Parallele Multiprozessorsysteme
Das Ende der Hardware Miniaturisierung?

Ferhat Beyaz, betreut von Freddy Lopez Villafuerte
Institut für Informatik - Freie Universität Berlin, Takustrasse 9, 14195 Berlin, Deutschland

I. Einleitung

Diese Facharbeit wurde im Rahmen einer Proseminararbeit für Technische Informatik an der *Freien Universität Berlin* unter der Leitung von *Georg Wittenburg* geschrieben.

Die Entwicklung von Hardware für Computer und andere elektronische Geräte ist erstaunlich mit der Mikroelektronik und dem Erstellen neuer Techniken gewachsen um so in sehr kleinem Raum Millionen von Transistoren und andere elektronische Geräte zu installieren. Ziel dieser Arbeit ist es, die Grenzen der fortlaufenden Miniaturisierung auf elektronischer Hardware (Nanotechnologie) zu zeigen, welche Grund für neue Entwicklungen und die Einführung der parallelen Multiprozessoren ist.

Die Arbeit teilt sich also grob in drei Teile. Im ersten Teil, welcher Kapitel II und III beinhaltet, lernen wir *Gordon Moore* und seine bekannte These, welche unter dem Namen *Mooresche Gesetz* bekannt wurde. Wir werden feststellen, dass die fortlaufende Computerminiaturisierung immer mehr an Bedeutung gewonnen hat und dass die Entwicklungen unberechenbar steil waren. Der zweite Teil der Arbeit, bestehend aus Kapitel IV, wird sich mit den Grenzen der fortschreitenden Nanotechnologie befassen. Hierbei wird vor allem auf die physikalischen Grenzen eingegangen. Wir werden erkennen, dass die Entwicklungen der Nanotechnologie der letzten Jahre und Jahrzehnte nicht unbegrenzt weitergeführt werden können. Nachdem dies geklärt ist, kommen wir zum letzten Teil, welcher aus Kapitel V besteht. Hierbei stellen wir parallele Multiprozessoren in ihrem Aufbau und genauer Funktionsanalyse vor. Außerdem soll auf viele Problematiken eingegangen werden, die sich durch das parallele Abarbeiten von Prozessen ergeben, wie beispielsweise dem korrekten Scheduling.

Alle Links in den Referenzen am Ende der Arbeit wurden am 25. Dezember 2008 um 14.00 Uhr auf Aktualität und Vollständigkeit geprüft.

II. Hält sich das Mooresche Gesetz noch heute?

2.1) Eine kurze Definition

Im Jahr 1965 stellte *Gordon Moore*, einer der Gründer von *Intel*, das nach ihm benannte *Mooresche Gesetz* fest. Es besagt, dass sich die Anzahl der Transistoren auf einem Computerchip alle 18 Monate verdoppelt, was gleichzeitig bedeutet, dass diese Chips kleiner werden [1]. Diese Regel, die er aufgrund der rasanten Entwicklung der Halbleiterindustrie traf, hat er 1975 relativiert, sodass er die Verdoppelung der Transistoren eines Chips auf etwa alle zwei Jahre voraussagte [2]. Eine weitere weitreichendende Folge ist, dass dieser Platz für weitere Transistoren genutzt werden kann, weswegen sich das Reduzieren der Transistorzahl auf einem Chip doppelt auswirkt [3]. Es entsteht also Leistung durch technologischen Fortschritt, aber auch durch das Sparen von Platz auf Chips und die damit verbundene Zunahme der Transistoren pro cm^2. Wichtig ist an dieser Stelle, dass das *Mooresche Gesetz* nur empirisch ist und somit streng wissenschaftlich gesehen keinerlei Bedeutung hat. Trotzdem sind viele Entwickler heutzutage davon überzeugt, dass dieser Theorie zu folgen sei oder gar, dass sie längst bewiesen ist („[…] ist Moores These längst bewiesen […]“) [4].

2.2) Das Ende des Mooreschen Gesetzes?

Als *Gordon Moore* 2003 an der *International Solid-State Cicuits Conference* (*ISSCC*) einen Vortrag über die Zukunft der Halbleiterbranche hielt [5], konnte man sowohl Zuversicht als auch Warnungen aus seinen Worten leiten. Er ging davon aus, dass das nach ihm benannte Gesetz, das so genannte *Mooresche Gesetz*, in wenigen Jahren seine Gesetzmäßigkeit verliere. Auch er könne nicht genau voraussagen, wie es dann weiterginge. Worin er sich aber sicher war, war dass das Ende seines Gesetzes auch gleichzeitig der Beginn einer neuen Zeitrechnung bedeuten würde. Hierbei sei auf Abbildung 1 verwiesen (frei übersetzt): „Wir stehen wahrscheinlich vor einer neuen Zeit. Es gibt sicherlich kein Ende der Kreativität“.

Fig. 1. “Wir stehen wahrscheinlich vor einer neuen Zeit. Es gibt sicherlich kein Ende der Kreativität.“

Wie es *Moore* vorausgesehen hat, sind in den letzten Jahren und vor allem jüngst starke Entwicklungen im Bereich der Nanotechnologie zu verzeichnen. Die Entwicklungen gingen soweit, dass *Moore* seine These sogar nachträglich modifizieren musste. Mittlerweile ist obiges Gesetz aber mehr als nur eine bloß empirisch festgestellte Theorie. Mittlerweile richten große Prozessorkonzerne ihre Forschung für die nächsten Jahre nach dem *mooreschen Gesetz*: „Wir arbeiten hart daran, dass das Mooresche Gesetz unsere Industrie auch in Zukunft weiter antreibt. In unseren Forschungs-Labors haben wir bereits Pläne für die Entwicklungen der nächsten 10 bis 15 Jahre festgelegt“, sagt *Craig Barrett*, *CEO* der *Intel Corporation* [6].

2.3) Einige Vergleiche

Moore schätzte, dass die Zahl der Transistoren auf einem Chip im Jahre 2003 bei 10^8 liegt. Das sind 100mal mehr, als es Ameisen auf der Erde gibt. Bereits in 2003 hat die Industrie jedoch wie erwähnt 1000000000000000000 (10^8) Transistoren hergestellt. Um die Analogie aufrecht zu erhalten, müsste jede Ameise daher ein Paket mit 100 Transistoren auf ihrem Rücken tragen. Nach Angaben der *U.S. Semiconductor Industry Association* produzierte die Halbleiter-Industrie im Jahr 2004 mehr Transistoren als Reiskörner in der Welt geerntet wurden [7].

Intel selbst schätzt auf ihrer Webseite, dass ein einzelner Transistor auf einem ihrer Chips umgerechnet etwa soviel kostet wie ein schwarzgedrucktes Zeichen in einer gewöhnlichen Tageszeitung. Ein guter Vergleich: Ein Flug von New York nach Paris kostet etwa umgerechnet 700€ und dauert 7 Stunden. Würden das beschriebene *Mooresche Gesetz* auch in der Flugzeugbranche gelten, so würde der Flug etwa einen Cent kosten und der Passagier wäre in weniger als einer Sekunde in Paris [8].

III. Die Entwicklung der Mikroprozessoren

3.1) Die 60er Jahre – Der Anfang des Computerzeitalters

Nachdem in den 60er Jahren die ersten integrierten Schaltungen von *Fairchild Semiconductor* und parallel von *Texas Instruments* entwickelt wurden, kamen nacheinander TTL (*transistor-transistor logic*) und CMOS (*complementary metal oxide semiconductor*) Schaltungen auf den Markt, was dazu führte, dass nun kein Zweifel über den Erfolg von Computertechnologien mehr bestand. Im Zuge dieser Entwicklungen formulierte *Moore* das weiter oben beschriebene Gesetz [9].

3.2) Die 70er Jahre – Computer warden überall eingesetzt

Zu Beginn der 70er Jahre waren Multiprozessoren noch nicht eingeführt, doch die rasche Verbreitung von Elektronikartikeln und der beginnende Preiskrieg kurbelten das Geschäft an. Zu dieser Zeit haben bereits drei führende Gruppierungen den Computer auf einen Chip gepackt, *The Central Air Data Computer* (*CADC*), der *Intel* 4004, und der *Texas Instruments TMS* 1000, wobei *CADC* in der US amerikanischen Luftwaffe und der *TMS* 1000 in den Taschenrechnern eingesetzt wurde (allerdings nicht standalone).

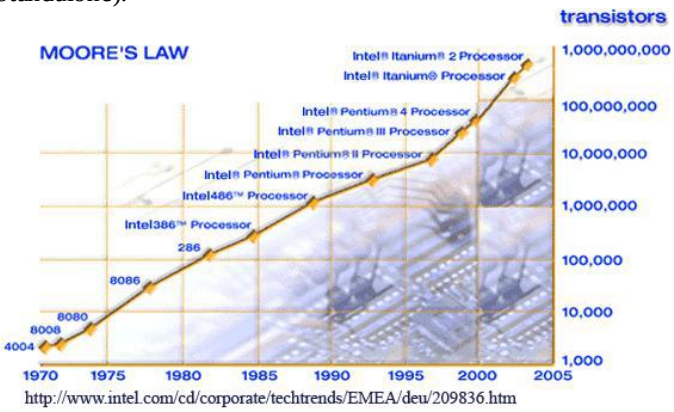

Fig. 2. Auf der Ordinate wird die Transistorzahl auf den jeweiligen Prozessoren zu der Zeit der Entwicklung dieser Systeme, welche auf der Abszisse eingetragen ist, ins Verhältnis gesetzt.

Zu bemerken ist, dass die Ordinate Werte zwischen Eintausend und einer Milliarde, die Abszisse dagegen nur Werte zwischen 1970 und 2005, was 35 Jahren entsprechen, einträgt. Man erkennt also, dass es ein rasanter Anstieg in der Anzahl der Transistoren gegeben haben muss.

Man merkt also, dass die Computerchips schon zu dieser Zeit praktischen Bezug haben und vielseitig eingesetzt werden konnten. Der *Intel* 4004 Prozessor, getaktet mit 108 kHz und im Besitz von 2300 Transistoren, wird als *erster Computer auf einem einzigen Chip* gefeiert. *Intels* 4004 wird später vom 8080 ersetzt. Dessen 8-Bit-Prozessor enthält 4700 Transistorfunktionen und taktet mit 2 MHz [10].

