frac{m(2m+k-1)}{2m+k-1}=m$.
<u>Worst-case</u>: Im schlechtest-möglichen Fall sieht dies ganz anders aus: Da insgesamt m^2 Prozesse auf m Prozessoren zu verteilen sind, existiert mindestens ein Prozessor, der mindestens m Prozesse ausführen muß (Schubladenprinzip !). Bekommt dieser Prozessor nun alle m schlechten Prozesse zugewiesen, dann beträgt die Ausführungsdauer des Programms $m(k+1)$ Zeiteinheiten. Dies ergibt einen Speedup von $S(m)=\frac{m(2m+k-1)}{m(k+1)}=\frac{2m+k-1}{k+1}$, der für k gegen Unendlich gegen 1 geht:

$$\lim_{k\to\infty} S(m) = \lim_{k\to\infty}\frac{2m+k-1}{k+1} = \lim_{k\to\infty}\frac{\frac{2m}{k}+1-\frac{1}{k}}{1+\frac{1}{k}} = \frac{0+1-0}{1+0} = 1.$$

Dieses Beispiel zeigt, daß es - ähnlich wie beim Scheduling - auch bei der Prozeß-Verteilung Situationen gibt, in denen Programme einen Speedup gleich der Prozessorenzahl, praktisch keinen Speedup oder einen Wert irgendwo dazwischen erfahren. Für verteilte Programme ist es aber unmöglich, einen Speedup "in der Nähe von m" zu garantieren. In Beispiel 3 wird nichts übrig bleiben, als die einzelnen Prozesse zufällig den verschiedenen Prozessoren zuzuweisen, und zu hoffen, daß dann ein akzeptabler Speedup herauskommt.

6. Scheduling auf Shared-Memory-Systemen

Die bisherigen Probleme basieren auf der Tatsache, daß die einzelnen Aktionen an bestimmte Prozessoren fest zugewiesen sind. In allen Worst-case-Beispielen existiert mindestens ein Prozessor, der die Wahl zwischen verschiedenen, ausführbaren Aktionen hat, während andere Prozessoren unbeschäftigt warten.

Im wesentlichen kann man zwei verschiedene Ursachen für Schedulingprobleme bei verteilten Programmen verantwortlich machen, die sich aus der anfangs gestellten Forderung ergeben, daß ein Prozeß auf einem einzigen Prozessor ausgeführt werden muß. Die erste Ursache macht sich bemerkbar, wenn ein Prozeß interne Parallelität enthält. Hierbei könnten mehrere Aktionen auf verschiedenen Prozessoren nebeneinander ausgeführt werden, anstelle einer sequentiellen Abarbeitung in beliebiger Reihenfolge auf nur einem Prozessor (siehe Beispiel 1). Die zweite Ursache besteht in der Ansammlung einer großen Menge an Zeitanforderungen durch einen Prozessor, der durch einen seiner Prozesse in den Wartezustand versetzt wurde, weil dieser Prozeß seinerseits auf das Ende eines Prozesses eines anderen Prozessors warten muß. Die aufgestockten Zeitanforderungen müssen dann aber später abgearbeitet werden, was die Ausführung des Gesamtprogramms verzögert. Ein Prozessor ohne momentan ausführbare Aktionen kann also noch eine Menge beliebiger Größe an aufgeschobener Arbeit besitzen.
Die eben geschilderten Probleme kann man vermeiden, wenn man die Forderung fallen läßt, daß jeder Prozeß - und damit alle in ihm enthaltenen Aktionen - nur auf einem einzigen Prozessor ausgeführt werden darf. Deshalb wird im weiteren allen Prozessoren erlaubt, zu jedem Zeitpunkt jeden Prozeß auszuführen. Unter dieser Voraussetzung gilt, daß jeder arbeitserhaltende Schedule (mit Länge ω) weniger als doppelt so lang wie der beste Schedule (mit Länge ω_{opt}) ist ($\omega < 2\omega_{opt}$). Dieses Ergebnis läßt sich beispielweise aus dem in Abschnitt 2 erwähnten Theorem von Jaffe folgern ($\omega \le (k+1-1/\max\{m_1, m_2, \dots, m_k\})\cdot\omega_{opt}$), da nur noch ein Typ von Aktionen existiert ($k=1$), die auf allen verfügbaren $m_1 = m$ Prozessoren ausgeführt werden dürfen. Dann ergibt sich nach Jaffe:

$$\omega \le (1+1-1/\max\{m\})\cdot\omega_{opt} = (2-\tfrac{1}{m})\cdot\omega_{opt} < 2\omega_{opt}.$$

Somit sind alle Probleme bezüglich des Worst-case-Speedups gelöst.

