# Summary

## Kurs 1727
# Parallel Programming

**Zusammenfassung der kursrelevante Kapitel aus dem Buch „Parallel Programming“ von Wilkinson/Allen – 2nd Edition**

März 2006

# Inhaltsverzeichnis

(es wurde die Kapitelnummerierung des Buches übernommen)

**1. EINFÜHRUNG PARALLELE COMPUTER ..... 4**
**1.1 Bedarf an mehr Rechengeschwindigkeit ..... 4**
**1.2 Potential für eine schnellere Berechnung ..... 4**
1.2.1 Speedup Faktor ..... 4
1.2.2 Wie groß ist der max. Speedup? ..... 5
1.2.3 Message-Passing Computations ..... 6
**1.3 Arten paralleler Computer ..... 6**
1.3.1 Shared-Memory Multiprozessor System ..... 6
1.3.2 Message-Passing Multicomputer ..... 7
1.3.3 Distributed Shared Memory ..... 9
1.3.4 MIMD und SIMD Klassifizierung ..... 10
**1.4 Cluster Computing ..... 10**
1.4.1 Vernetzte Computer als Computing Plattform ..... 10
1.4.2 Cluster Konfigurationen ..... 11
1.4.3 Aufsetzen eines dedizierten Beowulf-Cluster ..... 12

**2. MESSAGE-PASSING COMPUTING ..... 13**
**2.1 Grundlagen der Message-Passing Programmierung ..... 13**
2.1.1 Möglichkeiten zur Programmierung ..... 13
2.1.2 Prozess Erzeugung ..... 13
2.1.3 Message-Passing Routinen ..... 14
**2.2 Verwendung von Computer-Cluster ..... 15**
2.2.1 Software Tools ..... 15
**2.3 Evaluierung paralleler Programme ..... 15**
2.3.1 Gleichungen für die Ermittlung der parallelen Ausführungszeit ..... 15
2.3.2 Zeitkomplexität ..... 16
*Formale Definitionen: (Details siehe Kurs „Theoretische Grundlagen der Informatik“)* ..... 16
2.3.3 Anmerkungen zur asymptotischen Analyse ..... 17
2.3.4 Kommunikationszeit für Broadcast/Gather ..... 17
**2.4 Debugging und empirische Evaluierung paralleler Programme ..... 18**
2.4.1 Low-Level Debugging ..... 18
2.4.2 Visualisierungs-Tools ..... 18
2.4.3 Debugging Strategien: ..... 18
2.4.4 Programmevaluierung ..... 18
2.4.5 Anmerkungen zur Optimierung paralleler Programme ..... 18

**3. EMBARRASSINGLY PARALLEL COMPUTATIONS ..... 19**
**3.1 Ideale parallele Berechnung ..... 19**
**3.2 Beispiele für Embarrassingly Parallel ..... 19**
3.2.1 Geometrische Transformationen von Bildern ..... 19
3.2.2 Mandelbrot Set ..... 20
3.2.3 Monte Carlo Methoden ..... 21

**4. PARTITIONING UND DIVIDE-AND-CONQUER STRATEGIEN ..... 22**
**4.1 Partitioning ..... 22**
4.1.1 Partitioning Strategien ..... 22
4.1.2 Divide and Conquer ..... 23
4.1.3 M-ärer Divide-and-Conquer ..... 24
**4.2 Beispiele für Partitioning und Divide-and-Conquer ..... 25**

4.2.1 Sortierung mittels Bucket Sort ..... 25
4.2.2 Numerisches Integrieren ..... 26
4.2.3 N-Body Problem ..... 27

**8. PROGRAMMIERUNG MIT SHARED MEMORY ..... 29**
**8.1 Shared Memory Multiprozessoren ..... 29**
**8.2 Konstrukte zur Spezifizierung von Parallelität ..... 29**
8.2.1 Erzeugung paralleler Prozesse ..... 29
8.2.2 Thread ..... 29
**8.3 Sharing Data ..... 31**
8.3.1 Erzeugung von Shared Data ..... 31
8.3.2 Zugriff auf Shared Data ..... 31
**8.4 Parallele Programmiersprachen und Konstrukte ..... 32**
8.4.1 Sprachen ..... 32
8.4.2 Sprachkonstrukte ..... 33
8.4.3 Analyse der Abhängigkeiten ..... 33
**8.6 Performance Issues ..... 34**
8.6.1 Zugriff auf shared Daten ..... 34

**10. SORTIER-ALGORITHMEN ..... 34**
**10.1 Allgemeines ..... 34**
10.1.1 Sortierung ..... 34
10.1.2 Speedup Potential ..... 35
**10.2 Compare-and-Exchange Sortier-Algorithmen ..... 35**
10.2.1 Compare and Exchange ..... 35
10.2.2 Bubble Sort und Odd-Even Transposition Sort ..... 35
10.2.3 Mergesort ..... 36
10.2.4 Quicksort ..... 37
10.2.5 Odd-Even Mergesort ..... 38
10.2.6 Bitonic Mergesort ..... 38
**10.3 Sortieren auf speziellen Netzwerken ..... 39**
10.3.1 2D-Sortieren ..... 39
10.3.2 Quicksort auf einem Hypercube ..... 40
**10.4 Andere Sortieralgorithmen ..... 41**
10.4.1 Rank Sort ..... 41
10.4.5 Implementierung von Sortieralgorithmen auf einem Cluster ..... 42

**11. NUMERISCHE ALGORITHMEN ..... 42**
**11.2 Implementieren der Matrix Multiplikation ..... 42**
11.2.1 Algorithmus ..... 42
11.2.2 Direkte Implementierung ..... 43
11.2.3 Rekursive Implementierung ..... 44
11.2.4 Mesh Implementierung ..... 44
11.2.5 Andere Methoden zur Matrix Multiplikation ..... 46
**11.3 Lösen eines linearen Gleichungssystems ..... 46**
11.3.1 Lineare Gleichungen ..... 46
11.3.2 Gauss'sche Eliminierung ..... 46
11.3.3 Parallele Implementierung ..... 47
**11.4 Iterative Methoden ..... 48**

# 1. Einführung Parallele Computer

## 1.1 Bedarf an mehr Rechengeschwindigkeit

Es besteht ein kontinuierlicher Bedarf an immer größerer Rechenleistung, um

- Probleme schneller und genauer zu lösen
- und um auch größere Instanzen eines Problems lösen zu können,

beides in Bezug auf Verarbeitungs-Ressourcen und Speicher-Ressourcen. Beispiele dafür sind vor allem die Bereiche der Wettervorhersage und Klimaforschung ebenso wie in der Astronomie.

## 1.2 Potential für eine schnellere Berechnung

### 1.2.1 Speedup Faktor

Der Speedup Faktor ist eine Maßzahl für die relative Änderung der Performance bei Verwendung eines parallelen Systems für die Lösung eines Problems anstatt eines Ein-Prozessor Systems.

$$S(p) = \frac{Ausführungszeit\ auf\ Single\,\mathrm{Pr}ozessor\ System}{Ausführungszeit\ auf\ Multi\,\mathrm{Pr}ozessor\ System\ mit\ p\,\mathrm{Pr}ozessoren}$$

also

$$S(p) = \frac{t_S}{t_P}$$

mit

$t_S$ ... als Ausführungszeit des besten sequentiellen Algorithmus auf Single-Prozessor
$t_P$ ... als Ausführungszeit für die Lösung der selben Aufgabe auf einem Multi-Prozessor

Der **max. Speedup** $p$ bei $p$ Prozessoren (*linearer Speedup*) ist nur erreichbar, wenn

- die Aufgabe in gleich lang dauernde Prozesse unterteilbar ist,
- mit einem Prozessor je Prozess und
- kein zus. Overhead für die Parallele Lösung entsteht *(Kommunikation)*.

$$S(p) \le \frac{t_S}{t_S / p} = p$$

Ein eventueller *superlineare Speedup* mit $S(p) > p$ ist meist zurückzuführen auf

- einen suboptimalen sequentiellen Algorithmus
- ein spez. Feature der System-Architektur, das parallele Prozesse begünstigt
- der wesentlich größere verfügbare Speicher eines parallelen Systems.

Ein superlinearer Speedup in mehreren Größenordnungen ist aber durchaus bei Suchalgorithmen erreichbar, wenn der Suchraum auf parallele Prozessoren aufgeteilt wird. Im worst case muss der sequentielle Algorithmus den gesamten Suchraum abarbeiten, die parallel Prozesse aber nur jenen Bruchteil, den sie zugewiesen bekommen haben.