Im Jahre 1978 beginnt mit dem 8086 und seinem 16-Bit-Prozessor die Erfolgsgeschichte von *Intels* 80*x*86 Reihe. Alle nun erschienenen Produkte dieser Familie waren abwärtskompatibel, was bedeutet das Software, die zuvor auf dem 8086 lief, auf jedem später verkauften *Intel Prozessor* der *x*86 Reihe lief. Der 8086 Prozessor hatte nun bereits 29 000 Transistoren und eine Taktfrequenz von 5 (später 10) MHz. Er konnte mindestens 330 000 Befehle pro Sekunde abarbeiten.

3.3) Die 80er Jahre – 32bit CPUs setzen sich durch

Mit dem 80286 stellte *Intel* im Jahr 1982 eine 16-Bit CPU mit fast 130 000 Transistoren vor, bevor dann 1985 endgültig das Zeitalter der 32-Bit Prozessoren mit dem 80386 von Intel anbricht. Die Transistorzahl steigt weiter auf mehrere 100 000 bis über eine Million Transistoren (*Intel* 80486).

3.4) Das Zeitalter "Intel"

Da *Intel* zu spüren bekommen hatte, was es bedeuten kann, wenn die Konkurrenz den gleichen Namen verwenden darf wie das Original, nannte sie ihren neuen Prozessor 1993 nicht 80586, sondern *Pentium* (Pente (πέντε), griech. „fünf"). Dieser besaß über 3,1 Millionen Transistoren und hatte intern anfangs eine Taktrate von 60 bzw. 66 MHz. Auch in den folgenden Jahren hat sich die Leistung stets verbessert. Die sechste *Intel* Generation gab es mit 5,5 Millionen Transistoren und einer Taktrate von bis zu 200MHz. Im Jahr 1997 kommt der Intel Pentium 2 Prozessor raus, der mit 7,7 Millionen Transistoren zur Referenz wurde und 2001 vom Pentium 3 mit Transistorzahlen zwischen 9,5 (*Katmai- Chip*) und 44 Millionen (*Tualatin- Chip*) [11]. Der anschließend erschienene Pentium 4 ist mit noch mehr Transistoren in einem einzigen Kern die Krönung an jahrzehntelanger Hardwareminiaturisierung.

IV. Nanotechnologie und die Physikalischen Grenzen

4.1) Wo liegen die Grenzen?

Wir haben nun festgestellt, dass sich gemäß dem *Mooreschen Gesetz* alle zwei Jahre die Leistung der CPUs verdoppelt. Führt man die Kurve weiter, so wären die MOSFET-Transistoren im Jahr 2010 geschätzt bei 35nm und würden den Nanometer im Jahr 2040 erreichen, was bedeutet, dass obiges empirische Gesetz spätestens auf atomarer, aber

sehr wahrscheinlich schon vorher ein Ende haben wird. Durch die konstante Miniaturisierung ist wesentlich mehr Platz auf den einzelnen Komponenten. Wir wissen von weiter oben, dass der Preis für einen einzelnen Transistor im Zuge der beschriebenen Entwicklungen stark runtergegangen ist. Allerdings sind die Kosten für einen integrierten Schaltkreis seit vielen Jahren konstant bei 100€ pro cm^2 [12].

Da der Preis nun konstant bleibt, ist die geeignete Methode um verschiedene Technologien zu vergleichen die Anzahl an Operationen pro Sekunde pro cm^2. Um nun die Rechenleistung zu steigern, gibt es im Prinzip zwei Wege. Zum einen kann man die Operationen pro Sekunde erhöhen oder man erhöht die Dichte der Komponenten. Wichtig ist aber nun, dass grenzenlose Erhöhung der Dichte sowie der Operationen pro Sekunde nicht möglich ist.

Im Folgenden möchten wir folgendes Diagramm diskutieren und daraus wichtige Schlussfolgerungen ziehen.

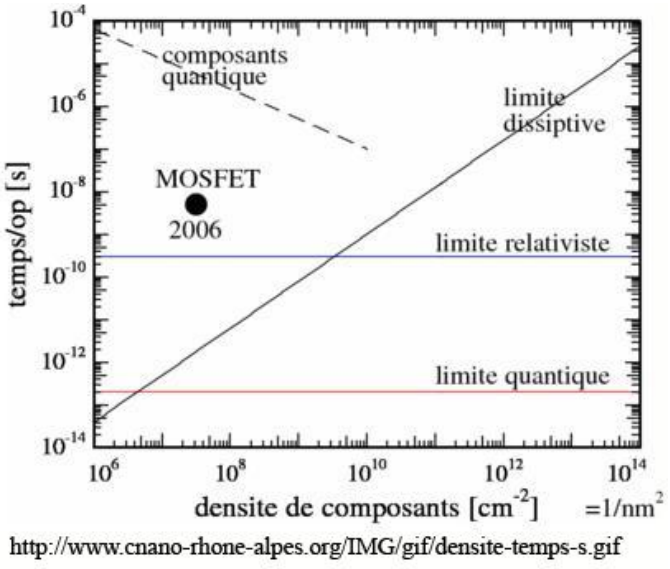

Fig. 3. Auf der Ordinate ist die Zeit pro Operation eingezeichnet, wohingegen die Abszisse die Dichte der Komponenten pro $cm^2 = \left(1/_{nm^2}\right)$ bezeichnet.

Der schwarze Punkt stellt den Stand der MOSFET Transistoren auf einem Prozessor aus dem Jahr 2006 dar. Er soll symbolisieren, wie die sich die neuen Entwicklungen im nanotechnologischen Bereich im Verhältnis zu den noch zu besprechenden Grenzen verhält.

4.2) Thermische Grenzen

Bei steigender Anzahl an Rechenoperationen steigt natürlich auch die erzeugte Wärme, die dann eventuell nicht mehr schnell genug abtransportiert werden kann. Steht der Prozessor in einer Umgebung mit Raumtemperatur, so entstehen thermischen Fluktuationen, also das Hin- und Herschwanken der Wärme, wobei die dabei entstehende Energie $5 \cdot 10^{-21}$ Joule beträgt (wir betrachten weiterhin einen Bereich von cm^2).

Damit der gegebene Schaltkreis nun aber quasi immun gegen derartige Fluktuationen ist, sprich, damit er keine Fehler produziert, muss jede Operation 10mal mehr Energie abgeben, was bedeutet, dass $5 \cdot 10^{-20}$ Joule abgegeben werden müssten. Die abgegebene Leistung eines integrierten Schaltkreises (pro cm^2) ist die Beziehung:

$$\frac{Komponentendichte}{Zeit/Operationen}.$$

In der Praxis liegt der Wert deutlich höher, dies ist im Grunde nur ein Minimum. Die abtransportierbare thermische Energie wird von der globalen thermischen Leitfähigkeit stark begrenzt.

Dieses Limit in Abbildung 3 wird als *limite dissiptive* bezeichnet. Unter *dissipativen Strukturen* versteht man genauer offene Systeme, die Energie mit der Umgebung austauschen. Der Bereich unter der Geraden ist nicht mehr zugänglich, da dies bedeuten würde, dass zu wenig thermische Energie abgegeben werden würde. Außerdem kann ein Schaltkreis nur bei einigen zehn Kiloherz betrieben werden, damit das sichere Abtransportieren der Energie gesichert ist. Vergleicht man zwei Technologien, so ist diejenige die bessere, die bei gegebenem Preis gerade eine bestmögliche Annäherung an die *limite dissiptive* schafft [13].

4.3) Die Heisenbergsche Unschärferelation

Nachdem wir nun obiges thermisches Problem genauer besprochen haben, richten wir den Blick auf eine weitere physikalische Grenzmäßigkeit. Die so genannte *Heisenbergsche Unschärfe- oder Unbestimmtheitsrelation* besagt, dass Ort und Impuls eines Teilchens niemals zugleich genau gemessen werden können. Je genauer man den Ort eines Teilchens misst, desto ungenauer wird die Messung des Impulses und umgekehrt.

Die gemeinsame Unschärfe beider Größen ist stets in der Größenordnung des *Planckschen Wirkungsquantums*. Sie ist damit sehr klein und ist in der klassischen Physik nicht erkennbar. Das *Plancksche Wirkungsquantum* ist eine universelle Naturkonstante, die gerade für ein beliebiges Teilchen das Produkt seiner Geschwindigkeit, seiner Masse sowie der Wellenlänge ist. Das ganze hört sich jetzt weit hergeholt an, allerdings ist es für folgende Betrachtung wertvoll. Nach obigem (Heisenbergschen) Prinzip der Unschärfe ist die Dauer einer Quantenfluktuation der Energie E dargestellt durch h/E, wobei h das *Plancksche Wirkungsquantum* ist [14].

Quantenfluktuation kommt daher, dass im mikroskopischen nicht mehr die klassischen Gesetze der Physik gelten. So kann es beispielsweise passieren, dass ein Elektron im Chip einen Weg nimmt, den es eigentlich gar nicht nehmen dürfte. Es „tunnelt“ beispielsweise durch einen Isolator.

Dies kann dazu führen, dass der Prozessor einen Fehler in seiner Rechnung hat. Wenn die Kommutationsenergie eines Bits größer als $5 \cdot 10^{-20}$ Joule ist, so ist die Zeit der Quantentransistion 10^{-14} Sekunden.

Kommutationsenergie ist diejenige Energie, die benötigt wird um ein Bit vom Zustand 0 in den Zustand 1 zu bringen oder umgekehrt. Damit jegliche Quantenfehler ausgeschlossen werden können, muss die oben beschriebene Kommutationszeit länger sein. Geeignet wären eher etwa $5 \cdot 10^{-14}$ Sekunden.

Diese Zeit wird durch die horizontale Linie *limite quantique* im unteren Teil der Abbildung dargestellt. Sie hat ihren Schnittpunkt mit *limite dissiptive* bei einer Dichte von

etwa 10^7, welche in aktuellen Schaltkreisen auch erreicht wird, was gleichzeitig aber auch bedeutend, dass wir an diesem Punkt bereits an eine wesentliche technologische Grenze gestoßen sind. Damit sind die thermischen Effekte bei Raumtemperatur wesentlich stärker limitiert als die eben beschriebenen Quanteneffekte [15].

4.4) Relativistisches Limit der Informationsverbreitung

Wir wissen, dass keine Information mit einer Geschwindigkeit übertragen werden kann, die sogar die Lichtgeschwindigkeit schlägt. Damit gibt es schon einmal eins grobes Limit für jeglichen Informationsaustausch.