Doch man kann noch mehr zeigen: Sind alle Prozesse sequentiell, dann läßt sich der erreichbare Speedup bei m Prozessoren mittels der Prozessorenanzahl und der durchschnittlichen Parallelität abschätzen. Dabei versteht man unter der durchschnittlichen Parallelität den Speedup, der sich mit unbegrenzt vielen Prozessoren erreichen läßt. Bei Eager et al. [3] ist zu finden, daß für ein Programm mit durchschnittlicher Parallelität p, und mit einer Laufzeit T_1 bzw. T_m auf einem bzw. m Prozessor/en für den Speedup $S(m)=T_1/T_m$ gilt

$$\tfrac{1}{2}\min\{p,m\} \le \frac{mp}{m+p-1} \le S(m) \le \min\{p,m\}$$

sofern der Schedule arbeitserhaltend ist. Dabei ist die untere Schranke $\frac{mp}{m+p-1}$ für $S(m)$ sogar bestmöglich, d.h. wird in einigen Situationen tatsächlich angenommen. Da der Beweis dieses Theorems relativ umfangreich ist, erfolgt er getrennt im Anschluß an diesen Abschnitt.
Das Theorem besagt, daß der erreichbare Speedup $S(m)$ gegen p bzw. m strebt, falls $m \gg p$ bzw. $m \ll p$ ist. Gilt beispielsweise $p > m$ für ein bestimmtes Programm, so garantiert das Theorem, daß - unter den genannten Voraussetzungen - der Speedup mindestens $\frac{1}{2}\min\{p,m\}=\frac{m}{2}$ beträgt. Mit wachsendem p konvergiert der Wert von $S(m)$ gegen m, da $\lim_{p\to\infty}\frac{mp}{m+p-1}=\lim_{p\to\infty}\frac{m}{\frac{m}{p}+1-\frac{1}{p}}=\frac{m}{0+1-0}=m$ ist, und somit $m\le \lim_{p\to\infty}S(m)\le m$, also $\lim_{p\to\infty}S(m)=m$ gilt. Dieses Ergebnis hat eine weitreichende Bedeutung: Egal

welchen Schedule man erzeugt, solange er nur arbeitserhaltend ist, ist nach Eagers Theorem automatisch ein guter Speedup garantiert.
Eager geht in seinem Theorem davon aus, daß jede Aktion auf jedem Prozessor ausgeführt werden darf. Vorausgesetzt, daß alle Prozesse sequentiell sind, genügt es allerdings schon zu fordern, daß Prozesse - zwecks der Freigabe ihres Prozessors - verschoben werden dürfen, sobald sie durch das Warten auf einen anderen Prozeß angehalten sind. Unter diesen beiden Forderungen können nicht mehr die zwei oben genannten Ursachen auftreten.
Die Methode, Prozesse zu verschieben, bietet sich bei einem Shared-Memory-Multiprozessorsystem geradezu an, da dort - durch den gemeinsam genutzten Speicher - keine aufwendigen Datentransfers für das Verschieben eines Prozesses zu einem anderen Prozessor nötig sind. Allerdings ist dann zu beachten, daß das Theorem von Eager et al. den Mehraufwand für das Verschieben von Prozessen, sowie für die Aktualisierung der Informationen, welche Prozesse gerade ausführbar und welche Prozessoren gerade frei sind, außer acht läßt, d.h. nicht in Rechnung stellt. Dies bedeutet, daß der Speedup in der Praxis vom theoretisch ermittelten Wert (zum Teil erheblich) abweichen kann.
Auch wenn man die vorgestellte Strategie nicht hundertprozentig in eine Implementierung für parallele, funktionale Sprachen umsetzen kann/möchte, so stellt sie zumindest einen Anhaltspunkt dar. Beispielsweise müssen die Prozesse nicht unbedingt unter allen vorhandenen Prozessoren verschiebbar sein, sondern vielleicht nur zwischen benachbarten Prozessoren. Andererseits könnte man versuchen, Prozesse erst dann zu verschieben, wenn nur noch ein gewisser Anteil aller Prozessoren (beispielsweise die Hälfte) nichts mehr zu tun hat. Der Grund für solch einen Ansatz besteht in der Tatsache, daß sich die Blockierung des Prozessors durch einen angehaltenen Prozeß - je nach aktuellem Zustand des Gesamtsystems - unterschiedlich auf die Laufzeit des Programms auswirkt. Sind von allen m Prozessoren nur noch wenige frei, dann sollte durch Verschieben des angehaltenen Prozesses eher ein Prozessor freigemacht werden, als ihn noch zusätzlich zu den anderen (nutzlos) zu blokkieren. Sind aber viele der m Prozessoren unbenutzt, dann kann man sich den Mehraufwand für das Verschieben des wartenden Prozesses sparen, schließlich sind noch genügend freie Prozessoren vorhanden.