***Effizienz***:
Ist eine Maßzahl für die Auslastung der Prozessoren

$$E = \frac{Ausführungszeit\ mit\ einem\,\mathrm{Pr}ozessor}{Ausführungszeit\ mit\ Multiprozessorsystem\ \times Anzahl\ der\,\mathrm{Pr}ozessoren} = \frac{t_S}{t_P \times p}$$

damit folgt $E = \frac{S(p)}{p} \times 100\%$ für $E$ in %. *(E=100% wenn alle Prozessoren voll ausgelastet sind und S(p)=p).*

### 1.2.2 Wie groß ist der max. Speedup?

Faktoren, die in der parallelen Ausführung den Speedup begrenzen sind

1. Idle-Zeiten der beteiligten Prozessoren
2. Zusätzlicher Ausführungscode der parallelen Version *(z.B. Konstante lokal neu berechnen)*
3. Kommunikationszeit zwischen den Prozessen

Außerdem muss davon ausgegangen werden, dass nicht der gesamte Code parallelisierbar ist, sondern immer bestimmte Teile *(z.B. Initialisierungen)* nur sequentiell ausgeführt werden können. Bezeichnet man diesen sequentiellen Teil als *f* so ergibt sich für die gesamte Ausführungszeit mit *p* Prozessoren

$$t_G = ft_s + (1-f)t_S / p$$

und somit für den Speedup

$$S(p) = \frac{t_S}{ft_S + (1-f)t_S / p} = \frac{p}{1+(p-1)f} \qquad \textit{[Amdahl's Gesetz]}$$

woraus folgt, dass der max. Speedup allein begrenzt ist durch den seriellen Anteil *f*, also

$$\lim_{p\to\infty} S(p) = \frac{1}{f}$$

***Skalierbarkeit***:

Diese beschreibt einerseits ein HW-Design das es erlaubt, das System zu vergrößern und so die Performance zu erhöhen -> *Architektur-Skalierbarkeit*.

Andererseits beschreibt sie in Zusammenhang mit parallelen Algorithmen die Eigenschaft, steigende Datenmengen mit nur sehr geringer und begrenzter Steigerung der Abarbeitungsschritte zu bewältigen -> *algorithmische Skalierbarkeit*.

Eine kombinierte Architektur- bzw. algorithmische Skalierbarkeit würde also bedeuten, dass für eine spezielle Architektur und Algorithmus eine Steigerung der Aufgabengröße mit einer Vergrößerung des Systems *(also mehr Prozessoren)* bewältigt werden kann. Wobei eine Steigerung der Aufgabengröße als Vergrößerung der Eingangsdatenmenge verstanden wird und nicht zwingend einen linearen Anstieg der notwendigen Abarbeitungsschritte bedingt *(siehe z.B. Matrizen-Addition -> linear und Matrizen-Multiplikation - quadratisch)*. Üblicherweise verbessert aber eine Steigerung der Aufgabengröße die relative Performance des Systems, da eine höhere Parallelität erreicht werden kann.

*Gustafson's* Gesetz basiert auf diesem Skalierbarkeits-Konzept und geht im Gegensatz zu *Amdahl* davon aus, dass nicht der sequentielle Anteil einer Aufgabe konstant ist sondern die parallele Ausführungszeit. Dies begründet sich darin, dass in der Praxis die ausgewählte Aufgabengröße für ein Multiprozessor-System sich an der Anzahl der verfügbaren Prozessoren orientiert *(also große MPS werden mit großen Aufgaben gefüttert)*. Dies ist genauso plausibel wie Amdahls Annahme eines konstanten sequentiellen Anteils einer Aufgabe unabhängig von deren Größe.

Gustafson's Speedup wird als „*skalierter" Speedup* bezeichnet und ergibt mit $t_P = ft_S + (1-f)t_S / p = 1$ und der Aufteilung von $t_S$ in einen konstanten und von *p* unabhängigen Teil $ft_S$ und einen dynamischen Teil $(1\text{-}f)t_S$ für

$$S_S(p) = \frac{ft_s + (1-f)t_S}{ft_S + (1-f)t_S / p} = p + (1-p)ft_S \qquad \textit{[Gustafson's Gesetz]}$$

Neben der Skalierung mit konstanter Aufgabengröße und jener mit konstanter paralleler Ausführungszeit gibt es noch jene mit begrenztem Speicher. Dabei wird die Aufgabe so skaliert, dass sie in den verfügbaren Speicher passt. Wird die Anzahl der Prozessoren erhöht steigt damit auch der verfügbare Speicher. Dies kann zu einer signifikanten Erhöhung der

Ausführungszeit führen, da ja auch die Aufgabengröße wieder an den größeren Speicher angepasst wird.

### 1.2.3 Message-Passing Computations

Der Nachrichtenaustausch zwischen den Prozessen kann zu einem signifikanten Overhead bei parallelen Programmen mit Message-Passing führen. Die parallele Ausführungszeit setzt sich dann aus einem Kommunikationsteil und einem Berechnungsteil zusammen:

$$t_P = t_{comm} + t_{comp}$$

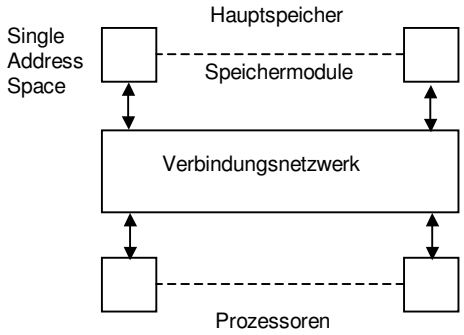
Je granularer eine Aufgabe in parallele Prozesse aufgeteilt wird, desto stärker beeinflusst die Kommunikationszeit die gesamte parallele Ausführungszeit. Als Metrik dient dabei die

$$Computation/communication\ ratio = \frac{t_{comp}}{t_{comm}} \quad .$$

## 1.3 Arten paralleler Computer

### 1.3.1 Shared-Memory Multiprozessor System

Alle Prozessoren des MPS habe Zugriff auf alle Speichermodule innerhalb des Systems. Der gesamte Shared-Memory wird in einem einzigen Adressraum abgebildet. Der Zugriff erfolgt normalerweise für den Programmierer transparent über irgendeine Form eines Verbindungsnetzwerkes. Das ausführbare Programm und die Daten werden für die Prozessoren im Shared-Memory abgelegt.




Für die Programmierung von Shared-Memory MPS gibt es spezielle Spracherweiterung für bestehenden Programmiersprachen wie z.B. OpenMP für C/C++ und Fortran oder UPC(Unified Parallel C). Sie bieten spezielle Sprachelemente für z.B. die Deklaration von Shared-Variablen oder paralleler Codeabschnitte.

Bei kleineren Shared-Memory MPS mit 2 bis 4 Prozessoren ist der gemeinsame Speicherzugriff meist noch relativ einfach über einen Bus realisierbar und die Zugriffszeit für alle Prozessoren gleich (UMA – uniform memory access).

Für größere MPS mit Shared-Memory ist dies allerdings nur mehr sehr schwer realisierbar und es wird meist eine hierarchische oder verteilte Speicherstruktur verwendet. Allerdings kann damit kein gleichförmiger Speicherzugriff mehr ermöglicht werden, sondern nahe liegender Speicher ist für einen Prozessor schneller erreichbar als entfernter (NUMA – non uniform memory access). Das Problem der unterschiedlichen Speicherzugriffzeiten in einem NUMA Design kann zwar durch den lokalen Cache der Prozessoren etwas entschärft werden, allerdings kann in diesem Fall die Wahrung der Datenkonsistenz eine recht komplexe Aufgabe sein.

## 1.3.2 Message-Passing Multicomputer

Als Alternative zu Shared-Memory MPS kann ein paralleles Mehrprozessorsystem auch aus mehreren Computern aufgebaut sein, dass über ein Netzwerk untereinander verbunden ist. Jeder Computer bzw. Prozessor hat seinen eigenen Speicher, der aber von anderen Prozessoren nicht zugreifbar ist. Über ein Verbindungsnetzwerk können Nachrichten bzw. Daten zwischen den Prozessoren od. Prozessen ausgetauscht werden.

Bei der Programmierung wird die Aufgabe ebenfalls in parallel bearbeitbare Abschnitte unterteilt die dann auf parallel laufende Prozesse verteilt werden. Es können dafür spezielle parallele oder erweiterte sequentielle Programmiersprachen verwendet werden. Ein gebräuchlicher Ansatz ist allerdings der Einsatz von speziellen Message-Passing Library Routinen zusammen mit konventionellen sequentiellen Programmiersprachen. Die Kommunikation zwischen den Prozessen erfolgt dann ausschließlich über den Versand von Nachrichten.

### *Netzwerke für Multicomputer:*

Die wichtigsten Parameter für das Netzwerkdesign sind

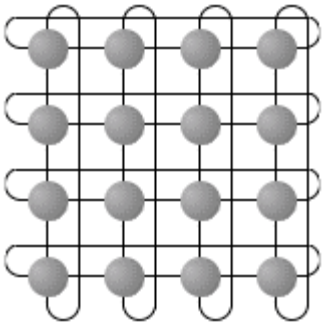
- *Bandbreite* – also die Übertragungsgeschwindigkeit (bit/s)
- *Network latency* - also die Zeit für den Nachrichtentransport durch das Netzwerk.
- *Communication latency* – die gesamte Zeit für das Versenden einer Nachricht inklusive SW-Overhead (Protokollstack u. OS-Aufrufe) und des Interface-Delays.
- *Message latency* od. startup time – ist die Zeit für das Versenden einer Nachricht ohne Nutzdaten, also praktisch nur der SW- und HW-Overhead (routing, packing, unpacking).
- *Kosten*
- *Komplexität*

Die Anzahl der phys. Links zwischen 2 Knoten ist ein wichtiger Faktor zur Bestimmung der maximalen Verzögerung einer Nachricht auf dem Weg von Knoten A nach B. Der *Durchmesser* eines Netzwerks beschreibt die minimale Anzahl solcher Links zwischen den beiden am weitesten auseinander liegenden Knoten, also die maximale Distanz, die eine Nachricht zurücklegen muss, wenn nur die kürzesten Routen betrachtet werden.