Für einen integrierten Schaltkreis der Größe 1 cm ist eine minimale Zeit von $0{,}3/\sqrt{\varepsilon}$ Nanosekunden erforderlich, um die Information über den Prozessor zu verteilen. ε ist dabei die so genannte Permittivitätszahl, also die dielektrische Leitfähigkeit oder Permittivität, die gerade die Durchlässigkeit eines Materials für elektrische Felder angibt.

Diese neue Grenze ist ein wenig eingrenzender und somit nützlicher für unsere Betrachtung. Mit geschicktem Design der einzelnen Schaltkreise und der bezüglich der Informationsübertragung effizienten Anordnung der einzelnen Komponenten ist eine lokale Informationsübertragung möglich. Wir stellen also fest, dass es zwar ein relativistisches Limit der Informationsausbreitung existiert, die Übertragungsdauer aber durch geschicktes Design gering gehalten werden kann, was aber nicht unendlich geht, da auch geschickte Konstruktion eines Schaltkreises Grenzen hat. Diese Begrenzung ist in obiger Abbildung als *limite relativiste* dargestellt [16].

4.5) Die RC Verzögerung

Betrachten wir nun eine geladene Komponente eines integrierten Schaltkreises. Es ist logisch, dass eine minimale Zeit notwendig ist, um diese Komponente durch einen anderen Schaltkreis zu „entladen“, der aber nur eine begrenzte Leitfähigkeit hat, die wir im Folgenden mit G bezeichnen. Betrachten wir nun die Kapazität C der Komponente, so geschieht das oben beschriebene „Entladen“ in der Zeit $\mathrm{T} = \frac{\mathrm{C}}{\mathrm{G}}$.

G ist allerdings noch durch die Ausgangsimpedanz, also dem komplexen Wechselstromwiderstand Z, und dem Widerstand des zugeführten Leiters begrenzt. Nun betrachten wir die Auswirkungen des letzteren Punktes auf G. Sei die Länge L und Breite w eines Leiters kleiner als die Fermiwellenlänge λ_F der Elektronen. Durch dieses Limit ist folgende Leitfähigkeit gegeben:

$$G = \left(\frac{2 \cdot e^2}{h}\right) \cdot K,$$

wobei K den Wert

$$K = 2 \cdot \left(\frac{w \cdot l_e}{\lambda_F \cdot L}\right)$$

hat und l_e die freie elastische Weglänge eines Elektrons ist, wobei diese Weglänge gemittelt ist. Somit erhalten wir:

$$G = 2\left(\frac{e^2}{h}\right) \cdot 2\left(\frac{l_e \cdot w}{\lambda_F \cdot L}\right) = \frac{4e^2 l_e w}{\lambda_F h L}.$$

Solange l_e kleiner bleibt als L, so bleibt G invariant gegenüber der Verkleinerung der Größenordnung (der Breite w und der Länge L). Wird die Leitung seitlich verkleinert, würde die Fermiwellenlänge der Elektronen unterschritten werden, was dazu führt, dass die Komponente isolierend wirkt. Daher ist nur ein finaler Wert für obige Gleichung zulässig, um weitere Leitfähigkeit der Komponente zu garantieren. Die Kapazität einer Komponente der Größe L liegt bei etwa $0{,}04\varepsilon L$. Die entsprechende Zeit für das „Entladen“ liegt bei

$$T = \frac{0{,}1}{\sqrt{Dichte\ der\ Komponenten}}.$$

Diese nun festgestellte Begrenzung ist in der Abbildung als gestrichelte Linie *composants quantique* dargestellt [17].

4.6) Ein vorzeitiger Schluss

In der Abbildung sind außerdem die MOS-FET Komponenten aus dem Jahr 2006 eingezeichnet. Es ist zu erkennen, dass noch ein wenig Spielraum im Bereich der Dichte und der Operationszeit der Komponenten existiert.

Allerdings nähert man sich schon immer mehr der absoluten physikalischen Grenzen, die durch die Linien dargestellt sind. Eigentlich ist diese Annäherung sogar noch stärker, da in obiger Betrachtung andere Effekte und Einflüsse wie beispielsweise die Coulombeffekte in den MOS-FETs weggelassen wurden, da sie den Rahmen dieser Facharbeit sprengen. Nimmt man diesen beispielsweise dazu, so ist die Veränderung der obigen Konzeption notwendig.

V. Parallele Multiprozessorsysteme

5.1) Einleitung

Wie im vorangehenden Kapitel bereits festgestellt, ist ein Ende der Computerminiaturisierung spätestens auf atomarer Ebene zu erwarten.

Die Grundidee von parallelen Multiprozessoren ist mehrere Prozessoren zu verbinden und so ein mächtigeres Computersystem zu schaffen. Im Grunde sind Parallelrechner eine Ansammlung von Prozessoren, die auf eine Art verbunden sind, die es den Prozessoren erlaubt Daten auszutauschen und ihre Aktivität zu koordinieren. Parallele Rechner brauchen, um ihre volle Rechenleistung und die Effizienz auszuschöpfen, auch parallele Algorithmen [18].

Parallele Abarbeitung bedeutet also im Prinzip das gleichzeitige Abarbeiten mehrerer unabhängiger Tasks auf mehreren Prozessoren oder das Aufteilen eines einzelnen Tasks auf mehrere Prozessoren. Im Folgenden werden wir eine Einteilung der Parallelrechner wagen, indem wir verschiedene Klassifikationsschemata betrachten.

5.2) Klassifikation

5.2.1) Das Flynnsche Klassifikationsschema

Parallelrechner lassen sich verschieden klassifizieren. Schenkt man dem so genannten *Flynn´schen Klassifikationsschema* (*Flynn'sche Taxonomie*) Bedeutung, so beachtet man im Grunde die Klassifikation nach der Art der Befehlsausführung.

Das Schema teilt verschiedene Rechner danach ein, welche Operationen auf welchen Daten ausgeführt werden können. Man unterscheidet SISD (*Single Instruction, Single Data*), SIMD (*Single Instruction, Multiple Data*), MISD (*Multiple Instruction, Single Data*) und MIMD (*Multiple Instruction, Multiple Data*), die im Folgenden besprochen werden sollen. Da uns aufgrund der Thematik nur die Multiprozessoren interessieren, betrachten wir im Folgenden auch nur SIMD und MIMD Rechner.

5.2.1.1) Single Instruction Multiple Data

Hierbei führen mehrere Prozessoren zu einem Zeitpunkt denselben Befehl aus (Prozessorarray). Solche Rechner kommen nur in speziellen Anwendungen wie der Bildverarbeitung oder der Spracherkennung vor. Jeder einzelner Prozessor kann einen lokalen Speicher besitzen. Hierbei spricht man von einem Distributed-Memory-Rechner.

Die verbreiteteste Ausführung eines SIMD-Rechners hat einen gemeinsamen Speicher. Allerdings müssen die einzelnen Zugriffe auf den Speicher organisiert werden, da jeder Prozessor jeden Teil des Speichers schreiben und lesen darf und es so zu Zugriffskonflikten kommen kann. Viele Großrechner der 70er und 80er Jahre waren von diesem Typ, der man auch Vektorrechner nannte.

Zu dieser Art von Rechnern sind auch Personal Computer und Workstations mit jeweils mehreren Prozessoren zu zählen. Ein aktuelles Beispiel ist der Rechner *MasPar* (in Karlsruhe mit 16 000 Prozessoren, Connection Machine). Im Allgemeinen sind bis zu 65000 Prozessoren in einem solchen Prozessorarray zu realisieren [19].

5.2.1.2) Multiple Instruction Multiple Data

Eigentlich sollte noch vor dieser Klassifikation der MISD Rechner besprochen werden, allerdings sind diese Rechner unzweckmäßig und werden nicht verwendet. Ein MISD-Rechner wendet mehrere Anweisungsströme mit einer Verarbeitungseinheit an. Wenden wir uns nun den MIMD-Rechnern zu.

Hierbei steuern mehrere Anweisungsströme eine gleiche Anzahl von Verarbeitungseinheiten. Wie bei einem SIMD-Rechner gibt es auch beim MIMD Rechner sowohl Distributed-Memory sowie Shared-Memory, wobei bei letzterem die Zugriffe auf den Speicher kontrolliert werden müssen.

Da die Daten für alle Prozessoren hierbei auf demselben Speicher liegen, ist ein Austausch nicht notwendig, wohingegen bei einem Distributed- Memory- Rechner der Austausch mit Hilfe von Kanälen realisiert wird.

Zugriffe sind nur auf lokale Speicher realisierbar, weswegen keine globale Kontrolle der Speicherzugriffe notwendig ist. In diese Kategorie fallen die meisten Multiprozessorsysteme oder Multiprozessoren.

5.2.2) Klassifikation nach der Speicherorganisation

Hierbei unterscheidet man zwischen Rechnern, die für die einzelnen Prozessoren einen einzigen Speicher vorsehen, wobei die Zugriffe von einer Speicheransteuerung geregelt und gelenkt werden. Bei vielen Prozessoren ist der Hauptspeicher allerdings der begrenzende Faktor. Es lassen sich Systeme mit bis zu 16 Prozessoren so realisieren. Wenn man mehr Prozessoren zu einem System zusammenschließen will, so ist die bessere Alternative ein Parallelrechner mit verteiltem Speicher. Hierbei hat jeder Prozessor seinen eigenen lokalen Speicher, worauf er im Allgemeinen auch nur zugreifen kann. Um trotzdem auf die Daten anderer Prozessoren zugreifen zu können, müssen „Botschaften“ ausgetauscht werden. Hierbei sei vor allem auf das Grid-Computing oder Metacomputer hingewiesen, was den Zusammenschluss (weit) entfernter Computer meint [20].

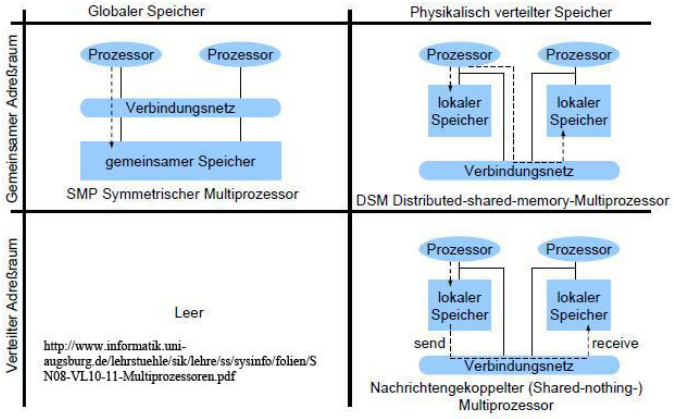

Fig. 4. Diese Abbildung zeigt verschiedene Kategorisierungen durch Speicherorganisation. Dabei wird Adressraum in gemeinsamen und verteilten eingeteilt. Der Speicher an sich kann sowohl global als auch verteilt sein, weswegen eine solche Betrachtung auch sinnvoll ist.