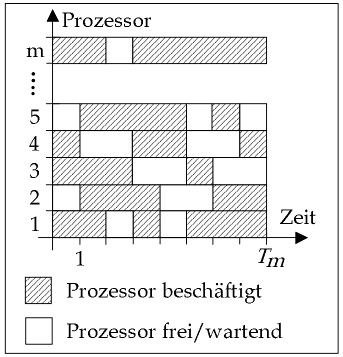
6.1 Theorem von Eager et al.

Zunächst werden zwei Begriffe eingeführt, die in Eagers Theorem bzw. dessen Beweis auftreten:
Subtask: Eine *Subtask* entspricht einer "unabhängigen Einheit sequentieller Arbeit". Dies bedeutet, daß eine Subtask - der bisherigen Bezeichnung folgend - einen sequentiellen Prozeß darstellt. Ein Programm besteht nun aus einer Menge von Subtasks, unter denen gewisse Abhängigkeiten definiert sind.

Durchschnittliche Parallelität: Unter der *durchschnittlichen Parallelität* eines Programms (average parallelism) versteht man den Speedup, der sich mit unbegrenzt vielen Prozessoren erreichen läßt (= $\lim_{m\to\infty} S(m) = \lim_{m\to\infty} T_1 / T_m$).

THEOREM (untere Schranke für den Speedup): Bezeichne p die durchschnittliche Parallelität eines Programms und $S(m)$ den Speedup bei m Prozessoren. Dann gilt für jeden arbeitserhaltenden Schedule: $S(m) \geq \frac{mp}{m+p-1}$.
Diese Schranke ist bestmöglich, d.h. kann auch angenommen werden.

Beweis: Es sei ein beliebiges Programm mit der durchschnittlichen Parallelität p gegeben. T_∞ sei die Laufzeit dieses Programms auf unendlich vielen Prozessoren. Mit der Definition der durchschnittlichen Parallelität ergibt sich für die Laufzeit des Programms auf einem Prozessor $T_1 = T_\infty \cdot p$. $T_\infty \cdot p$ könnte man aber auch als eine Art Maß für die "geleistete Arbeit" auffassen.
Nun wird der Fall betrachtet, daß das Programm auf m Prozessoren mittels eines beliebigen, arbeitserhaltenden Schedules ausgeführt wird:



Da die gleiche "Arbeit verrichtet" wird und der Schedule arbeitserhaltend ist, beträgt die Gesamtzeit, in der die Prozessoren beschäftigt sind, nach wie vor $T_\infty \cdot p$. Die Ausführungsdauer beläuft sich nun aber auf $T_m = \frac{1}{m}(T_\infty \cdot p + I(m))$ (siehe Skizze), wobei $I(m)$ die Gesamtzeit - summiert über alle Prozessoren - bezeichnet, in der die Prozessoren während der Ausführung des Programms untätig sind (total idle time). Für den Speedup bei m Prozessoren ergibt sich dann:

$$S(m) = \frac{T_1}{T_m} = \frac{m \cdot T_\infty \cdot p}{T_\infty \cdot p + I(m)} = \frac{mp}{p + I(m)/T_\infty}.$$

Im folgenden soll $I(m)$ abgeschätzt werden, um die gewünschte Schranke für $S(m)$ zu erhalten. Dazu bezeichne $\omega(t)$ die Menge der Subtasks des Programms, die zum Zeitpunkt t noch nicht vollständig abgearbeitet sind. Zur Menge $\omega(t)$ gehören alle Subtasks, die zum Zeitpunkt t noch gar nicht initialisiert sind, und die, die zwar schon initialisiert und teilweise bearbeitet, aber noch nicht beendet sind. Der Zeitbedarf jeder Subtask aus $\omega(t)$ ist gleich ihrem originalen Zeitbedarf im ursprünglichen Programm, vermindert um den Wert, der der bisherigen Bearbeitung - sofern vorhanden - entspricht. Die Präzedenzrelation unter den Subtasks in $\omega(t)$ ist analog zu der im Programm definiert.
Definiere weiter $L(t)$ als die Länge eines längsten Pfades in $\omega(t)$, wobei sich die Länge eines Pfades aus der Summe der Zeitanforderungen der auf diesem Pfad befindlichen Subtasks errechnet.
$\Rightarrow$ $L(t)$ fällt von T_∞ am Beginn der Ausführung des Programms bis auf Null am Ausführungsende.
Ist $L(t)$ zum Zeitpunkt t nichtfallend, dann existiert eine Subtask am Anfang des längsten Pfades in $\omega(t)$, die gerade nicht ausgeführt wird. Da deren Ausführung durch keine Präzedenzregel verhindert ist (Anfang eines Pfades !), und da der Schedule arbeitserhaltend ist, können zum Zeitpunkt t keine untätigen Prozessoren existieren.
$\Rightarrow$ Prozessoren können nur dann untätig sein, wenn $L(t)$ fällt. Die Gesamtzeit, in der $L(t)$ fällt, ist aber genau T_∞, da die Rate, mit der $L(t)$ fällt, 1 pro Zeiteinheit

beträgt. Da maximal $m-1$ Prozessoren gleichzeitig untätig sein können, ergibt sich für die Gesamtzeit, in der Prozessoren nichts tun, also für $I(m)$:

$I(m) \le T_\infty \cdot (m-1) \quad \Leftrightarrow \quad I(m)/T_\infty \le m-1$.