Die *bisection width* beschreibt die minimale Anzahl von Links die durchtrennt werden müssen, um ein Netzwerk in 2 gleiche Hälften zu teilen, also die minimale Anzahl von Verbindungen zwischen 2 Netzwerkhälften. Die *bisection bandwith* ist die kollektive Bandbreite über diese Links für die Übertragung von Nachrichten zwischen den beiden Netzwerkhälften.

Da bei der Vernetzung der Knoten bei größeren Systemen *(ab ca. 8 Knoten)* eine direkte Verbindung eines Knoten mit jedem anderen Knoten schnell nicht mehr sinnvoll durchführbar ist *(bei c Knoten wären c(c-1)/2 Links notwendig)*, werden Topologien mit eingeschränkter Verbindungsstruktur eingesetzt, wovon sich 2 Typen etabliert haben.

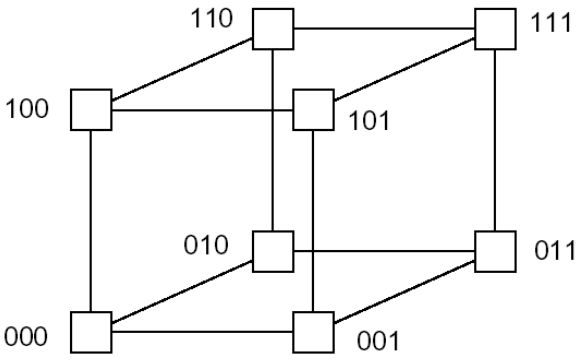
**Mesh**: Die Knoten werden in einem 2-dimensionalen Netz (Gitter) angeordnet, wo jeder Knoten mit seinen 4 Nachbarknoten verbunden ist. Der Durchmesser dieses Netzes beträgt bei p Knoten $2(\sqrt{p}-1)$. Werden die freien Enden des Netzes noch jeweils mit der gegenüberliegenden Seite verbunden so nennt man dieses Netz *Torus*. Der Durchmesser reduziert sich dann auf $\sqrt{p}$. Bei einer 3-dimensionalen Variante ist jeder Knoten mit 2 Knoten in der X-, Y- und Z-Ebene



verbunden. Mesh haben den Vorteil des rel. einfachen Layouts und der leichten Erweiterbarkeit.

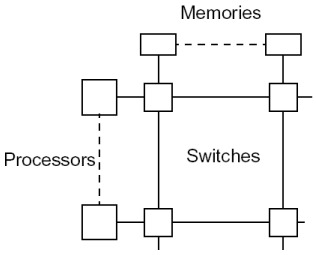
**Hypercube Netzwerk**: In einem *d*-dimensionalen (binären) Hypercube Netzwerk ist jeder Knoten mit einem Knoten in jeder Dimension des Netzwerks verbunden. Dabei erhält jeder Knoten eine *d*-bit Binäradresse, wobei jedes Bit eine Dimension repräsentiert. Jeder Knoten wird mit jenen Knoten verbunden, deren Adresse sich nur in einem Bit von der eigenen Adresse unterscheiden.

Der Durchmesser des Hypercupe beträgt $\log_2 p$ bei $p$ Knoten, also nur ein logarithmisches Wachstum bei Erweiterung des Netzes. Außerdem existiert ein „*minimal distance deadlock-free*" Routing-Algorithmus, der sog. *e-cube* oder *left-to-right* Algorithmus. Es wird dabei die Adresse des Start- und des Zielknoten XOR verknüpft und die 1-Bits des Ergebnis geben an, welche Dimensionen für das Routing verwendet werden sollen. In jedem Zwischenknoten wird dann erneut die aktuelle Knotenadresse mit der Zielknotenadresse XOR verknüpft und so weiter fortgefahren.



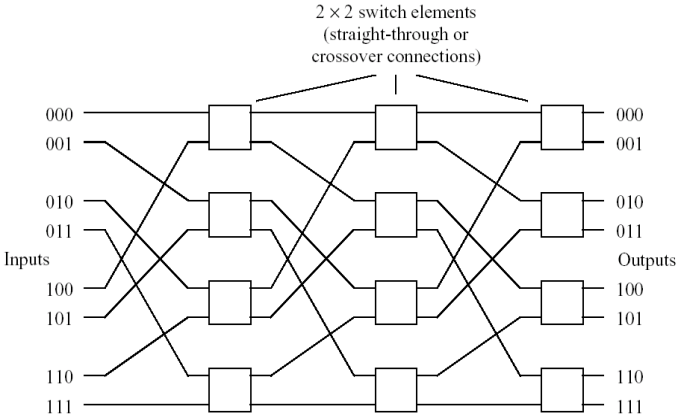

Ein *d*-dimensionaler Hypercube besteht im Prinzip aus zwei *d-1* dimensionalen Hypercubes. Die bisection width ist *p/2* bei *p* Knoten. Hypercubes werden heute aber kaum mehr eingesetzt.

**Crossbar Switch**: Für jede Verbindung wird ein Switch verwendet. Eingesetzt werden sie hauptsächlich in Shared-Memory Systemen für die Verbindung der Speichermodule mit den Prozessoren.




**Tree Networks**: Netzwerke mit Baumstruktur sind eine alternative Switch-Konfiguration. Nur die Blätter des Baums sind Prozessoren bzw. Computer, alle anderen Knoten sind Switches. Der Nachteil dieser Struktur ist, dass sich der Netzwerkverkehr zur Wurzel hin steigert, was ein deutlicher Flaschenhals für der Kommunikation bedeuten kann. Der Vorteil hingegen ist die Höhe des Baumes nur logarithmisch mit der Anzahl der Prozessor-Knoten steigt also $\log_2 p$ bei $p$ Prozessoren und einem binären Baum. Die Topologie ist aber nicht auf binäre Bäume beschränkt, ebenso wenig muss der Baum komplett sein (jede Ebene ist voll). *(Höhe eines k-ären Baums ist $\log_k p$ bei p Blättern im vollständigen Baum)*

**Multistage Interconnection Networks**: MIN sind Netzwerkklassen, die eine bestimmte Anzahl von Switch-Ebenen haben, wobei diese Ebenen mit ihren benachbarten Ebenen über verschiedene symmetrische Wege verbunden sein können. Ein Beispiel dafür ist das Omega-Netzwerk. Allerdings ist das Omega-Netzwerk stark blockierend, es gibt aber immer eine Pfad zwischen jedem Eingang und jeden Ausgang in einem freien Netzwerk.



***Kommunikationsmethoden***:
Es gibt zwei grundlegende Methoden für den Nachrichtentransfer zwischen Quelle u. Ziel:

**Circuit Switching**: Der gesamte Verbindungspfad wird für die Dauer des Nachrichtentransfers exklusiv reserviert *(wie beim analogen Telefon)*. -> wird wegen Ineffizienz nicht mehr eingesetzt.

**Packet Switching**: siehe Netzwerktechnik. Die Pakete müssen in den einzelnen Knoten (z.B. Router) zwischengespeichert werden, bis die entsprechende ausgehenden Verbindung frei ist. Dafür gibt es mehrere Ansätze:

- *store-and-forward packet switching*: klassischer Ansatz. Paket wird solange im Knoten zwischengespeichert bis die ausgehende Verbindung frei für den Weiterversand ist. -> *Latenzzeit wächst proportional mit der Anzahl der Zwischenknoten im Verbindungspfad.*
- *cut-through*: Im Gegensatz zu store-and-forward wird das Paket nur gepuffert, wenn keine freie abgehende Verbindung verfügbar ist. Also im Idealfall wird das Paket in jedem Knoten sofort weitergeleitet.
- *wormhole routing*: Die Nachricht wird in kleinere Einheiten, sog. *flits* (flow control digits) unterteilt mit einer Größer von 1 bis 2 Bytes. Nachdem der Nachrichten-Header übertragen wurde werden die nachfolgenden flits übertragen, sobald ein Link verfügbar ist. *(Parallelübertragung möglich, wenn eine Leitung pro flit-Bit vorhanden.)* Über eine request/acknowledge Signalisierung zwischen den Knoten werden die flits von Knoten zu Knoten übertragen.
    - Vorteil: geringerer Bufferbedarf in den Knoten und eine konstante Latenzzeit unabhängig von der Anzahl der Knoten auf dem Verbindungspfad *(falls die flits sehr viel kleiner als die Gesamtnachricht sind)*.
    - Nachteil: Die Verbindung muss für das wormhole reserviert sein, da eine Vermischung mit anderen Paketen nicht möglich ist.

**Locking**: Je nach Qualität des Routingalgorithmus und der verwendeten Verbindungsmethode besteht eine mehr od. weniger große Gefahr, dass es zu einer Blockierung der Nachrichtenübertragung kommt.