Da bei symmetrischen Multiprozessoren der Speicher gemeinsam genutzt werden soll, sollte auch ein einheitlicher und damit gemeinsamer Adressraum vorhanden sein. Beim physikalisch verteilten Speichern muss man allerdings zwischen einem gemeinsamen und verteilten Adressraum unterscheiden.

DSM (Distributed- shared- memory- Multiprozessoren) haben zwar mehrere lokale Speicher, die aber einen gemeinsamen Adressraum haben. Selbstverständlich ist dafür eine Einheit notwendig, die die Schreib-Leseoperationen koordiniert, da bei solchen Systemen der Prozessor quasi aus seinem Speicher „springen“ und auf den Speicher des andere zugreifen kann.

Bei Shared- nothing Multiprozessoren dagegen ist kein gemeinsamer Adressraum vorhanden, weswegen die Adresse bei einem eventuellen Zugriff auf den Fremdspeicher erst übersetzt werden müssen. Das Verbindungsnetz tritt hierbei also auch als „Übersetzer“ für physikalische Speicheradressen auf.

5.2.3) Vergleich verschiedener Multiprozessorsysteme

Betrachtet man die oben genannten Speicherorganisationen und das *Flynn'sche Modell*, so gibt es natürlich dutzende Einteilungsmöglichkeiten. Wir wollen nun aber überlegen, wie eine einfache Kategorisierung stattfinden kann. Im Folgenden unterscheiden wir die bereits oben genannten Symmetrischen Multiprozessorsysteme von den Asymmetrischen und den „Massive“ Parallelprozessoren.

5.2.3.1) Asymmetrische Multiprozessoren

Eine der Hauptcharakteristika dieses Prozessortypus ist das *Master- Slave- Modell*. Ein Prozessor übernimmt dabei das Betriebssystem und weist die verschiedenen Prozesse an die jeweiligen anderen Prozessoren zu.

Der „Hauptprozessor“ ist dabei der so genannte *Master* und die anderen die *Slaves*. Alle Prozessoren sind für gewöhnlich an einen Systembus gebunden, der die Daten in einen gemeinsamen Speicher transportieren kann.

Die einzelnen Betriebssystemaufrufe werden alle zum Master weitergeleitet, der sie dann bearbeitet. Die folgende Darstellung soll diesen Sachverhalt illustrieren.

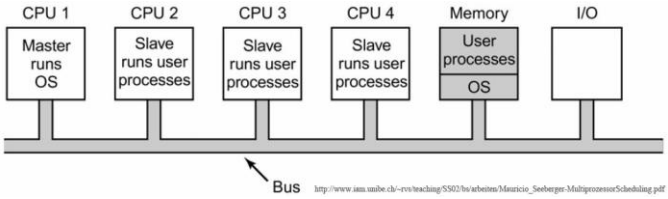

Fig. 5. Der „Hauptprozesser“ (ganz links im Bild), bezeichnet als *Master*, kontrolliert und organisiert alle Zugriffe der anderen CPUs, bezeichnet als *Slaves*. Der Speicher und die IO- Geräte werden gemeinsam genutzt.

Nicht jede CPU muss dabei Zugang zu allen Systembefehlen haben, auch ein bestimmter Umfang an Systembefehlen für einen bestimmten Prozessor lassen sich einrichten.

Der wohl größte Vorteil dieses Systems ist seine einfache Implementierung, da beispielsweise keine komplizierten oder einfach speziellen Scheduling Algorithmen implementiert werden müssen.

Außerdem ist lediglich eine Liste mit allen abzuarbeitenden Prozessen gebraucht, was bedeutet, dass die Implementierung und das Benutzen von Datenstrukturen auch recht einfach geregelt sind.

Spätestens nun merkt man, dass es eventuell zu einer zu starken Auslastung des *Masters* kommen kann, da jeder Prozessor über ihn angesprochen wird, was gleichzeitig auch der wohl größte Nachteil dieses Prozessortyps ist. Der *Master* ist also im Grunde der „Flaschenhals“ dieser Betrachtung [21].

5.2.3.2) Symmetrische Multiprozessoren

Wie im ASMP (*Asymmetrischen Multiprocessing*) sind auch hier alle Prozessoren an einen gemeinsamen Systembus gekoppelt. Hierbei kann aber nun jeder angekoppelte Prozessor Code des Betriebssystems ausführen, sodass eine solche Realisierung eines Masters wie im ASMP nicht notwendig ist. Der Flaschenhals obiger Darstellung wird also umgangen indem die Prozessorlast verteilt wird.

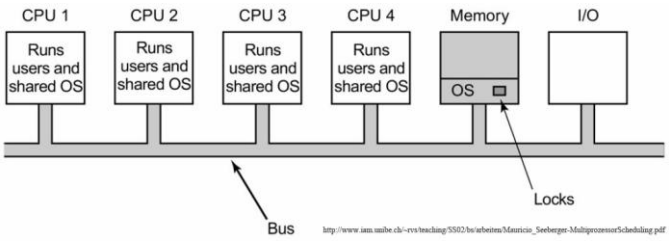

Fig. 6. Dargestellt sind vier CPUs, die alle einen gemeinsamen Speicher haben, Jeder Prozessor kann eine Hand voll Befehle ausführen. Sobald ein Prozessor auf den Speicher zugreifen will, muss eine Anfrage versendet werden, ob auf den Speicher zugegriffen werden kann. Wollen verschiedene CPUs auf dasselbe Datum zugreifen, so erhält lediglich ein Prozessor die nötigen Rechte, die anderen warten.

Es gibt aber einige Fragestellungen, die nun neu aufkommen. Was passiert beispielsweise, wenn verschiedene Prozessoren ein und denselben Prozess ausführen wollen. Derartige Probleme führen eventuell schnell zu Dateninkonsistenzen.

Um diesem Problem zu entgehen, lässt sich ein Monitor, der überwacht, dass ein bestimmter Code nur von einem Prozessor genutzt werden kann, einrichten.

Wollen zwei verschiedene Prozessoren auf dasselbe Datum warten, so erhält ein Prozessor die nötigen Rechte, die anderen müssen warten, was aber keine wirkliche Verbesserung gegenüber dem *Master- Slave- Modell* darstellt.

Ein Vorteil ist aber, dass sich symmetrische Multiprozessoren auf einfachen Wegen verbessern lassen. Da viele Teile des Systems unabhängig voneinander arbeiten können, kann man sich nun überlegen, ob man nicht einen solchen oben erwähnten Monitor für das gesamte Betriebssystem nimmt, sondern für viele kleine *critical regions*.

Es kann natürlich immer noch dazu kommen, dass verschiedene Prozessoren auf ein und dasselbe Areal im Speicher zugreifen wollen. Das Problem lässt sich also nicht gänzlich aus der Welt schaffen. Allerdings lässt sich durch diese Aufteilung einen höheren Grad an Parallelität gewinnen. Man kann natürlich auch mit verschiedenen Synchronisationsverfahren wie Semaphoren die Konsistenz der Daten realisieren [22].

Viele moderne Multiprozessorsysteme nutzen diese Art der Organisation. Das Problem oder die Schwierigkeit bei der Programmierung des passenden Betriebssystems liegt allerdings darin, das Betriebssystem in wirklich voneinander unabhängige Regionen aufzuteilen. Außerdem muss die Synchronisation der kritischen Tabellen und Daten realisiert sein.

5.2.3.3) Massive Multiprocessing

Hierbei handelt es sich meist um verteilte Systeme (*distributed systems*), was im Grunde das Zusammenschließen von mehreren unabhängigen Rechnern zu einem Rechennetz. Jeder einzelne Rechner besitzt einen eigenen Arbeitsspeicher sowie ein Betriebssystem. Sie teilen sich lediglich unter Umständen gemeinsame Ein-/ und Ausgabegeräte.

Dieser Ansatz ist die vielleicht einfachste Variante ein Multiprozessorsystem zu organisieren und zu realisieren. Es zieht aber einige negative Konsequenzen mit sich.

Jeder Prozessor besitzt seine eigene Prozesstabelle, was bedeutet, dass einzelne Prozesse nicht zwischen verschiedenen Prozessoren verteilt werden können. Dadurch ist die Last auf verschiedene Prozessoren ungleich verteilt ist, was im schlimmsten Fall dazu führen kann, dass ein Prozessor unter Volllast läuft, während andere nur einen Bruchteil ihrer Leistung erbringen.

Die Kommunikation zwischen verschiedenen Prozessoren ist teuer und vor allem langsam, was ein weiterer negativer Punkt ist. Allerdings gibt es hierfür einige Optimierungsverfahren, die nicht weiter besprochen werden, da sie sonst den Rahmen dieser Arbeit sprengen.

Es lässt sich streiten, ob MPP (Massive Multiprocessing) bzw. Rechnernetze wirklich in die Kategorie der Multiprozessorsysteme gehören, allerdings müssen auch hier Prozesse über verschiedene Prozessoren verteilt werden, weswegen spezielle Multiprozessor Scheduling Algorithmen notwendig sind. Ob die einzelnen Prozessoren nun wirklich alle zusammen an einem gemeinsamen Bus kommunizieren oder alles über Netzwerkkabel oder Internet funktioniert, spielt für diese Betrachtung also keine große Rolle.

5.3) Prozesssynchronisation

5.2.1) Einleitung

Prozessorganisation ist in einem Multiprozessorsystem unumgänglich. Nahezu jeder Scheduling- Algorithmus zieht kritische Daten, also Daten, die von mehreren Prozessoren bearbeitet werden wollen, heran. Um nun dafür zu sorgen, dass verschiedene Prozessoren nicht auf demselben Datum arbeiten bzw. dieses anfordern muss ein strenges Regelwerk her, die Rede ist vom *Mutual Exclusion Protocol* (*Mutex*).