$\Rightarrow S(m) = \frac{mp}{p+I(m)/T_\infty} \ge \frac{mp}{p+m-1}$. $\Diamond_{\text{Untere Schranke}}$

Beweis für die Annahme der Schranke: Betrachte dazu folgendes Beispiel einer Programmstruktur, für die First-Come-First-Served-Scheduling, also eine arbeitserhaltende Strategie, eingesetzt wird. Das Programm besteht aus einer Subtask, die nach ihrer Beendigung $km+1$ ($k \in \mathbb{N}$ und $km > p$) zusätzliche Subtasks aufruft, wobei jede dieser $km+1$ Subtasks y Zeiteinheiten zur Bearbeitung beansprucht. Die Ausführung der ersten Subtask kostet x Zeiteinheiten. Die Forderung $km > p$ sichert lediglich, daß die Laufzeit x auf alle Fälle positiv ist (vgl. Zeile 4 des folgenden Beweises). Dann gilt:

$T_\infty = x+y \quad \Leftrightarrow \quad x = T_\infty - y$ und

$T_\infty \cdot p = T_1 = (km+1)y + x$

$\Rightarrow T_\infty \cdot p = kmy + y + T_\infty - y$

$\Rightarrow x = \frac{1}{km} T_\infty \cdot (km-p+1)$ und $y = \frac{1}{km} T_\infty \cdot (p-1)$

$\Rightarrow T_1 = T_\infty \cdot p$ und

$$T_m = x + \left\lceil \frac{km+1}{m} \right\rceil \cdot y = \frac{1}{km} T_\infty \cdot (km-p+1) + (k+1)\frac{1}{km} T_\infty \cdot (p-1) = T_\infty + \frac{p}{m} T_\infty - \frac{1}{m} T_\infty$$

$$\Rightarrow S(m) = T_1 / T_m = \frac{T_\infty \cdot p}{T_\infty \cdot (1+p/m-1/m)} = \frac{mp}{m+p-1}$$

$\Diamond_{\text{Theorem}}$

<u>Corollar</u>: Es gelten die Voraussetzungen wie im Theorem. Dann gilt:

$$\frac{1}{2}\min\{m,p\} \underbrace{\le}_{1.} \frac{mp}{m+p-1} \underbrace{\le}_{2.} S(m) \underbrace{\le}_{3.} \min\{m,p\}.$$

<u>Beweis</u>:

1. Ungleichung: o.B.d.A. gelte $m \ge p \Rightarrow$ Zu zeigen: $\frac{p}{2} \le \frac{mp}{m+p-1}$.

Es gilt $m \ge p \quad \Rightarrow \quad m+1 \ge p$

$\Rightarrow \quad pm + p \ge p^2$ (denn $p = \frac{\overbrace{T_1}^{>0}}{\underbrace{T_\infty}_{>0}} > 0$)

$\Rightarrow \quad 2mp \ge pm - p + p^2 = p(m-1+p)$

$\Rightarrow \quad \frac{mp}{m-1+p} \ge \frac{p}{2}$

2. Ungleichung: Aussage des Theorems.

3. Ungleichung: Da man die Laufzeit eines Programms auf einem Prozessor unter dem Einsatz von m Prozessoren höchstens um den Faktor m beschleunigen kann, gilt $S(m) \le m$. Weiter gilt $S(m) \le \lim_{m \to \infty} S(m) = p$, wobei das Gleichheitszeichen aus der Definition der durchschnittlichen Parallelität folgt.

$\Rightarrow S(m) \le m \wedge S(m) \le p \Rightarrow S(m) \le \min\{m,p\}$.

$\Diamond_{\text{Corollar}}$

7. Zusammenfassung und weitere Aspekte

In den Abschnitten 3 bis 5 wurde anhand dreier Beispiele gezeigt, daß man keine automatischen Prozeß-Plazierer und Scheduler für verteilte Programme konstruieren kann, die einen akzeptablen Speedup garantieren. Ein universelles Multiprozessorsystem zur parallelen Verarbeitung, das neben dynamischer Erzeugung von Prozessen auch beliebige Synchronisation unter den Prozessen erlauben muß, muß die kostengünstige Verschiebung von Prozessen unter den Prozessoren ermöglichen, um einen zufriedenstellenden Speedup zusichern zu können.