- *Livelock*: kann speziell bei Verwendung adaptiver Routingalgorithmen auftreten und bezeichnet die Situation, wenn ein Packet im Netz „umherirrt“ ohne jemals den Zielknoten zu finden.
- *Deadlock*: Speziell das *store-and-forward* und das *wormhole-routing* sind dafür anfällig. Ein *Deadlock* tritt ein wenn ein Paket nicht an den nächsten Knoten übertragen werden kann, weil dessen Buffer voll ist und dieser aber nicht übertragen kann weil der Buffer des nächsten Knoten auch voll ist usw.

Ein allgemeiner Lösungsansatz für das *Deadlock*-Problem ist die Verwendung von *virtual channels*. Jeder virtual channel besitzt einen separaten Buffer und mehrere solcher Kanäle sind dann einem physikalischen Kanal zugeordnet. Sie verwenden den physikalischen Kanal in einem time-multiplexed Verfahren. Damit ist bei einem Deadlock eben nur ein virtual channel blockiert, nicht aber der physikalische Kanal.

### 1.3.3 Distributed Shared Memory

Das Message-Passing Paradigma hat den Nachteil, dass bei der Programmierung explizit Message-Passing Aufrufe eingebaut werden müssen, was das Ganze fehleranfällig und schwer zum Debuggen macht. Allerdings hat es den Vorteil, dass keine speziellen Synchronisierungsmechanismen für den simultanen Zugriff auf Daten benötigt werden.

Trotzdem (oder gerade deshalb) ist der Shared-Memory Ansatz für die Programmentwicklung attraktiver. Eine Erweiterung stellt das Konzept des Distributed-Shared-Memory dar, also der Speicher ist physikalisch verteilt auf verschiedene Prozessoren bzw. Computer. Jeder Prozessor hat aber Zugriff auf den gesamten verteilten Speicher über einen gemeinsamen Adressraum der vom Virtual Memory Management System verwaltet wird (-> *virtual shared memory*). Physikalisch ist das System aufgebaut wie ein Message-Passing Multi-Computer System, nur dass hier der lokale Speicher eines jeden Computers Teil des Shared-Memory ist. Für den Programmierer ist das Message-Passing aber transparent und das System präsentiert sich wie ein Shared-Memory Multiprozessor System.

### 1.3.4 MIMD und SIMD Klassifizierung

Nach Flynn(1996) gibt es folgende Klassifizierungen von Computersystemen:

- *SISD – Single Instruction Stream Single Data Stream*: Damit sind die klassischen Single-Prozessor Computer gemeint, wie wir sie normalerweise als Desktop-Computer kennen.
- *MIMD – Multiple Instruction Stream Multiple Data Stream*: Allgemeines Multiprozessor System, jeder Prozessor hat ein separates Programm und arbeitet auf separaten Daten.
- *SIMD – Single Instruction Stream Multiple Data Stream*: Es gibt nur ein Programm aber mehrere Data-Streams. D.h. die Anweisungsstrom aus einem Programm wird an mehrere Prozessoren verteilt und diese arbeiten damit aber dann auf verschiedenen Daten. Typischerweise sind die einzelnen Prozessoren Arithmetik-Porzessoren ohne *program-control* unit sondern nur mit einer einzigen *instruction control unit* für das Holen der einzelnen Befehle aus dem Speicher. Alle Prozessoren arbeiten die Befehle synchron ab, allerdings auf unterschiedlichen Daten. SIMD-Systeme arbeiten meistens auf Datenfeldern in Bereichen wie der Simulation phys. Systeme oder der Low-Level Bildverarbeitung (z.B. Feld- od. Vektorrechner). Darüber hinaus haben z.B. Prozessoren der Pentium-Familie ab Pentium II ebenfalls SIMD-Befehle für Multimedia implementiert (MMX-Befehle).
- *MISD – Multiple Instruction Stream Single Data Stream*: Existiert in dieser Form nicht, außer man betrachtet Pipelined-Architekturen od. spezielle fehlertolerante Systeme als solche.

Weitere Klassifizierungen *(nicht nach Flynn)* wären

- *MPMD – Multiple Program Multiple Data*: Jeder Prozessor hat sein eigenes Programm und arbeitet auf eigenen Daten. Typischerweise ist es aber meist so, das es in einer Master-Slave Konstellation 2 verschiedene Programme gibt, jenes für den Master und jenes für einen od. mehrere Slaves *(siehe auch einige EA's zu diesem Kurs).*
- *SPMD – Single Program Multiple Data*: Es gibt nur ein Programm für alle Prozessoren, diese bearbeiten damit aber verschiedene Daten und zwar asynchron zueinander. Die Programme können natürlich so ausgebildet sein, das bestimmte Programmteile nur von bestimmten Prozessoren *(abhängig von ihrer ID wie z.B. in einigen EA's Master und Slaves)* abgearbeitet werden. Die Prozessoren arbeiten aber unabhängig voneinander und nicht synchron.

## 1.4 Cluster Computing

### 1.4.1 Vernetzte Computer als Computing Plattform

Die Verwendung von vernetzten Workstations als eine virtuelle Computing Plattform bietet einige signifikante Vorteile gegenüber speziell entworfenen parallelen Computer Systemen oder Supercomputern:

1. Workstations und PC's mit sehr hoher Leistungsfähigkeit sind bereits zur äußerst niedrigen Kosten erhältlich.
2. Die neuesten Prozessortechnologien können mit wenig Aufwand in das Gesamtsystem integriert werden und das System kann auch einfach durch Hinzufügen weiterer Rechen- bzw. Speicherressourcen erweitert werden.
3. Existierende Applikationen können direkt oder entsprechend modifiziert verwendet werden.

Für die einfache kollektive Nutzung der Workstations wurden spezielle Tools zur parallelen Programmierung entwickelt. Das wichtigste Message-Passing Projekt war hierbei PVM (Parallel Virtual Machine -> siehe FEU Cluster), später folgte dann die Standard Message-Passing Library MPI (Message Passing Interface).

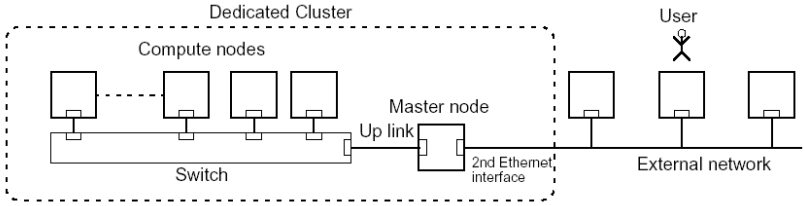
***Ethernet Verbindungen***:
Die gebräuchlichste Kommunikations-Methode für vernetzte Computer ist Ethernet. Ursprünglich als klassischer Bus konzipiert wird heute eigentlich nur mehr Switched-Ethernet eingesetzt, wo praktisch zwischen zwei Knoten immer eine P2P-Verbindung aufgebaut wird (außer bei Multicast- oder Broadcastnachrichten). Durch Verwendung von Switches wird eine simultane Verbindung mehrerer Knotenpaare ermöglicht und das nicht-deterministische Verhalten von Ethernet aufgrund des Kollisionsauflösungsprotokolls (CSMA/CD) stark entschärft. Die Message-Latenz ist bei Ethernet aber immer noch signifikant (Protokollstack, Queueing in den Switches) besonders wenn man den zusätzlichen Overhead durch die Message-Passing SW mit berücksichtigt. *(Details siehe Netzwerktechnik)*

***Netzwerk Adressierung***:
Es wird standardmäßig auf das TCP/IP Protokoll aufgesetzt *(Details siehe Netzwerktechnik)*.

## 1.4.2 Cluster Konfigurationen

***Verwendung existierender vernetzter Computer:***
Mit den heute verfügbaren modernen Betriebssystemen zusammen mit einer Message-Passing Software wie PVM ist es relativ einfach, ein bestehendes Computernetzwerk als Cluster zu verwenden währenddem auf den betroffenen Stationen weiter wie vorher gearbeitet wird. Es ergeben sich dabei allerdings einige praktische Probleme durch die „Mehrfachnutzung" als Cluster und Einzelstation. Ein User könnte einen der Cluster-Computer abschalten oder zum Absturz bringen (Entwickler) während dem der parallele Prozess im Laufen ist. Dies ist für einen professionellen Betrieb nicht akzeptable.

***Dedizierte Computer Cluster:***
Damit wird das Problem mit den lokalen Usern umgangen. Der Zugriff auf die Cluster-Computer erfolgt über remote-Zugriffe.



***Beowulf Cluster:***

Damit werden kosteneffektive Cluster bezeichnet die nur aus weit verbreiteten low-cost Komponenten bestehen und keine Spezialkomponenten verwenden. Also z.B. Standard PC's mit Linux als OS und Standard Ethernet als Netzwerk. Es wird auch als ein Cluster aus „commodity" (Gebrauchsartikel) Computer bezeichnet.