5.2. 2) Test and Set Lock

Jedes dieser *Mutex*- Protokolle besitzt als elementare Operation *Test and Set Lock*. Hierbei wird ein Speicherwort gelesen und in ein Register geschrieben, während es zeitgleich eine 1 ins Wort schreibt.

Dieses Zeichen stellt sicher, dass die Datei in Verwendung ist und somit definitiv nicht abgefragt werden kann. Für das Lesen sowie das Schreiben ist jeweils ein Bustakt notwendig.

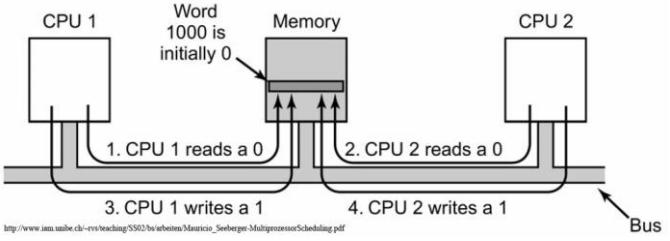

Fig. 7. In der Mitte der Abbildung ist der gemeinsame Speicher dargestellt, der von CPU 1 (links im Bild) und CPU 2 (rechts im Bild) genutzt wird. Der Bus ist als solcher markiert. Die Pfeile zeigen, dass von den beiden CPUs Schreib/- bzw. Leseoperationen durchgeführt werden.

In diesem Dualprozessorsystem wollen zwei verschiedene Prozessoren auf dasselbe Wort zugreifen um eine 1 hineinzuschreiben, damit das Datum ausschließlich von einem Prozessor genutzt werden kann.

Wenn nun CPU1 das Wort liest, so wird es im nächsten Schritt das „Wird bereits genutzt“ Bit gesetzt. Wenn aber nachdem CPU1 gelesen hat, CPU2 ein Takt danach anfängt zu lesen, so gehen beide Prozessoren davon aus, dass sie das Datum für sich haben.

Nun kommt es aber dazu, dass beide gleichzeitig daran arbeiten wollen, weil beide CPUs eine 0 gelesen haben. Beide Prozessoren haben sich an die Mutex Regel gehalten, allerdings kommt es trotzdem zu einer Fehlfunktion.

Die Lösung ist, vor dem Lesen den Bus zu sperren, sodass problemlos auf die Datei zugegriffen werden kann, ohne Gefahr zu laufen, dass der beschriebene Fehler erneut auftreten kann. Nach dem Schreibvorgang wird der Bus natürlich wieder freigegeben, damit die anderen Prozessoren darauf zugreifen können. Hierbei entstehen nun allerdings andere Probleme [23].

5.2. 3) Ethernet binary exponential backoff algorithm

Wenn ein Bus nun gesperrt ist, so überprüfen die anderen Prozessoren ununterbrochen die Leitung, ob der Lese-/ Schreibvorgang abgeschlossen ist.

Man nennt dies auch *Spinning*. Allerdings wird dadurch Rechenleistung des jeweiligen Prozessors verschwendet und es kommt eventuell zu einer zu großen Last für den Bus, was das Gesamtsystem verlangsamt.

Eine mögliche Variante um das Problem vom *Spinning* zu umgehen, ist der *Ethernet binary exponential backoff algorithm*. Hierbei wird nicht kontinuierlich überprüft, ob der Bus frei ist, sondern in Abständen, die durch das Einfügen von Verzögerungen, realisiert werden.

Dabei wächst die Dauer einer solchen Verzögerung exponentiell bis zu einem gegebenen Maximum, wo die Dauer der Verzögerung dann wieder bei einem Takt anfängt. Durch kurze Reaktionszeiten von beispielsweise einem oder zwei Takte, erhält man erneut das Problem vom *Spinning*.

Durch zu lange Reaktionszeiten kann der Prozessor aber mehrere Buszyklen, in denen dieser frei ist, verpassen [24].

5.2. 4) Variablen Spinning

Eine andere Methode um *Spinning* vorzubeugen ist jeder CPU eine private Variable zu geben, die dem jeweiligen Prozessor dann mitteilt, ob die angeforderte Datei wirklich benutzt werden kann. Dieses Verhalten wird manchmal auch als *privates Spinning* (seltener lokales *Spinning*) bezeichnet, da der Prozessor stets nach seiner eigenen Variable nachfragt.

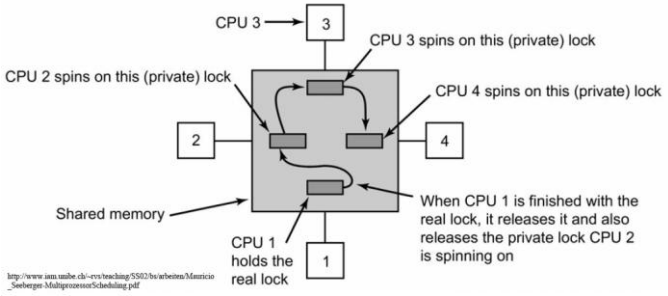

Fig. 8. Dargestellt sind vier CPUs, die „gleichzeitig“ auf einen gemeinsam genutzten Speicher zugreifen wollen. Um bei gleichzeitigem Zugriff auf ein Datum Fehler zu vermeiden werden zusätzliche Variablen eingesetzt, die den Zugriff regeln.

Nachdem CPU 1 die Datei benutzt hat, wird die eigene Variable „entlockt“ sowie die Variable für CPU 2 so gesetzt, dass CPU 2 nun auf die Datei zugreifen kann.

Dies geschieht dann auch mit den anderen Prozessoren, was stark an eine verkettete Liste erinnert. Wenn ein Prozessor

merkt, dass die Datei benutzt wird, so muss er sich an das Ende der Kette anreihen.

Der Vorteil ist evident: Die einzelnen Variablen sind im Cache eines jeden Prozessors, sodass *Spinning* problemlos durchgeführt werden kann, ohne dabei den Bus weiter zu belasten. Der Algorithmus muss allerdings darauf achten, dass sich nicht zwei verschiedene Prozessoren an das Ende gleichzeitig einfügen, was ebenfalls ein kleines Problem darstellt. Trotzdem gilt dieser Algorithmus als effizient und zuverlässig [25].

5.4) Amdahls Gesetz

5.4.1) Der Ursprung

Gene Amdahl arbeitete in den 60er Jahren bei *IBM* als Computerentwickler und entwickelte vor allem leistungsfähige Vektorrechner. Während der Computerrevolution stieg natürlich auch der Anspruch der einzelnen Unternehmen und Forschungsgruppierungen. Für das mittlerweile immer kühnere Forschungsprojekt waren immer leistungsfähigere Rechner notwendig. Da die Entwicklung der Nanotechnologien und der Computerminiaturisierung zwar immer weit fortschritt, diese Entwicklung aber schlichtweg immer noch nicht reichte, erhofften die Wissenschaftler die nötige Leistung in (parallelen) Multiprozessorsystemen zu finden. Es wurde mathematisch bewiesen, dass eine enorme Steigerung der Leistung möglich war, doch diese blieb aus. Computer wurden zwar leistungsfähiger, allerdings in keinster Weise in dem Ausmaß, den man sich erhofft hatte [26].

5.4. 2) Defintion des Amdahlschen Gesetzes

Wie gesagt, kam es nicht zu dem Leistungsaufschwung, den man sich erhofft hatte. Damit solch ein Netzwerk an Rechnern mit n Prozessoren wirklich das n- fache der Leistung eines einzelnen Rechners erreicht, müssen die einzelnen Prozessoren wirklich parallel, also zu jeder Zeit gleichzeitig arbeiten können. Allerdings gibt es bereits in einfacheren Programmen Programmsequenzen, die sich nicht unbegrenzt parallelisieren lassen. An diesem Teil des Programms kann also nur eine feste Anzahl an Prozessoren gleichzeitig arbeiten, die restlichen arbeiten zu diesem Zeitpunkt nicht.

Damit steht aber nicht die maximale Leistung aller Prozessoren zur Verfügung, was die gesamte Rechnerleistung stark mindert. Wenn ein Programmteil lediglich von einem einzigen Prozessor berechnet werden kann, so nennt man diesen Code „sequentiell“, ansonsten bezeichnet man ihn als „parallel“ [27].

Das *Amdahlsche Gesetz* ist folgendermaßen definiert [28]:

Jeder Algorithmus besteht aus Teilen, welche sich nicht parallelisieren lassen. Seien s der serielle und p der parallele Anteil eines Algorithmus. Somit gilt: $s + p = 1$. Die Rechenzeit lässt sich somit auf einem Prozessor $t_1 \longrightarrow s + p$ und (bei idealem Verhalten) die Rechenzeit auf k Prozessoren durch $t_k \longrightarrow s + \frac{p}{k}$ ersetzen. Daraus folgt eine obere Schranke für den maximal erreichbaren Speedup:

$$S_{k,max} = \frac{s+p}{s+\frac{p}{k}} = \frac{1}{s+\frac{1-s}{k}} \leq \frac{1}{s}$$

Die Leistung eines Systems bei Hinzunahme von n Prozessoren ist geringer als die n - fache Leistung eines Einzelprozessors. Handelt es sich als um ein System von n Prozessoren bringt der $(n-1)$te hinzugenommene Prozessor einen größeren Leistungszuwachs als der zuletzt hinzugenommene Prozessor [29].

Damit stellen wir also fest, dass die Leistung von Systemen nicht parallel mit der Zahl der Prozessoren steigt. Die Maximalleistung eines solchen Zusammenschlusses von Prozessoren konvergiert gegen einen fixen Beschleunigungsfaktor.

5.4. 3) Gründe des Verlustes an Leistungssteigerung

Mit steigender Anzahl an Prozessoren in einem System steigt auch der Kommunikationsaufwand zwischen Prozessoren.

Ab einer bestimmten Anzahl an Prozessen ist der Aufwand für Interprozesskommunikation größer als für das Abarbeiten des Problems. Dies wird auch als Overhead bezeichnet, weswegen obige Formel von Amdahl eigentlich

$$S_{k,max} = \frac{1}{s+O(k)+\frac{1-s}{k}} \leq \frac{1}{s}$$

lautet. Außerdem gibt es einige Teile im Programmcode, die gar nicht oder nur sehr aufwendig parallelisiert werden können, was das vollkommene parallelisieren unmöglich macht.