Darüberhinaus ist bei Eager et al. [4] zu finden, daß in der Praxis einfache sogenannte Load-Sharing-Algorithmen für viele verteilte Anwendungen recht gute Laufzeitergebnisse abwerfen. Unter einem Load-Sharing-Algorithmus kann man sich einen Algorithmus vorstellen, der in verteilten Systemen dafür sorgt, daß die zu "erledigende Arbeit" auf alle Prozessoren möglichst gleichmäßig verteilt wird. Das Ziel ist dabei, die Bearbeitungsdauer so gering wie möglich zu halten. In diesem Zusammenhang heißen Algorithmen *einfach*, wenn sie nur wenige Informationen über den Zustand des Gesamtsystems benötigen, und wenn sie sich leicht implementieren lassen. So zeigen Eager et al. in [4], daß man schon mit einem extrem einfachen Algorithmus, der für die Suche eines geeigneten Prozessors zur Ausführung eines bestimmten Prozesses <u>keine</u> Informationen über den aktuellen Zustand des Systems verwendet, da er einen Zielprozessor rein zufällig auswählt, beachtliche Ergebnisse erzielen kann. Wenn man einen nicht garantierten Speedup in Kauf nehmen möchte/kann, so liefern derartige Load-Sharing-Algorithmen auch für eigene (verteilte) Programme meistens ein befriedigendes Ergebnis. Bei sicherheitsrelevanten Echtzeitprogrammen, für die bestimmte Maximallaufzeiten eingehalten werden müssen, darf man sich natürlich nicht auf dementsprechende Algorithmen verlassen.
Eine weitere Methode, einen hohen Speedup zu erreichen, könnte so aussehen, daß man einfach dem Programmierer die Verantwortung für die Plazierung der Prozesse und das Scheduling überläßt. Allerdings ist dies nur zweckmäßig, wenn die Zahl der Prozesse pro Prozessor fest ist. Dies ist jedoch gerade bei hochgradig dynamischen Programmen sehr unwahrscheinlich, so daß die letztgenannte Strategie keinesfalls für eine universelle, parallele Programmiersprache anwendbar ist. Eine Variante dieser Strategie könnte aber so aussehen, daß die Implementierung der Sprache die Prozesse zufällig den Prozessoren zuordnet, der Programmierer aber Prioritäten für die Prozesse vergibt, um den Scheduler zu steuern.

Der bisherigen Argumentationsweise folgend, könnte man annehmen, daß die beschriebenen Aspekte die einzigen sind, die den Speedup verteilter Programme entscheidend beeinflussen. So würde man annehmen, daß der Speedup auf einem Shared-Memory-System oder auf einem System, das zumindest kostengünstige Zugriffe auf entfernten Speicher erlaubt, einen akzeptablen Wert annehmen wird. Doch selbst auf solchen Systemen kann es zu Problemen kommen; zwar nicht mehr im "Kampf" um Prozessoren, sondern beispielsweise um Speicherzellen. Betrachtet man die Aktionen anstatt als Anforderung von Rechenkapazität als Zugriff auf Speicherzellen, dann entspricht die Reihenfolge, in der die Aktionen auf den Speicher zugreifen, der, in der sie ausgeführt werden. Und damit entstehen wieder die gleichen

Probleme, die zuvor diskutiert wurden, und die durch den Einsatz eines Shared-Memory-Systems zunächst beseitigt wurden. Genaugenommen entstehen diese Probleme bei jeder Art von Ressource, wenn jede Aktion auf eine bestimmte Instanz dieser Ressource in irgendeiner Form angewiesen ist.
Neben den Konsequenzen für die Laufzeit bei parallelen, deterministischen Programmen, kann das Scheduling auch die während der Ausführung des Programms benötigte Größe an Speicherplatz entscheidend beeinflussen. Verläßt man darüberhinaus den Bereich der reinen funktionalen Sprachen und erlaubt auch nichtdeterministische Programme, so können selbst geringfügige Unterschiede im Schedule, das Verhalten eines Programms bezüglich Laufzeit und Speicherplatz radikal verändern. Dies soll an dieser Stelle allerdings nicht weiter vertieft werden.