***Jenseits von Beowulf:***

**Netzwerke**: Neben Standard Ethernet und mittlerweile Gigabit-Ethernet wie sie in Beowulf-Cluster eingesetzt werden gibt es noch einige spezialisierte High-Performance Netzwerkverbindungsarten wie

- Myrinet, mit 2.4Gbit/s
- cLan
- SCI (Scalable Coherent Interface)
- QsNet
- Infiniband

**Cluster mit Mehrfachverbindungen**: Die Verwendung mehrerer paralleler Verbindungen soll den Kommunikations-Overhead reduzieren. Es gibt dafür mehrere Ansätze:

- Beowulf verwendet 2 Standard Ethernet Controller pro Computer und betreibt diese mit einer „channel bonding" Technik, d.h. ein virtueller Kanal ist mehreren physikalischen Kanälen zugeordnet.
- Verwendung einer Standard-Ethernet Verbindung als Setup-Kanal und eine Myrinet Verbindung für den parallelen Betrieb.
- Verwendung mehrerer Ethernet-Adapter zum Betrieb mehrerer paralleler Verbindungen.
- Verwendung spezieller Netzwerkadapter die mehrere parallele Leitungen betreiben können.
- Überlappende Verbindungsnetze mit Switches realisiert.

**Symmetrical Multiprocessors (SMP) Cluster**: Kleine, um einen Verbindungsbus angeordnete Shared-Memory Multiprozessor Systeme besitzen eine Symmetrie zwischen ihren Prozessoren und Speichermodulen und werden deshalb als SMP's bezeichnet. SMP's auf Basis von Dual-Pentium Computer sind sehr kosteneffizient und deshalb auch hervorragend für Clustering geeignet. Dabei kann zwischen den SMP's über Message-Passing kommuniziert werden und innerhalb der SMP's über Shared-Memory Methoden. Allerdings wird aus Gründen der Einfachheit meist generell Message-Passing eingesetzt, wobei dies innerhalb der SMP's wieder über Shared-Memory-Locations als Nachrichtenträger realisiert werden kann (ist schneller).

**Web Cluster**: Die Idee, „Web"-Computer in Form eines Clusters zu nutzen, wird auch „Grid Computing" genannt. Das erste bekanntere Projekt, das diese Form des Clustering nutzte, war das S.E.T.I. Projekt.

### 1.4.3 Aufsetzen eines dedizierten Beowulf-Cluster

***Hardware Konfiguration:***

Eine gängige Konfiguration ist die Verwendung eines Computers als Master-Knoten und die restlichen als Slave- oder Computing-Knoten *(siehe FEU-Cluster)*. Der Master-Knoten agiert als File-Server mit entsprechend viel Arbeits- und Festplattenspeicher. Als Verbindungsnetzwerk

wird Fast- oder Gbit-Ethernet eingesetzt. Der Master-Knoten hat einen zweiten Ethernet-Adapter zur Verbindung „nach außen“ und eine global erreichbare IP-Adresse. Die Slaves erhalten IP-Adressen, die nur innerhalb des Clusters erreichbar sind. Darüber hinaus kann noch ein Management-Knoten eingesetzt werden, um den Cluster zu überwachen, warten od. zu testen. Die einzelnen Slaves könne über einen separaten Kommunikationsweg (z.B. Serielle) mit dem Management-Knoten verbunden sein.

***Software Konfiguration:***

- Alle Computer im Cluster sind üblicherweise mit dem selbem Betriebssystem (meist Linux) ausgestattet.
- Der Master verwaltet alle Applikationsdateien der Slaves, diese haben über ein Network File System (meist NFS) Zugriff auf die Daten am Master.
- Am Master läuft die Message Passing Software (MPI oder PVM), ein Cluster Management Tool und die parallelen Applikationen.
- Die Message Passing Software ist als Middleware zwischen OS und User angeordnet.

Die Herausforderung dabei ist das erste Aufsetzen des Clusters, was einiges an OS- und Netzwerk-KnowHow erfordert. Es gibt aber mittlerweile spezielle Cluster Setup Packages wie z.B. Oscar ( Open Source Cluster Application Resources -> www.csm.ornl.gov/oscar ) die das Aufsetzen eines Clusters wesentlich vereinfachen.

# 2. Message-Passing Computing

## 2.1 Grundlagen der Message-Passing Programmierung

### 2.1.1 Möglichkeiten zur Programmierung

Die Programmierung eines Message-Passing Multicomputer Systems kann auf verschiedene Wege realisiert werden:

1. Entwurf einer speziellen parallelen Programmiersprache.
2. Erweiterung der Sprachelemente einer bestehenden sequentiellen Hochsprache.
3. Verwendung einer speziellen Library externer Routinen zusammen mit einer existierenden sequentiellen Hochsprache.

Das einzige allgemein bekannte Beispiel zu 1. ist die Sprache „*occam*“ für die Transputer (Message Passing Multiprozessor) der Fa. Inmos aus den späten 80er Jahren. Beispiele für 2. wären High Performance Fortran (HPF) bzw. Fortran M.

Die Programmierung in diesem Kurs verwendet die Methodik nach 3., wobei PVM als Message-Passing Library für die direkte Prozess-Prozess Kommunikation eingesetzt wird.

### 2.1.2 Prozess Erzeugung

Es gibt zwei grundlegende Methoden zur Prozesserzeugung, statische und dynamische.

- *Statische Prozesserzeugung*: Die Anzahl der zu erzeugenden Prozesse ist im Code festgelegt und steht somit bereits beim Programmstart fest.
- *Dynamische Prozesserzeugung*: Es können während der Ausführung von bereits laufenden Prozessen je nach Bedarf neue Prozesse erzeugt werden bzw. nicht mehr benötigte Prozesse beendet werden. Die dynamische Variante ist zwar effizienter allerdings auch mit signifikant mehr Overhead vor allem bei der Prozesserzeugung behaftet. Einfachste Art einer dynamischen Prozesserzeugung wäre z.B. mittels `spawn`-Befehl ein anderes Programm zu starten.

Als Programmiermodelle werden eingesetzt

- *MPMD Multiple Program Multiple Data*: ist das allgemeinste Modell. Für jeden Prozessor wird ein separates Programm entwickelt, wobei es normalerweise ausreicht, ein Master- und ein Slave-Programm zu entwickeln.
- *SPMD Single Program Multiple Data*: besonders geeignet für statische Prozesserzeugung. Ein Programm für alle Prozessoren (Prozesse), wobei über interne Verzweigung unter Verwendung von z.B. der Prozessor-ID Codebereiche den verschiedenen Prozessen zugeteilt werden können. Klassischer Ansatz für das Message Passing System MPI.

## 2.1.3 Message-Passing Routinen

### *Send und Receive Routinen:*

Eine einfache Form könnte folgend aussehen:

```
send(&msg, dest_id);
recv(&msg, src_id);
```

### *Synchrones Message-Passing:*

Darunter versteht man, dass z.B. die `send/recv` Routinen erst von ihrem Aufruf zurückkehren, wenn die versendete Nachricht beim Empfänger angekommen ist bzw. die erwartete Nachricht empfangen und in einem Buffer gespeichert wurde. Synchrone Übertragungs-Routinen haben also zusätzlich noch die Eigenschaft, dass sie die beteiligten Prozesse synchronisieren.

Synchrone `send/recv`-Routinen arbeiten auf Basis von Handshake–Kommunikation wie z.B. 3-Wege Handshake (`requ-ack-msg`) und benötigen deshalb nicht unbedingt einen eigenen Nachrichtenspeicher.

### *Blockierendes und nichtblockierendes Message-Passing:*

Obwohl in vielen Bereichen blockierend und synchron synonym verwendet werden, wird in der MPI Implementierung zwischen den beiden Begriffen unterschieden. Blockierend wird dabei so definiert, dass die Routine zurückkehrt, wenn die prozesslokale Aktion beendet ist (also lokal blockierend). D.h. dass z.B. eine zu versendende Nachricht im Sendebuffer gespeichert ist aber noch nicht an die Gegenstelle versandt wurde. Bei Verwendung der nichtblockierenden Variante muss der Programmierer selbst sicherstellen, dass z.B. nachfolgende Befehle den Sendespeicher des send-Aufrufes nicht vorzeitig verändern.

### *Nachrichten Auswahl:*

Um dem Empfänger eine zusätzliche Möglichkeit der nachrichtenspezifischen Bearbeitung zu bieten, können Nachrichten mittels Message-Tags klassifiziert werden, um im Empfängerprozess spezifische Empfangsroutinen einzusetzen, die nur bestimmte Nachrichtentypen entgegennehmen.