Ein anderes Problem stellt die Aufgabenteilung dar, da nicht für alle Programme a priori definiert ist, wie diese bei n parallelen Prozessoren abzuarbeiten sind. Eine solche Aufgabenteilung stellt also auch einige Schwierigkeiten dar. Außerdem beschränkt die Bandbreite des Busses die sinnvolle Anzahl an Prozessoren [30].

5.5) Multiprozessor Scheduling

Wir haben nun bereits festgestellt, dass sich ein Programm bzw. ein System nur bis zu einer Grenze sinnvoll parallelisieren lässt.

Nun betrachten wir den parallelisierten Teil und wollen uns genauer das Scheduling der einzelnen Prozesse anschauen. In einem Monoprozessorsystem kümmert sich der Prozessor nur darum, welcher Prozess als nächstes ausgeführt werden soll, wohingegen bei Multiprozessorsystemen noch die Frage relevant ist, wo dieser Prozess ausgeführt werden soll.

Außerdem kann es sein, dass manche Prozesse voneinander abhängig sind, weswegen eine leichte Trennung zum Teil nicht möglich ist. Betrachten wir hierfür die folgenden Multiprozessor Scheduling Algorithmen.

5.5. 1) List Timesharing

Dieser sehr einfache Scheduling Algorithmus ist für

unabhängige Prozesse gedacht und stellt eine systemweite Datenstruktur für alle bereiten Prozesse vor. Dabei listet man die Prozesse beispielsweise mit ihrer Priorität zusammen in einer verketteten Liste zusammen.

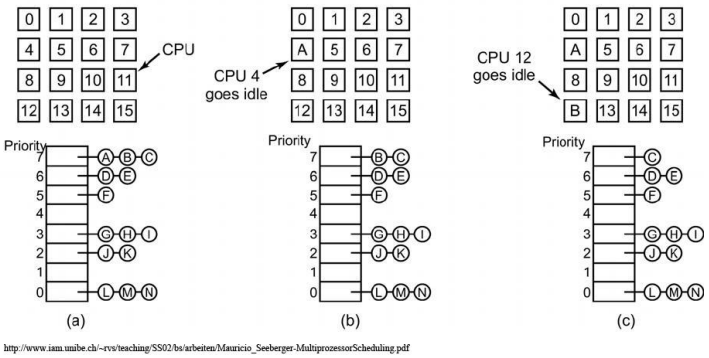

Fig. 9. Dargestellt sind drei Grafiken, die einen zeitlichen Ablauf darstellen sollen, in der 16 verschiedene Prozessoren, beschriftet von 0 bis 15, verschiedene Prozesse abarbeiten sollen. Die Prozesse sind hierbei in einer Tabelle oder einer Liste gespeichert und haben jeweils eine eigene Priorität. So haben Prozess G, H und I in a die Priorität 3. 7 ist hierbei die höchste Priorität, die ein Prozess haben kann.

Betrachten wir dazu Abbildung 9. Diese Abbildung stellt einen zeitlichen Ablauf dar. Wir werden uns am Anfang der Abbildung ganz links widmen, das ist die Grafik unter der (a) eingetragen ist. Hier sind zu Beginn 16 verschiedene Prozessoren beschriftet von 0 bis 15 abgebildet. Darunter finden wir 14 verschiedene Prozesse, die auf die Abarbeitung warten. Jeder dieser Prozesse hat eine eigene Priorität, so haben beispielsweise Prozess A, B und C die Priorität 7. In Grafik b sehen wir nun, dass CPU 4 Prozess A übernimmt und bearbeitet, damit ist CPU 4 bereits „beschäftigt“ und kann vorübergehend keine weiteren Anfragen annehmen. Es ist auch erkennbar, dass Prozess A aus der Liste der Prozesse verschwindet. Im nächsten Schritt, die Rede ist von Grafik c, übernimmt CPU 12 Prozess B, der ebenfalls die Priorität 7 hatte. Im Folgenden werden alle Prozessoren, mit Prozessen aus der Liste belegt. Warum nun CPU 4 und 10 ausgewählt wurden und nicht 0 und 1, was naheliegender wäre, da sie weiter vorne in der Liste liegen, ist damit zu erklären, dass immer eine CPU gesucht wird, die unbeschäftigt ist. Es ist daher anzunehmen, dass die anderen CPUs einen Prozess bearbeiteten, als die Anfrage für Prozess A oder B ankamen.

Ein großer Vorteil ist wie erwähnt die wirklich einfache Implementierung. Es gibt außerdem keinen Prozess der sich einfach im Leerlauf befindet, da die Prozessoren gleichmäßig verteilt werden. Allerdings kann es durch das systemweite Umsetzen einer solchen Prozessliste dazu führen, dass gerade diese Ressource stark ausgelastet wird, weswegen sich quasi Prozessorwarteschlangen bilden [31].

5.5. 2) Smart Scheduling

Wenn das Zeitintervall, welches für einen bestimmten Prozess bereitgestellt wurde, abgelaufen ist, so geschieht ein Kontextwechsel. Es lässt sich daher auch ein *Round- Robin* ähnlichen Scheduling Algorithmus wählen, welcher sich bereits auf Monoprozessorsystemen bewährt hat. Auf einem solchen System ist dieser Scheduling Algorithmus relativ linear. Für jeden Prozess wird für ein bekanntes Zeitintervall der Prozessor zur Verfügung gestellt. Nach diesem Zeitintervall wird der Prozess in eine Warteschlange eingereiht, sodass dieser zu einem späteren Zeitpunkt wieder seine Berechnungen fortführen kann. Unterdessen erhalten andere Prozesse die Ressourcen, die sie aber auch nur für das bestimmte Zeitintervall zur Verfügung gestellt bekommen.

In einem Multiprozessorsystem kann es nun zu einigen Schwierigkeiten kommen. Beispielsweise könnte ein Prozess einen *Spin- Lock* auf eine bestimmte kritische Systemressource haben. Wenn dieser Prozess nun in die Warteschlange eingereiht wird, weil das zugehörige Zeitintervall verstrichen ist, kann die entsprechende kritische Ressource von anderen Prozessen, die nun an der Reihe sind, nicht benutzt werden. So können die anderen Prozesse nichts anderes als *Spinning* verüben, solange der Prozess, der den Lock enthält, die Ressource nicht freigibt. Da der Prozess aber nun weiter hinten in der Warteschlange steht, bedeutet dies eine große Verschwendung von Rechenzeit, dem mit so genannten *Smart Scheduling Algorithmen* entgegenzuwirken ist. Hierbei wird jedem Prozess noch ein Flag zugeordnet, welches angibt, ob dieser einen *Spin- Lock* enthält. Wenn ein Prozess gerade ausgeführt wird, der diesen Spin- Lock enthält, so wird der Scheduler diesem Prozess eventuell mehr Zeit zur Verfügung stellen, damit dieser die Ressourcen auch freigeben kann [32].

5.5. 3) Affinity Scheduling

Ein anderes wichtiges Thema bei den Multiprozessorsystemen ist die Tatsache, dass sich trotz gleicher Bauarten zwei Prozessoren zur Laufzeit voneinander unterscheiden können. Dies gilt beispielsweise für den Cache. Wenn Prozess *A* schon für längere Zeit auf CPU lief, ist der Cache von CPU *k* voll von den Blöcken von Prozess *A*. Wird nach einiger Zeit wieder Prozess *A* auf diesem Prozessor ausgeführt, wird er höchstwahrscheinlich noch einige relevante Blöcke enthalte, was bedeutet, dass Prozess *A* besser auf CPU *k* läuft. Vorweggeladene Blöcke im Cache erhöhen wie wir wissen die Geschwindigkeit des Prozessors, sodass das oben beschriebene Verhalten leicht zu erklären ist.

Einige Multiprozessoren verwenden daher das *affinity scheduling* (*Vaswani* und *Zahorjan*, 1991). Die Idee ist dabei, massiven Aufwand zu betreiben, um nun Prozess *A* auf CPU *k* auszuführen, da dieser Prozess auf dieser CPU schneller berechnet werden kann. Eine Möglichkeit diese Affinität zu erzeugen, liegt in der Verwendung eines *two- level scheduling* Algorithmus. Wenn nun ein Prozess erzeugt wird, so wird er, beispielsweise auf Basis der kleinsten momentanen Last, einer bestimmten CPU zugeteilt. Diese Zuweisung ist gleichzeitig die oberste Stufe des Algorithmus. Nach und nach erhält jede CPU seine eigene Sammlung an Prozessen.

Die unterste Stufe des Algorithmus ist im Prinzip das eigentliche Scheduling. Hier wird nun versucht die Prozesse auf die CPUs zu verteilen, sodass obiges Schema mit dem Cache möglichst effizient durchgeführt werden kann. Wenn einer CPU kein Prozess zur Verfügung steht, nimmt der Prozess. Wenn es nun dazu kommt, dass eine CPU nichts

mehr zu tun hat, so kann die obere Schicht des *Affinity-Scheduling*, wartende Prozesse anderer CPUs diesem unbeschäftigten Prozessor zuweisen.

Die Vorteile dieses Algorithmus sind ersichtlich. Die Last wird einigermaßen gleichmäßig auf die CPUs verteilt. Außerdem wird, wann immer möglich, ein größtmöglicher Nutzen aus der zeitlichen und örtlichen Lokalität in Cache Speichern gezogen. Wenn man einem Prozessor nun seine eigene Prozessliste gibt, so minimiert das die Konkurrenz der anderen CPUs um diese Prozessorliste, da der Zugriff auf diese Liste von anderen unbeschäftigten CPUs relativ selten ist [33].

5.5. 4) Space Sharing

Es ist relativ selten, dass alle oder nahezu alle Prozesse voneinander abhängen. Heutzutage arbeiten die verschiedenen Betriebssysteme eher mit Threads. Ein Prozess besteht nun aus einem oder mehreren Threads, die sich im Grunde wie kleinere Prozesse verhalten. Daher kann der Scheduler beide in seinen Algorithmus integrieren. Allerdings hängen die Threads meist stärker voneinander ab, weswegen sich keiner der genannten Algorithmen gut verwenden lässt. Wenn ein Prozessor mehrere Threads, die voneinander abhängen, abarbeitet, so wären die Ressourcen eines Multiprozessorsystems nicht ausgeschöpft. So bringt es nämlich keinen Vorteil ein System mit mehreren Prozessoren einzusetzen, wenn diese sich je so verhalten wie einzelne Prozessoren. Daher wäre es effektiv, wenn alle Threads wirklich parallel abgearbeitet werden können. Hierbei spricht man dann von Space Sharing [34].