8. Anhang

A) Theorem von Jaffe

Für das Verständnis des Theorems und vor allem des Beweises müssen zunächst eine Reihe neuer Begriffe eingeführt werden:
k-Typ Task-System: Ein *k-Typ Task-System* wird durch ein Viertupel $(\Gamma, <, \mu, v)$ modelliert, in dem Γ die Taskmenge, $<$ die Präzedenzrelation unter den Tasks, $\mu : \Gamma \to \mathbb{R}$ eine Zeitfunktion, die jeder Task ihren Zeitbedarf zuordnet, und $v : \Gamma \to \{1, \ldots, k\}$ eine Typfunktion bezeichnet, die jeder Task einen Typ aus $\{1, \ldots, k\}$ zuordnet. Ist $v(T) = i$, dann muß die Task T auf einem Prozessor vom Typ i ausgeführt werden. Wenn im weiteren die Variable i benutzt wird, so gilt stets $1 \leq i \leq k$ als vorausgesetzt.
Ein *Schedule* für $(\Gamma, <, \mu, v)$ ist eine totale Funktion $s : \Gamma \to \mathbb{R}$. $s(T)$ bezeichnet dabei die Startzeit, $s(T) + \mu(T)$ die Endzeit der Task T. $T \in \Gamma$ heißt zum Zeitpunkt t ausgeführt, wenn $s(T) \leq t < s(T) + \mu(T)$ gilt. Ein *gültiger Schedule* für $(\Gamma, <, \mu, v)$ auf einer Menge gleichschneller Prozessoren $P = \{P_{ij} \mid 1 \leq i \leq k, 1 \leq j \leq m_i\}$ ist ein Schedule für $(\Gamma, <, \mu, v)$, für den die nächsten beiden Punkte erfüllt sind:

(1) $\forall i = 1, \ldots, k : \forall t \in \mathbb{R}$: die Anzahl an Typ-$i$ Tasks, die zur Zeit t ausgeführt werden, ist $\leq m_i$.
Damit wird garantiert, daß zu jeder Zeit nur soviele Tasks von jedem Typ bearbeitet werden, wie entsprechende Prozessoren zur Verfügung stehen.

(2) Für $T, T' \in \Gamma$: Wenn $T < T' \Rightarrow s(T) + \mu(T) \leq s(T')$. Die Startzeit von T' ist mindestens so groß wie die Endzeit von T.
Diese Forderung sichert die Einhaltung der Präzedenzrelation $<$, indem nur Tasks ausgeführt werden dürfen, deren Vorgänger bereits abgearbeitet sind.

Die *Endzeit* eines gültigen Schedules ist die größte Endzeit aller Tasks aus Γ. Ein gültiger Schedule mit minimaler Endzeit heißt *optimaler Schedule*.

Kette: Unter einer *Kette C* versteht man eine Folge von Tasks $C = (T_1, \ldots, T_n)$ mit $T_j \in \Gamma$, so daß für alle j aus $1 \leq j < n$ $T_j < T_{j+1}$ gilt. Die *Länge* von C ist definiert als $\Sigma_{j=1}^{n} \mu(T_j)$ und entspricht damit der Ausführungsdauer der Kette. Die (*statische*) *Höhe* einer Task $T \in \Gamma$ ist gleich der Länge einer längsten Kette, die bei T beginnt. Die

(statische) Höhe von $(\Gamma, <, \mu, v)$ ist gleich der Länge einer längsten Kette mit einer beliebigen Task $T \in \Gamma$ als Startpunkt.

Dynamische Höhen: Bezeichne s einen Schedule für $(\Gamma, <, \mu, v)$ und t eine Zeit unter der Endzeit von s. Die *Höhe* einer Task T zum Zeitpunkt t ist als die Länge der längsten bei T beginnenden Kette definiert, wobei zur Länge nur die bisher nicht bedienten Zeitanforderungen beitragen. Analog ist die Höhe von $(\Gamma, <, \mu, v)$ zum Zeitpunkt t als die Länge einer längsten Kette definiert, die bei einer beliebigen Task $T \in \Gamma$ beginnt, mit der Einschränkung, daß T noch nicht vollständig abgearbeitet sein darf.
Zur Abkürzung bezeichnet μ_i die gesamte Zeitanforderung aller Tasks vom Typ i und h die (statische) Höhe von $(\Gamma, <, \mu, v)$.

<u>THEOREM</u>: Sei $(\Gamma, <, \mu, v)$ ein k-Typ Task-System, für das ein Schedule auf einer Menge gleichschneller Prozessoren generiert werden soll. ω_{opt} sei die Endzeit eines optimalen Schedules, ω die eines beliebigen, arbeitserhaltenden Schedules. Dann ist dieser höchstens $k + 1 - 1/\max\{m_1, \ldots, m_k\}$ -mal schlechter als der optimale Schedule für $(\Gamma, <, \mu, v)$, d.h.

$$\omega \leq (k + 1 - 1/\max\{m_1, \ldots, m_k\}) \cdot \omega_{opt} .$$

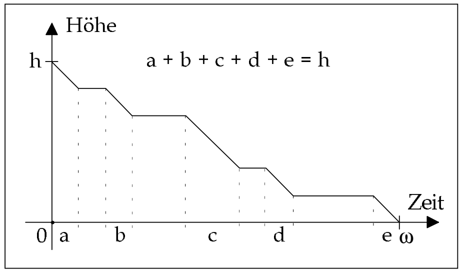
<u>Lemma 1</u>: Sei $(\Gamma, <, \mu, v)$ wie oben. ω_{opt} sei die Endzeit eines optimalen Schedules für $(\Gamma, <, \mu, v)$. Dann gilt: $\omega_{opt} \geq \max\{h, \mu_1/m_1, \ldots, \mu_k/m_k\}$.