### *Broadcast, Gather und Scatter:*

- *Broadcast*: Eine Nachricht wird an alle Prozesse oder einer Gruppe von Prozessen gesandt. Im `bcast`-Befehl wird ein Root-Prozess definiert, der die Daten an alle anderen identifizierten Prozesse versendet. Da der Broadcast erst ausgeführt wird, wenn alle beteiligten Prozesse den `bcast`-Befehl bearbeitet haben, erreicht man dadurch auch einen Synchronisierungs-Effekt. In manchen Implementierungen (z.B. in MPI) erhält auch der Root-Prozess die Broadcast-Nachricht. Wegen der häufigen

Verwendung von Broadcast-Befehlen ist eine effiziente Implementierung äußerst wichtig.
- *Scatter*: Jedes Element eines Arrays wird an einen anderen Prozess versendet, wobei der Array-Index mit der Prozess-ID der Empfänger korrespondiert (die Daten werden also verteilt). Hier hängt es ebenfalls von der Implementierung ab, ob der Root-Prozess in die Datenverteilung miteingeschlossen ist.
- *Gather*: Als Gegenstück zu Scatter werden damit Daten von verschiedenen Prozessen eingesammelt und in einem Array analog zu Scatter abgelegt.
- *Reduce*: dabei wird die Gather-Operation noch mit einer arithmetischen/logischen Operation verknüpft und z.B. die eingesammelten Daten gleich aufsummiert, bevor sie im Root-Buffer als Ergebnis des Reduce-Kommandos abgelegt werden.

## 2.2 Verwendung von Computer-Cluster

### 2.2.1 Software Tools

- *PVM Parallel Virtual Machine*: Freie SW-Umgebung für heterogene Systeme. Unterstützt dynamische Prozesserzeugung. Library-Routinen für C und Fortran -> siehe PVM-Manual der FEU
- *MPI Message Passing Interface*: definiert Standard aber nicht Implementierung, ebenfalls als Library für C oder Fortran aufgebaut.

## 2.3 Evaluierung paralleler Programme

### 2.3.1 Gleichungen für die Ermittlung der parallelen Ausführungszeit

Die parallele Ausführungszeit besteht aus zwei Teilen, der Computation-Zeit $t_{comp}$ und der Kommunikationszeit $t_{comm}$, also

$$t_P = t_{comp} + t_{comm}.$$

**Computation-Zeit**: Die Abarbeitungszeit ist i.A. eine Funktion der Anzahl der Prozessoren $p$ und der Datenelemente $n$, also

$$t_{comp} = f(n, p).$$

Der Einfachheit halber wird als Zeiteinheit ein Abarbeitungsschritt (computational step) verwendet und weiters wird davon ausgegangen, dass es sich um ein homogenes System handelt, also alles gleiche Prozessoren.

**Kommunikationszeit**: Diese hängt ab von
- der Anzahl der Nachrichten,
- der Größe der Nachrichten,
- der Verbindungsstruktur und
- des Übertragungsmodus.

Als erste Annäherung gilt für die Kommunikationszeit einer Nachricht

$$t_{comm} = t_{startup} + wt_{data}$$

mit
- $t_{startup}$ als Startup-Zeit oder auch Nachrichten-Latenz, also im Prinzip die Zeit für das Senden einer Nachricht ohne Daten und beinhaltet die Verzögerung durch den Sende- bzw. Empfangsstack (also das Verpacken/Entpacken der Nachricht) nicht aber die Übertragung. Also der reine Protokolloverhead. $t_{startup}$ wird als konstant betrachtet.
- $t_{data}$ als Übertragungszeit für das Versenden eines Datenwortes
- $w$ als Anzahl der übertragenen Datenwörter.

In der Praxis wird aber in den meisten Fällen die StartUp-Zeit dominierend sein, außer bei sehr großen Datenmengen.

Es folgt damit für die gesamte Kommunikationszeit aller Nachrichten $t_{comm} = \sum_{i=1}^{n} t_{comm_i}$ .

**Benchmark Faktoren**: Der *SpeedUp-Faktor* (*siehe 1.2.1*) und das *Computation/Communication Ratio* (*siehe 1.2.3*) sind beide Abhängig von der Anzahl der Prozessoren und Datenelemente und geben somit Auskunft über die Skalierbarkeit der parallelen Lösung mit steigender System- bzw. Problemgröße (*wenn man davon ausgeht, dass die Problemgröße nur von der Anzahl der zu bearbeitenden Daten abhängt*). Das *Computation/Communication Ratio* im Besonderem spiegelt den Einfluss der Kommunikation wider.

**Latency Hiding**: Um die Kommunikationszeit als dominierenden Faktor der Gesamtzeit zu entschärfen, gibt es verschiedene Ansätze, diese mit sinnvoller Rechenzeit zu überbrücken und somit quasi zu „verbergen“:

- Verwendung nichtblockierender Send-Routinen um während des Versenden der Nachricht bereits schon wieder nachfolgenden Code ausführen zu können. Dies ist auch schon teilweise mit den lokal-blockierenden Send-Routinen möglich, da ja die Übertragung zum Empfänger bereits nicht mehr blockiert.
- Zuweisung mehrerer Prozesse auf einen Prozessor und Ausnutzung der Multitaskingfähigkeit des OS. Blockiert ein Prozess so wird vom Scheduler ein anderer bereiter Prozess aktiviert. -> „*parallel slackness* von *m/p*“ beschreibt das Verhältnis *m*-Prozesse zu *p*-Prozessoren falls $p < m$.
- Multithreading.

Die Effizienz des Latency Hiding bei Multitasking- bzw. Multithreadingansätzen hängt allerdings stark von der Qualität des OS-Schedulers bzw. dessen Scheduling-Strategie ab.

## 2.3.2 Zeitkomplexität

***Formale Definitionen:*** *(Details siehe Kurs „Theoretische Grundlagen der Informatik“)*

**O-Notation:**

$f(x) = O(g(x))$ bedeutet, dass es positive Konstanten *c* und $x_0$ gibt, so dass

$0 \le f(x) \le cg(x)$ *für alle* $x \ge x_0$ gilt.

*f(x)* wächst also mit ansteigendem *x* maximal so schnell wie *g(x)*. Damit wird die worst-case Situation beschrieben. (-> obere Schranke)

**Θ-Notation:**

$f(x) = \Theta(g(x))$ bedeutet, dass es positive Konstanten $c_1$, $c_2$ und $x_0$ gibt, so dass

$0 \le c_1 g(x) \le f(x) \le c_2 g(x)$ *für alle* $x \ge x_0$ gilt.

*f(x)* wächst also mit ansteigendem *x* gleich schnell wie *g(x)*.

**Ω-Notation:**

$f(x) = \Omega(g(x))$ bedeutet, dass es positive Konstanten *c* und $x_0$ gibt, so dass

$0 \le cg(x) \le f(x)$ *für alle* $x \ge x_0$ gilt.

*f(x)* wächst also mit ansteigendem *x* mindestens so schnell wie *g(x)*. Damit wird best-case Situation beschrieben. (-> untere Schranke)

***Zeitkomplexität paralleler Algorithmen:***

Bei der Analyse der Zeitkomplexität, welche Terme niedrigerer Ordnung vernachlässigt – hier also $t_{startup}$, erhält man für die Kommunikationszeit die Komplexität $O(n)$, also $t_{comm}$ steigt linear mit der Anzahl der Daten. Die Komplexität der gesamten parallelen Lösung ist dann die Summe aus den Komplexitäten für die Kommunikation und die Berechnung.

*Computation/Communication Ratio*: Da die Kommunikation normalerweise recht kostenintensiv ist (im Sinne von Berechnungszeit) sollte Idealerweise die Komplexität der Berechnung $t_{comp}$ größer sein als jene der Kommunikation, damit mit steigender Problemgröße die Kommunikation überhaupt an Dominanz verliert und es zu einer Performance-Verbesserung kommt.

***Kosten und kostenoptimale Algorithmen:***

Die Kosten für einen Algorithmus können definiert werden als

*Kosten = (Ausführungszeit) x (Anzahl der verwendeten Prozessoren)*

Ein kostenoptimaler Algorithmus ist jener, bei dem die Kosten proportional zur Ausführungszeit auf einem Single-Prozessor System sind, also

$$Kosten = t_P \times p = k \times t_S \text{ mit } k = \text{konstant},$$

bzw. in Bezug auf die Zeitkomplexität, wenn die

*Parallele Zeitkomplexität x Anzahl der Prozessoren = Sequentieller Zeitkomplexität*

ist.

## 2.3.3 Anmerkungen zur asymptotischen Analyse

Bei der Zeitkomplexitätsanalyse nach der O-Notation werden asymptotische Methoden verwendet, wo die betrachtete Variable (hier die Anzahl der Daten bzw. der Prozessoren) für die Analyse gegen Unendlich tendiert, was aber eigentlich nie der praktischen Situation entspricht, da immer die Anzahl der Prozessoren beschränkt ist und große Datenmengen schnell nicht mehr zu managen sind. Darüber hinaus werden Terme niedrigerer Ordnung in der Analyse nicht berücksichtigt, die aber z.B. bei der Kommunikation von kurzen Nachrichten durchaus eine entscheidender Faktor für die Zeitkomplexität sein können.

***Shared Memory Programme:***

Hier fällt der Kommunikationsaspekt gänzlich weg (bis auf z.B. Zugriffskonflikte) und die Zeitkomplexität bezieht sich praktisch nur auf die Berechnung, wie in sequentiellen Programmen auch. Damit hat die Analyse nach der O-Notation wesentlich mehr Aussagekraft als bei Message Passing Systemen.