5.5. 4.1) Statische Partitionierung

Angenommen eine Gruppe von Threads seien zusammen von einem Prozess erzeugt wurden. Der Scheduler überprüft nun, ob genug Prozessoren frei sind, um je einen Thread aufzunehmen. Ist dies nicht der Fall, so wird gewartet, bis genug Prozessoren frei zur Aufnahme der Threads sind. Sobald der Thread auf dem jeweiligen Prozessor gestartet wurde, bleibt er dort, bis er terminiert. Erst dann wird der Prozessor wieder in die Liste der freien Prozessoren eingetragen. Wenn ein Thread auf Zugriffe zur Peripherie wie IO Geräte wartet, gibt dieser den Prozessor nicht frei, was bedeutet, dass sich der Prozessor solange im Leerlauf befindet.

In diesem Vorgang werden die Prozessoren in kleine statische Gruppierungen partitioniert, welche dann die Threads dieses Prozesses ausführen.

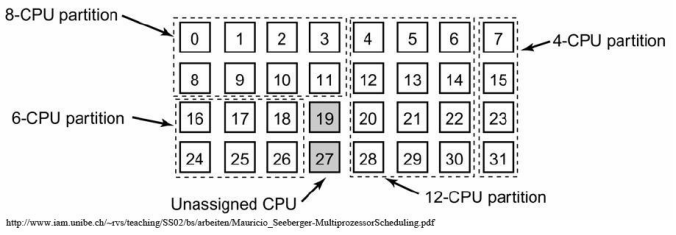

Fig. 10. Dargestellt sind 32 Prozessoren, die in vier Partitionen der Größe 4,6,8 und 12 eingeteilt sind. Die beiden grau markierten Prozessoren sind keiner Partition zugewiesen. Wenn nun ein Prozess abgearbeitet werden will, so kann er dies in obiger Abbildung nur dann, wenn er lediglich zwei Threads besitzt. Im anderen Fall muss er warten, bis genug Prozessoren frei sind.

Damit ist nun auch klar, dass sich die Anzahl an Partitionen mit der Zeit entsprechend der Belegung und Zuweisung der Prozessoren verändert.

Bisher ist geklärt, wie die Threads auf die CPUs verteilt werden, allerdings ist noch nicht geklärt, welche Prozesse bevorzugt werden, wenn mehrere Prozesse gleichzeitig anstehen und die Anzahl an Prozessoren nicht zur Bearbeitung aller solcher Prozesse reicht. Im Monoprozessorsystem kennen wir dafür viele verschiedene Scheduling Algorithmen, es bietet sich daher an, die Lösung analog zu übernehmen. Beispielsweise kann *Shortest Job First* auch für unsere Zwecke genutzt werden. Der analoge Prozess auf Multiprozessorsystemen wählt den Prozess aus, dessen Faktor $CPU\ Anzahl \cdot Laufzeit$ der kleinste der möglichen Kandidaten ist.

Allerdings ist in der Praxis eine solche Information oft nicht verfügbar, weswegen sich so ein Scheduler nur schwer realisieren lässt. *First come, first serve* ist nach Studien (*Krueger* et al., 1994) in der Tat nur schwer zu schlagen. Außerdem ist dieser Algorithmus auch wesentlich leichter zu implementieren [35].

5.5. 4.1) Dynamische Partitionierung

Im vorigen Partitionierungsmodell fordert ein Prozess einfach eine Anzahl an CPUs an, die es zur Abarbeitung seiner Threads benötigt. Bekommt er diese nicht, so wartet er bis die nötige Anzahl an Prozessen frei ist. Es gibt nun die Möglichkeit das Ganze ein wenig flexibler und dynamischer zu gestalten. Es handelt sich um die Einführung eines zentralen Servers, der festhält, welche Prozesse in der Bearbeitung stehen, welche sich in der Warteschlange befinden und die jeweiligen minimalen und maximalen Prozessoranforderungen der jeweiligen Prozesse (Tucker und Gupta, 1989). Nun fragt jeder Prozess den Server, wie viele Prozessoren noch frei stehen und wie viele er noch zusätzlich benutzen darf. Der Server passt dadurch seine Liste entsprechend der neuen Daten an.

Betrachten wir dazu als Beispiel einen Webserver, der eine beliebige Anzahl an Threads am Laufen haben kann. Angenommen der Webserver hat nun 10 Threads aktiv und es werden weitere 5 Prozessoren für andere Threads gebraucht. Dann muss der Webserver seine Last auf nun noch 5 Prozessoren senken, damit wieder 5 frei sind. Diese können dann genutzt werden. Wenn nun die nächsten 5 Webserver-Threads mit ihrer Verarbeitung fertig sind, könnten sie nun terminieren, anstatt neue Anfragen zu bearbeiten und so dem System vorübergehend fünf freie Prozessoren überlassen. Nun hat der Webserver immer noch fünf Prozessoren, die seine Weiterarbeit garantieren. Sobald das System wieder weniger ausgelastet ist, können wieder weitere Threads gestartet werden. Dieses Schema ermöglicht es, die Partitionsgrößen entsprechend zur gegenwärtigen Arbeitslast flexibel und dynamisch anzupassen [36].

5.5. 5) Gang Scheduling

Vorteil des Space- Sharing ist das Eliminieren der Multiprogrammierung, welche den durch Kontextwechsel entstehenden Overhead, beseitigt. Der größte Nachteil

dagegen ist die verschwendete Zeit, die entsteht, wenn eine CPU blockiert bzw. nichts zu tun hat, bis sie wieder freigegeben wird. Wir stellen also fest, dass vorgegangene Methoden nicht alle Nachteile zufriedenstellend beseitigen können. Ein weiterer Ansatz möchte nun die Vorteile beiden vorherigen Methoden zusammenführen. Die Prozesse sollen sich zum einen die gesamte zur Verfügung stehende Prozessorressource gleichmäßig teilen können und zum anderen sollen trotzdem mehrere Threads eines Prozesses gleichzeitig abgearbeitet werden können, um so die Parallelität zu gewährleisten. Die nun beschriebene Strategie soll vor allem für die Prozesse wirksam sein, die mehrere Threads kreieren, die dann untereinander kommunizieren können, was heutzutage für die meisten Prozesse zutrifft.

Im Folgenden soll gezeigt werden, was letztlich passieren kann, wenn zusammengehörige Prozesse oder Threads, die untereinander kommunizieren, allerdings getrennt geschedult werden. Dazu stellen wir uns folgendes vor: Wir haben ein Dualprozessorsystem mit zwei Prozessen, wobei diese je zwei Threads haben. Wir unterscheiden die Threads A_0 und A_1, die zum Prozess A gehören von B_0 und B_1 vom Prozess B.

Die Threads A_0 und B_0 werden auf CPU_0, die anderen beiden Threads auf CPU_1, mit *Timesharing Algorithmen* geschedult. Die Threads A_0 und A_1 müssen nun oft miteinander kommunizieren. Dies passiert folgendermaßen: A_0 sendet A_1 eine Meldung, welche von A_1 beantwortet wird, indem dieser dem Thread A_0 wieder eine Meldung schickt. Dies passiert dann sehr oft und hintereinander. Wie es der Zufall will, werden nun aber A_0 und B_1 auf den Prozessoren gestartet. Dazu ist Abbildung 11 zu betrachten.

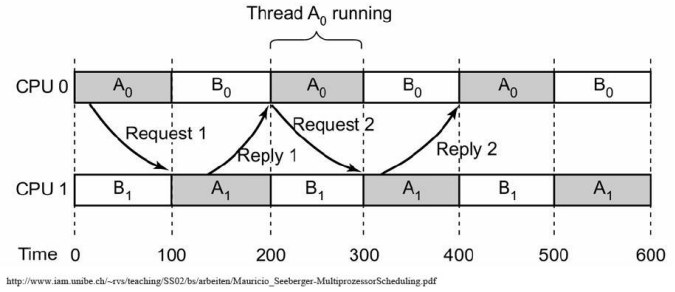

Fig. 11. Dargestellt sind die Threads A_0, A_1, B_0 und B_1, die auf zwei verschiedenen Prozessoren CPU 0 und CPU 1 ausgeführt werden. Dazu ist im unteren Teil der Abbildung eine Zeitleiste abgebildet, sodass das Abarbeiten der einzelnen Threads auch zeitlich mit verfolgt werden kann.

In den ersten 100 Zeiteinheiten verschickt der Threads A_0 eine Anfrage an Thread A_1, welcher auf CPU 1 ausgeführt wird. Dieser Thread verschickt daraufhin die erste Antwort an Thread A_0, welcher allerdings noch gar nicht ausgeführt wird, schließlich wird auf CPU 1 gerade B_1 ausgeführt. Erst einen Zeitabschnitt später erhält A_1, die Nachricht und beantwortet sie umgehend. Allerdings erhält A_0 die Antwort nicht sofort, da dieser Thread wiederrum nicht mehr auf CPU 0 ausgeführt wird. Analog funktioniert das auch für die Threads B_0 und B_1.

Wir erkennen aus Abbildung 11, dass es sich um eine Art „Frage- Antwort- Spiel“ handelt, welches sich alle 200 Zeiteinheiten wiederholt, was nicht sehr geeignet ist, da die Abarbeitung der Threads unnötig in die Länge gezogen wird, da sie ständig auf die Antwort der anderen Threads warten müssen. Die Lösung zu diesem Problem ist das Gang Scheduling, welches eine Verbesserung des Co- Schedulings (*Osterhout*, 1982) darstellt. Hierbei wird das *Gang Scheduling* in drei Teile eingeteilt:

i. Gruppen von zusammengehörigen Threads werden als eine Einheit betrachtet, die wir auch als „Gang“ bezeichnen können. Diese Gang wird daher auch genauso geschedult.
ii. Die Abarbeitung der Mitglieder einer Gang wird wirklich parallel abgearbeitet, dies geschieht auf verschiedenen Prozessoren mit Timesharing-Verfahren.
iii. Alle Mitglieder einer Gang werden zum gleichen Zeitabschnitt gestartet und beendet.