<u>Beweis</u>: Pro Zeiteinheit können höchstens m_i Zeitanforderungen für Tasks vom Typ i erfüllt werden (für jedes i), da nur m_i Typ-i Prozessoren vorhanden sind.
$\Rightarrow$ Mindestens $\lceil \mu_i/m_i \rceil$ Zeiteinheiten sind nötig, um alle Typ-i Zeitanforderungen zu erfüllen.
$\Rightarrow \omega_{opt} \geq \max\{\lceil \mu_1/m_1 \rceil, \ldots, \lceil \mu_k/m_k \rceil\} \geq \max\{\mu_1/m_1, \ldots, \mu_k/m_k\}$.
Weil h die Länge der längsten Kette in $(\Gamma, <, \mu, v)$ ist, die demzufolge sequentiell abgearbeitet werden muß, gilt offensichtlich $\omega_{opt} \geq h$.
$\Rightarrow \omega_{opt} \geq \max\{h, \mu_1/m_1, \ldots, \mu_k/m_k\}$. $\Diamond_{\text{Lemma 1}}$

<u>Lemma 2</u>: Sei $(\Gamma, <, \mu, v)$ wie oben. ω bezeichne die Endzeit eines beliebigen, arbeitserhaltenden Schedules für $(\Gamma, <, \mu, v)$. Dann gilt:

$$\omega \leq \mu_1/m_1 + \cdots + \mu_k/m_k + h \cdot (1 - 1/\max\{m_1, \ldots, m_k\}).$$

<u>Beweis</u>: Sei s der gegebene, arbeitserhaltende Schedule mit der Endzeit ω.
Man trage nun die Höhe gegen die Zeit auf und unterteile das Intervall $[0, \omega]$ auf der Zeitachse in Abschnitte, in denen die Höhe fällt, und in Abschnitte, in denen sie konstant bleibt (siehe Skizze rechts). In den fallenden Abschnitten fällt die Höhe um den Faktor 1 pro Zeiteinheit, da die Höhe der höchsten Task um eben diesen Faktor reduziert wird.



$\Rightarrow$ Die Summe der Längen aller fallenden Intervalle beträgt h.

$\Rightarrow$ Es bleibt zu zeigen, daß die Summe der Längen aller konstanten Intervalle
$\leq \mu_1/m_1 + \cdots + \mu_k/m_k - h/\max\{m_1, \ldots, m_k\}$ ist.

<u>Behauptung 1</u>: Zu jedem Zeitpunkt existiert in einem konstanten Intervall ein i, so daß alle m_i Prozessoren vom Typ i beschäftigt sind.

<u>Beweis</u> (durch Widerspruch): Sei t ein Zeitpunkt innerhalb eines konstanten Intervalls, so daß für alle i weniger als m_i Prozessoren beschäftigt sind. Da es für jedes i einen untätigen Prozessor gibt, und es sich um einen arbeitserhaltenden Schedule handelt, müssen alle ausführbaren Tasks gerade bearbeitet werden. Insbesondere wird dann aber auch die Task maximaler Höhe in diesem Augenblick bearbeitet.
$\Rightarrow$ Die Höhe wird reduziert. Dies ist aber ein Widerspruch dazu, daß t aus einem konstanten Intervall stammt. $\Diamond_{\text{Behauptung 1}}$

Es bezeichne h_i die Zeitanforderungen aller Tasks vom Typ i während den fallenden Intervallen. Da während eines fallenden Intervalls der Länge n die Höhe von $(\Gamma, <, \mu, v)$ um n reduziert wird, werden somit mindestens n Zeitanforderungseinheiten von Tasks bedient. Da die Summe der Längen aller fallenden Intervalle h beträgt, werden also insgesamt mindestens h Zeitanforderungen erfüllt; es gilt $\Sigma_{j=1}^{k} h_j \geq h$ [∗].
Die Gesamtlänge der konstanten Intervalle, in denen m_i Tasks vom Typ i (gleichzeitig) ausgeführt werden, beträgt offenbar höchstens $\lfloor(\mu_i - h_i)/m_i\rfloor$ ($\mu_i - h_i =$ Zeitbedarf aller Tasks vom Typ i in konstanten Intervallen). Die Behauptung 1 besagt nun, daß es zu jedem Zeitpunkt in einem konstanten Intervall ein i gibt, so daß alle m_i Prozessoren beschäftigt sind. Alle m_i Prozessoren können in konstanten Intervallen (für ein bestimmtes i) aber höchstens $\lfloor(\mu_i - h_i)/m_i\rfloor$ Zeiteinheiten aktiv sein. Alle m_i Prozessoren können für jedes i also in konstanten Intervallen höchstens $\lfloor(\mu_1 - h_1)/m_1\rfloor + \cdots + \lfloor(\mu_k - h_k)/m_k\rfloor =_{def} X$ Zeiteinheiten aktiv sein. Wegen Behauptung 1 kann somit die Gesamtlänge aller konstanten Intervalle höchstens X betragen.