## 2.3.4 Kommunikationszeit für Broadcast/Gather

Obwohl die Zeitkomplexität für einen Broadcast auf einer Single-Ethernet Verbindung nur O(1) beträgt, ist dies auf den meisten Computer-Cluster nicht anwendbar, da diese meist verschiedenste Netzwerkstrukturen einsetzen. Die *Broadcast*-Nachricht wird meist an mehrere Ziele gesendet die wiederum diese an mehrere weitere *N* Ziele (1:*N* fan-out) weiterleiten (z.B. switched Ethernet). Es folgt also für

$$t_{comm} = N(t_{startup} + wt_{data})$$

und damit eine Zeitkomplexität von $O(N)$. Wird als Verbindungsstruktur eine Baumstruktur eingesetzt, so ist die Zeitkomplexität zusätzlich noch von der Baumhöhe abhängig, also

$$t_{comm} = k(\log p)(t_{startup} + wt_{data})$$

für *p* Endknoten und einem *k*-ären Baum. *(Nachteil eines Baumes: wenn 1 Knoten ausfällt, dann sind alle darunter liegenden Knoten nicht mehr erreichbar.)*
Für den umgekehrten Weg, also *Gather*, gilt analoges.

## 2.4 Debugging und empirische Evaluierung paralleler Programme

### 2.4.1 Low-Level Debugging

Die parallele Abarbeitung besitzt Eigenschaften wie z.B. das Timing von Ereignissen, die von sequentiellen Debugger nicht erfasst werden (ist auch schon bei Multitasking-Applikationen problematisch). Die Instrumentierung des Codes bietet auch keine befriedigende Lösung, da neben der Beeinflussung der Berechnungszeit (wie auch bei sequentiellen Code) auch noch die Art der Verschachtelung der parallelen Abarbeitung beeinflusst wird. Abhilfe bieten deshalb nur spezielle parallele Debugger. *(-> Begleittext empfiehlt aber Einfügen von Debug-Outputs, aktivierbar über Preprozessor-Kommandos.)*

### 2.4.2 Visualisierungs-Tools

Es wird über Prüfpunkte (die vom OS ermöglicht werden) die Ausführung überwacht, was natürlich auch einen gewissen Einfluss auf das Zeitverhalten des betroffenen Knoten hat.

- *Space-Time Diagramm (od. Process-Time Diagramm)*: Zeigt die Ausführung des Programms, welche Prozesse blockiert sind, welche im Idle usw. *(-> Bild 2.17, S.72)*
- *Utilization-Time Diagramm*: Zeigt an, wie viel Zeit jeder Prozess für Kommunikation, Warten, Library-Aufrufe usw. verbraucht. Also ein Indikator für die Effizienz der Abarbeitung.

### 2.4.3 Debugging Strategien:

3-Schritte Ansatz für das Debugging von Message Passing Programmen:

1. Wenn möglich, das Programm zuerst als Single-Process laufen lassen und mit einem sequentiellen Debugger arbeiten.
2. Ausführung des Programms als Multitasking-Programm mit 2 bis 4 Prozessen auf einem Single-Prozessor System zur Überprüfung der korrekten Kommunikation.
3. Das Programm mit denselben 2 bis 4 Prozessen auf verschiedene Computer verteilen und den Einfluss des Netzwerkverhaltens auf Synchronisierung und Timing testen.

### 2.4.4 Programmevaluierung

- Messen der Ausführungszeit: interessante Codeteile mit einer Zeitmessung umschließen.
- Messung der Kommunikationszeit: z.B. mit der Pingpong-Methode od. auch Echo-Methode für die Point-To-Point Kommunikation.
- Profiling: Dabei wird typischerweise der Code durch den Profile mit entsprechenden Messpunkten instrumentiert um dann statistische Auswertung durchführen zu können, welche Codeteile wie oft abgearbeitet werden. Die Instrumentierung beeinflusst natürlich auch das Zeitverhalten des Programms.

### 2.4.5 Anmerkungen zur Optimierung paralleler Programme

Neben den Optimierungsmethoden, die auch bei sequentiellen Programmen angewendet werden können, gibt es für parallele Programme noch weitere Ansatzpunkte:

- Anzahl der Prozesse ändern um eine für das System oder Problem günstige Prozess-Granularität zu erreichen.
- Länge der Nachrichten vergrößern um den Einfluss der StartUp Zeit zu verringern.
- Es kann oft von Vorteil sein, Werte lokal neu zu berechnen anstatt sie an die jeweiligen Prozesse zu übertragen.
- Methoden des Latency-Hiding anwenden.

- Analyse des kritischen Pfads. D.h. Bestimmung der verschiedenen parallelen Teile des Programms und Ermittlung des längsten Pfades der die Gesamtausführungszeit dominiert.
- Berücksichtigung der Speicherhierarchien (First-Level Cache, Second-Level Cache, usw.). Z.B. Umordnung der Speicheranforderungen od. arithmetischer Operationen.

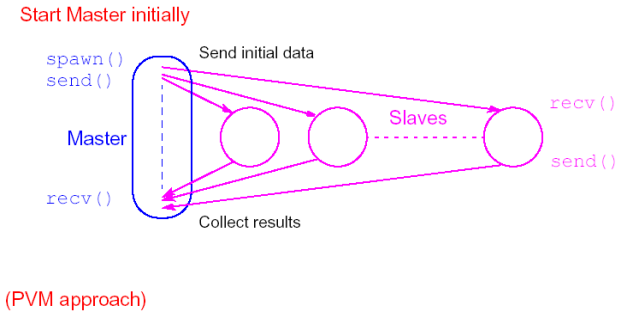
# 3. Embarrassingly Parallel Computations

Damit werden Programme bezeichnet, die sich relativ einfach in komplett voneinander unabhängige parallele Teile zerlegen lassen, wobei jeder Teil von einem separaten Prozessor bearbeitet werden kann.

## 3.1 Ideale parallele Berechnung

Darunter versteht man Berechnungen, die einfach in komplett voneinander unabhängige Teile zerlegt werden und auch parallel berechnet werden können (embarrassingly parallel). Eine Parallelisierung solcher Probleme erfordert keine speziellen Techniken oder Algorithmen und Idealerweise ist keine Kommunikation zwischen den parallelen Prozessen nötig. Jeder Prozess benötigt also seine eigenen Eingabedaten und die erzeugten Resultate werden nicht von anderen Prozessen benötigt. Die wesentliche Eigenschaft ist also, dass es Idealerweise keine Interaktion zwischen den Prozessen gibt.

Da in der Praxis am Beginn zuerst die Daten an die Prozesse verteilt werden müssen und am Ende die Resultate eingesammelt werden müssen, bietet sich ein klassischer Master-Slave Ansatz an, mit dynamischer oder statischer Prozesserzeugung, wobei der Master auch als Berechnungsprozess eingesetzt werden kann.




(PVM approach)

## 3.2 Beispiele für Embarrassingly Parallel

### 3.2.1 Geometrische Transformationen von Bildern

Hier geht es vor allem um Aufgabenstellungen aus den Bereichen Bildverarbeitung und Computer-Grafik. Bei geometrischen Transformationen werden nur die Koordinaten der Pixel verändert, unabhängig von den anderen Pixel, es ist also eine echte embarrassingly parallele Berechnung. Die häufigsten Transformationsarten sind

- Verschiebung
- Skalierung
- Rotation
- Clipping

Das Hauptaugenmerk bei der parallelen Programmierung liegt dabei auf der Aufteilung des Bitmap in Pixelgruppen für die einzelnen Prozessoren. Diese Gruppierung erfolgt üblicherweise durch Aufteilung des Bildes in quadratische/rechteckige Regionen (subimages) oder in Zeilen und Spalten.

***Analyse:***

Unter der Annahme, dass für die Transformation für jedes Pixel zwei Berechnungsschritte benötigt werden ( $x' = x + \Delta x; y' = y + \Delta y$ ), ergibt sich für die sequentielle Berechnungszeit von $n$ x $n$ Pixel $t_S = 2n^2$ und damit eine Zeitkomplexität von $O(n^2)$.

Die parallele Version für das Beispiel im Buch mit p Prozessoren, der zeilenweisen Bearbeitung durch die Slaves und einer Resultatstruktur von jeweils 4 Werten *(x/y-Koordinate für alte und neue Position)* je Pixeltransformation besteht aus:

- Kommunikationszeit: $t_{comm} = p(t_{startup} + t_{data}) + n^2(t_{startup} + 4t_{data}) = O(p + n^2)$
- parallele Berechnung: $t_{comp} = \frac{2n^2}{p} = O(\frac{n^2}{p})$

Damit folgt für die gesamte Bearbeitungszeit $t_P = t_{comm} + t_{comp} = O(n^2)$ für eine fixe Anzahl von Prozessoren.

Weiters folgt für die Computation/communication ratio = $O(\frac{n^2 / p}{p + n^2})$ = konstant bei steigender Problemgröße und fixer Anzahl von Prozessoren.

Da bei geometrischen Transformationen typischerweise der Berechnungsaufwand für die Transformation der einzelnen Pixel meist relativ gering ist verglichen mit dem Aufwand des Datentransfers der Teilbilder zu den einzelnen Slaves und der Ergebnisse retour zum Master (wie sie in einem Message Passing System auftreten), sind für diese Problemstellungen Shared Memory Multiprozessoren wesentlich besser geeignet als Message Passing Systeme.