Das Besondere ist nun, dass alle Prozesse synchron geschedult werden. Das bedeutet, dass die Zeit analog zum *Round- Robin-* Verfahren in diskrete Zeiteinheiten eingeteilt wird. Bei dem Start einer neuen Zeiteinheit werde alle Prozesse neu geschedult und bekommen so einen neuen Thread. So wiederholt sich der Scheduling Algorithmus beim nächsten Start einer Zeiteinheit. Zwischen den Zeiteinheiten werden die Threads normal ausgeführt und es werden keine zusätzlichen Scheduling Entscheidungen gefällt. Wenn es nun dazu kommt, dass ein Thread aus irgendeinem Grund blockiert wird, so passiert das nur maximal eine Zeiteinheit lang, da dann sowieso wieder neu geschedult wird. Da so eine Zeiteinheit meist sehr kurz ist, ist dies also eine wunderbare Technik hierfür. Im Folgenden ist eine Darstellung gegebenen, die den Ablauf des Gang- Scheduling besser visualisieren kann.

Prozessoren

Zeit	0	1	2	3	4	5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

http://www.iam.unibe.ch/~rvs/teaching/SS02/bs/arbeiten/Mauricio_Seeberger-MultiprozessorScheduling.pdf

Fig. 12. Dargestellt sind 6 Prozessoren mit den 5 Prozessen A, B, C, D und E, die wiederum 24 bereite Threads besitzen. Außerdem ist auf der Ordinate die Zeit eingetragen, die bei der Abarbeitung der einzelnen Threads verstreicht. Werden im ersten Zeitabschnitt die Threads von Prozess A abgearbeitet. Im zweiten Quantum dann die Threads B_0 bis B_2 und C_0 bis C_2. So kommen als nächstes die Threads von den Prozessen D und E. Nach dem Abarbeiten beginnt der Prozess wieder von vorne.

In Abbildung 12 erkennt man nun klar das Konzept vom *Gang- Scheduling*, das Zusammenausführen von mehreren Threads, die zu einem Prozess gehören, statt sie zu trennen. So kann nach dem abschicken einer Meldung sofort darauf geantwortet werden ohne dass eine längere Wartezeit zu

erwarten ist (siehe Abb. 11). Dies ist in einem durchschnittlichen System umso wichtiger, da dort meist sehr viel mehr Threads ausgeführt werden, als Prozessoren vorhanden sind, was noch eine andere Folge hat. So wird ein Prozess nach seiner Ausführung eine ganze Weile nicht wieder ausgeführt werden. Wenn nun diese Threads aber miteinander kommunizieren, es aber nicht können weil sie eben nicht zusammen ausgeführt werden, so verzögert sich die Abarbeitung des entsprechenden Prozesses gewaltig. Es ist also ersichtlich, dass auch der *Gang- Scheduling* Algorithmus nicht perfekt ist. Wie Abbildung 12 schon zeigt, werden nicht alle Threads von Prozess E gleichzeitig ausgeführt. Dabei sei in Abbildung 12 auf Zeiteinheit 2 und 3 verwiesen. Der Grund dafür ist einfach, dass zu wenige Prozessoren (im Vergleich zur Threadzahl) existieren. Allerdings ist dies immer noch besser als dass die Reihenfolge der Threadausführung durch den reinen Zufall bestimmt wird [37].

VI. KONKLUSION

Wir stellen fest, dass die Idee der wirklichen parallelen Bearbeitung von Daten bereits sehr gut umgesetzt werden kann. Die Ideen sind ausgereift und werden bereits heute eingesetzt. Selbst auf Ebenen wie des Scheduling sind bereits einige nützliche Algorithmen und Strukturen bekannt, sodass der reibungslose Bearbeitungs- und Kommunikationszyklus von parallelen Multiprozessoren kein Problem darstellt. Mit Hilfe dieser Technologien wird die Steigerung der Leistungsfähigkeit auch in Zukunft in beachtlichem Maße stattfinden, allerdings muss das Mooresche Gesetz diesem Umstand angepasst werden. So wird sich die Anzahl der Transistoren auf einem Chip nur bis zu einem gewissen Punkt steigern, was durch thermische, quantentheoretische und informationstechnische Schwierigkeiten bedingt ist. Sie stellen die wesentlichen Grenzen der bisherigen Entwicklung von Computerchips dar, was aber den Anfang der parallelen Multiprozessortechnologie bedeutet. Diese stellen eine neue Klasse von Rechnersystemen dar und bieten eigene algorithmische Schwierigkeiten sowie deren Lösungen. Um Moore nochmal zu zitieren: „Wir stehen wahrscheinlich vor einer neuen Zeit. Es gibt sicherlich kein Ende der Kreativität“.

REFERENZEN

[1] http://download.intel.com/museum/Moores_Law/Printed_Materials/Moores_Law_2pg.pdf (Absatz 1)
[2] http://www.itwissen.info/definition/lexikon/Mooresches-Gesetz-Moores-law.html (Absatz 1)
[3] http://download.intel.com/museum/Moores_Law/Printed_Materials/Moores_Law_2pg.pdf (Absatz 2)
[4] http://www.intel.com/cd/corporate/pressroom/EMEA/DEU/archive/2005/212674.htm (Absatz 1)
[5] http://www.heise.de/newsticker/Moores-Gesetz-In-zehn-Jahren-ist-Schluss--/meldung/34415 (Absatz 1)
[6] http://www.intel.com/cd/corporate/pressroom/EMEA/DEU/archive/2005/212674.htm (Fortschritt für Jedermann, Absatz 2)
[7] http://www.intel.com/cd/corporate/pressroom/EMEA/DEU/archive/2005/212674.htm (Die Relationen des Mooreschen Gesetzes)
[8] http://download.intel.com/museum/Moores_Law/Printed_Materials/Moores_Law_Perspective.pdf
[9] http://www.ibm.com/developerworks/library/pa-microhist.html?ca=dgr-mw08MicroHistory (Before the flood: The 1960s)
[10] http://www.ibm.com/developerworks/library/pa-microhist.html?ca=dgr-mw08MicroHistory (The first three)
[11] http://lehrer2.rz.uni-karlsruhe.de/~berberich/desiree/jahr/zentraleinheit.html (ab 1978)
[12] http://www.cnano-rhone-alpes.org/spip.php?article19 (“Limitations physiques aux circuits integers”, Absatz 1)
[13] http://www.cnano-rhone-alpes.org/spip.php?article19 (“Dissipation d’énergie”)
[14] http://www.joergresag.privat.t-online.de/mybkhtml/chap34.htm (Absatz 1 und 2)
[15] http://www.cnano-rhone-alpes.org/spip.php?article19 (Le principe d’incertitude de Heisenberg)
[16] http://www.cnano-rhone-alpes.org/spip.php?article19 (Limite relativiste de propagation de l’information)
[17] http://www.cnano-rhone-alpes.org/spip.php?article19 (Le délai RC)
[18] http://www3.informatik.uni-erlangen.de/Lehre/RA/WS2006/multiprozessoren.pdf (Folie 4)
[19] https://www.rz.uni-karlsruhe.de/rz/hw/sp/online-kurs/PARALLELRECHNER/node7.html
[20] http://www.informatik.uni-augsburg.de/lehrstuehle/sik/lehre/ss/sysinfo/folien/SN08-VL10-11-Multiprozessoren.pdf (Folie 5)
[21] Moderne Betriebssysteme, Andrew S. Tanenbaum, Pearson Studium; Auflage: 2., überarb. A. (15. Mai 2002), Seite 550 (Master- Slave-Multiprozessoren)
[22] Moderne Betriebssysteme, Andrew S. Tanenbaum, Pearson Studium; Auflage: 2., überarb. A. (15. Mai 2002), Seite 551 (Symmetrische Multiprozessoren)
[23] http://www.iam.unibe.ch/~rvs/teaching/SS02/bs/arbeiten/Mauricio_Seeberger-MultiprozessorScheduling.pdf (4.1 Test and Set Lock)
[24] http://www.iam.unibe.ch/~rvs/teaching/SS02/bs/arbeiten/Mauricio_Seeberger-MultiprozessorScheduling.pdf (4.2 Spinning)
[25] Moderne Betriebssysteme, Andrew S. Tanenbaum, Pearson Studium; Auflage: 2., überarb. A. (15. Mai 2002), Seite 555/ 556 (Multiporzessorsynchronisation)
[26] http://www.inf.fu-berlin.de/lehre/SS00/19540-V/bookupdate/kahmann_kretzschmar/amdahl.html (Der Ursprung des Amdahl´schen Gesetzes)
[27] http://www.inf.fu-berlin.de/lehre/SS00/19540-V/bookupdate/kahmann_kretzschmar/amdahl.html (Wie entsteht das Problem der geringen Leistungssteigerung?)
[28] http://www.numa.uni-linz.ac.at/Staff/haase/parvor/node36.html (Amdahlsches Gesetz)
[29] http://www.inf.fu-berlin.de/lehre/SS00/19540-V/bookupdate/kahmann_kretzschmar/amdahl.html (Die Aussage des Amdahlschen Gesetzes)
[30] http://tams-www.informatik.uni-hamburg.de/lehre/2000ws/proseminar/mikroprozessoren/09-multiprozessor.pdf (Folie 2-4)
[31] Moderne Betriebssysteme, Andrew S. Tanenbaum, Pearson Studium; Auflage: 2., überarb. A. (15. Mai 2002), Seite 559 (Timesharing)
[32] http://www.iam.unibe.ch/~rvs/teaching/SS02/bs/arbeiten/Mauricio_Seeberger-MultiprozessorScheduling.pdf (5.1.2 Smart- Scheduling)
[33] Moderne Betriebssysteme, Andrew S. Tanenbaum, Pearson Studium; Auflage: 2., überarb. A. (15. Mai 2002), Seite 560 (Timesharing)
[34] Moderne Betriebssysteme, Andrew S. Tanenbaum, Pearson Studium; Auflage: 2., überarb. A. (15. Mai 2002), Seite 560 (Space- Sharing)
[35] http://www.iam.unibe.ch/~rvs/teaching/SS02/bs/arbeiten/Mauricio_Seeberger-MultiprozessorScheduling.pdf (5.2.1 Statische Partitionierung)
[36] Moderne Betriebssysteme, Andrew S. Tanenbaum, Pearson Studium; Auflage: 2., überarb. A. (15. Mai 2002), Seite 561 (Space- Sharing)
[37] Moderne Betriebssysteme, Andrew S. Tanenbaum, Pearson Studium; Auflage: 2., überarb. A. (15. Mai 2002), Seite 563 (Gang- Sharing)