$\Rightarrow$ Gesamtlänge aller konstanten Intervalle $\leq X \leq (\mu_1 - h_1)/m_1 + \cdots + (\mu_k - h_k)/m_k$

$$
\begin{aligned}
&= \mu_1/m_1 + \cdots + \mu_k/m_k - h_1/m_1 - \cdots - h_k/m_k \\
&\leq \mu_1/m_1 + \cdots + \mu_k/m_k - (h_1/\max\{m_1, \ldots, m_k\} + \cdots + h_k/\max\{m_1, \ldots, m_k\}) \\
&= \mu_1/m_1 + \cdots + \mu_k/m_k - (h_1 + \cdots + h_k)/\max\{m_1, \ldots, m_k\} \\
&\leq \mu_1/m_1 + \cdots + \mu_k/m_k - h/\max\{m_1, \ldots, m_k\} \qquad \text{(wegen [∗])}
\end{aligned}
$$

$\Rightarrow \omega \leq \mu_1/m_1 + \cdots + \mu_k/m_k + h - h/\max\{m_1, \ldots, m_k\}$ $\Diamond_{\text{Lemma 2}}$

<u>Beweis</u> des Theorems: Gegeben sei ein k-Typ Task-System $(\Gamma, <, \mu, v)$. Sei $z =_{def} \max\{h, \mu_1/m_1, \ldots, \mu_k/m_k\}$. Nach Lemma 1 gilt $\omega_{opt} \geq z$, und nach Lemma 2 gilt:

$$
\begin{aligned}
\omega &\leq \mu_1/m_1 + \cdots + \mu_k/m_k + h(1 - 1/\max\{m_1, \ldots, m_k\}) \\
&\leq k \cdot \max\{h, \mu_1/m_1, \ldots, \mu_k/m_k\} \\
&\quad + \max\{h, \mu_1/m_1, \ldots, \mu_k/m_k\} \cdot (1 - 1/\max\{m_1, \ldots, m_k\}) \\
&= \max\{h, \mu_1/m_1, \ldots, \mu_k/m_k\} \cdot (k + 1 - 1/\max\{m_1, \ldots, m_k\}) \\
&= z \cdot (k + 1 - 1/\max\{m_1, \ldots, m_k\})
\end{aligned}
$$

$$\Rightarrow \frac{\omega}{\omega_{opt}} \leq \frac{z \cdot (k+1-1/\max\{m_1, \ldots, m_k\})}{z} = k + 1 - 1/\max\{m_1, \ldots, m_k\}$$

$\Diamond_{\text{Theorem}}$

B) Zur Erinnerung

Partielle Ordnung:
$R \subseteq A \times A$ sei eine zweistellige Relation. R heißt *partielle Ordnung*
:$\Leftrightarrow$ R ist reflexiv ($\forall a \in A : (a, a) \in R$),
antisymmetrisch ($\forall a, b \in A : (a, b) \in R \wedge (b, a) \in R \Rightarrow a = b$) und
transitiv($\forall a, b, c \in A : (a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$).

Transitive Hülle:
Es sei $G = (V, E)$ ein gerichteter Graph.
$H(G) = (V, E')$ heißt *transitive Hülle* von G
:$\Leftrightarrow E' = \{(v, w) \mid v, w \in V$ und es existiert eine Kantenfolge von v nach $w\}$.

9. Literatur

[1] Burton, F. W. und Rayward-Smith, V. J. (1994) Worst case scheduling for parallel functional programs. *J. Functional Programming*, 4 (1): 65-75, Januar. (Originaltext)

[2] Jaffe, J. M. (1980) Bounds on the scheduling of typed task systems. *SIAM J. Comput.*, 9: 541-551.

[3] Eager, D. L., Zahorjan, J. und Lazowska, E. D. (1989) Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38 (3): 408-423, März.

[4] Eager, D. L., Lazowska, E. D. und Zahorjan, J. (1986) Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.*, 12 (5): 662-675, Mai.