### 3.2.2 Mandelbrot Set

Im Unterschied zu geometrischen Bildtransformationen wird beim Mandelbrot Set das Bild erst erzeugt, was wesentlich mehr Berechnungen erfordert als bei der Transformation. Das Mandelbrot Set ist ein Satz von Punkten in einer komplexen Ebene welche durch Iteration einer komplexen Funktion der Form $z_{k+1} = z_k^{\,2} + c$ berechnet werden können, wobei $z_{k+1}$ die *(k+1)*te Iteration der komplexen Zahl $z = a + bi$ (mit $i = \sqrt{-1}$) und *c* eine komplexe Zahl ist welche die Position eines Punktes in der komplexen Ebene angibt. Die Iterationen werden so lange durchgeführt bis die Länge des Vektors gegeben durch $z_{len} = \sqrt{z_{real}^{\,2} + z_{imag}^{\,2}}$ größer *2* ist (also *z* geht eventuell gegen unendlich) oder bis eine definierte Anzahl max. Iterationen erreicht ist. Das Mandelbrot Set ist also quasi-stabil, d.h. *z* verändert sich nur innerhalb fester Grenzen.
Das Mandelbrot Set wird wegen der hohen Rechenintensivität gerne als Benchmark Test für parallel und sequentielle Computersysteme verwendet.

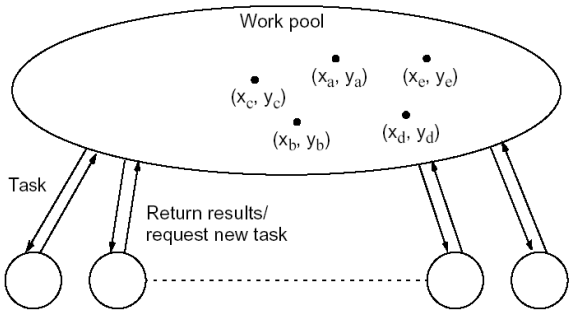
#### *Parallelisierung der Mandelbrot Set Berechnung:*

Das Mandelbrot Set ist besonders geeignet für eine Parallelisierung, da die Berechnung der einzelnen Pixel ohne jegliche Informationen über die benachbarten Pixel erfolgen kann. Die Pixelberechnung selbst ist allerdings weniger geeignet für eine Parallelisierung.

**Statische Task-Zuordnung**: Da die Anzahl der Iterationen für jedes Pixel verschieden sein kann und die beteiligten Prozessoren durchaus verschieden leistungsfähig sein können, entsteht bei der statischen Task-Zuordnung das Problem (jeder Prozess erhält eine Teilregion des Gesamtbildes zur Berechnung), dass bestimmte Prozesse schon wesentlich früher ihre Berechnungen durchgeführt haben als andere Prozesse und damit womöglich einen Großteil der Gesamtberechnungszeit ungenutzt bleiben, die Lastverteilung ist i.A. nicht optimal. Darunter leidet natürlich die Effizienz des Gesamtsystems.

**Dynamische Task-Zuordnung – Work Pool/Prozessor Farmen**: Um die Lastverteilung zu verbessern muss diese dynamisch gestaltet werden. Dies kann z.B. durch einen work pool Ansatz erreicht werden. Im work pool sind alle Teilaufgaben enthalten, die vom parallelen

System bearbeitet werden sollen. Jeder Prozess holt sich eine neue Teilaufgabe aus dem Pool sobald er im Idle ist. Damit werden zwar weitgehend „unbeschäftigte" Prozesse vermieden, trotzdem kann gerade beim Mandelbrot Set dadurch nicht gänzlich verhindert werden, dass einige Prozesse wesentlich früher mit ihren Berechnungen fertig sind als andere (wegen ungünstiger Pixelkoordinaten) und mangels weiterer Teilaufgaben im Pool immer noch relativ zur gesamten Bearbeitungszeit zu lange unbeschäftigt sind. Hier würde nur ein entsprechendes Preprocessing der Task-Zuordnung die Lastverteilung verbessern.




***Analyse:***

Unter der Annahme dass *n* Pixel berechnet werden soll und die Anzahl der Iterationen eine Funktion von *c* ist aber ein Limit *max_iter* nicht überschreitet folgt für $t_S \leq \max_iter \times n$ und damit eine sequentielle Zeitkomplexität von $O(n)$.
Die parallele Version besteht aus 3 Phasen:

- Datentransfer zu den *p-1* Slaves: $t_{comm_1} = (p-1)(t_{startup} + t_{data})$
- parallele Berechnung: $t_{comp} \leq \frac{\max_iter \times n}{p-1}$
- Transfer der Ergebnisse zum Master: $t_{comm_2} = k$, also konstant, da die Anzahl der Bildzeilen inkl. der enthaltenen Pixel fix ist, solange die Auflösung des Bildes unverändert bleibt.

Damit ergibt sich eine gesamte Bearbeitungszeit $t_P = t_{comp} + t_{comm_1} + t_{comm_2}$ und der potentielle Speedup nähert sich *p* wenn *max_iter* sehr groß ist.
Das Computation/communication ratio = $O(n)$ bei einer fixen Anzahl von Prozessoren.

### 3.2.3 Monte Carlo Methoden

Die Monte-Carlo Methode verwendet Prinzipien der Wahrscheinlichkeitsrechnung und Statistik, um komplexe Probleme zumindest näherungsweise zu lösen. Ein Monte-Carlo Algorithmus liefert also ein korrektes Ergebnis nur mit einer gewissen Wahrscheinlichkeit, ist aber in jedem Fall effizient *(siehe z.B. „Randomisierter Primzahlentest" – [Ottm02] S.656).*
Oder nach Wood: *„Bei einer Monte-Carlo Methode wird die Lösung eines numerischen Problem, durch ein Verfahren angenähert, welches stochastische Elemente einbezieht, die im ursprünglichen Problem gar nicht vorhanden sind."* -> siehe auch die näherungsweise Berechnung von $\pi$ (Kurstext S.94). Neben dem Einsatz in der numerischen Mathematik werden MC-Methoden auch noch für die

- Zuverlässigkeitsuntersuchung technischer Systeme
- Probleme des Operations Research (Lagerhaltung, Transportprobleme, usw.)
- Untersuchung von Erdbeben u. anderen Naturphänomenen
- Entscheidungsfindung durch Simulation im Investment Banking

eingesetzt.

Bei dem im Buch angeführten Bsp. wird eine MC-Methode für die näherungsweise Lösung des bestimmten Integral $I = \int_{x1}^{x2} (x^2 - 3x)dx$ eingesetzt. Dafür werden innerhalb der

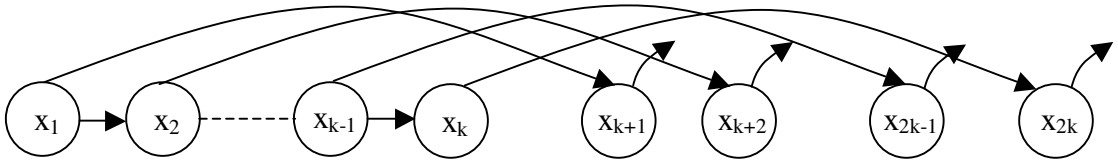
Schranken *x1 – x2* mittels eines Zufallszahlengenerators zufällige Werte für *x* erzeugt und dann als Näherung $I_{MC} = \sum_{N}(x^2 - 3x)$ über *N* Samples berechnet.

***Parallelisierung:***

Da die Iterationen für die Aufsummierung völlig voneinander unabhängig sind, ist die Problemstellung sehr einfach zu parallelisieren (embarrassingly parallel). Es wird jedem Slave eine Anzahl von Zufallszahlen im Bereich *x1 – x2* für die Aufsummierung übergeben, wobei jeder Slave nach erfolgter Teilsummenbildung neue Samples vom Master anfordert. Der Master erzeugt so viele Zufallszahlen, bis die gewünschte Näherung erreicht ist. Danach werden die Teilsummen von den Slaves und dem Master kollektiv über ein *reduce*-Kommando innerhalb der spezifizierten Prozessgruppe aufsummiert. Die Zufallszahlenerzeugung könnte auch in die Slaves verlagert werden, damit würde der Kommunikationsaufwand verringert. Eine entsprechende Unabhängigkeit der Generatoren in den Slaves könnte über die Initialisierung mit der Prozess-ID der Slaves erreicht werden.

***Parallele Zufallszahlen-Erzeugung:***

Ein häufig eingesetzter Pseudo-Zufallszahlengenerator ist der „*linear kongruente*“ Generator der eine Sequenz pseudo-zufälliger Zahlen nach der Funktion $x_{i+1} = (ax_i + c) \bmod m$ berechnet, wobei *a*, *c* und *m* Konstante sind und günstige Werte für einen guten Generator sind z.B. *a=16807*, $m = 2^{31} - 1$ (eine Primzahl) und *c=0*.
Für eine Parallelisierung werden nun z.B. bei *k* Prozessoren die erste *k* Werte sequentiell erzeugt und danach können mit diesen *k* Werten die nächsten *k* Werte parallel erzeugt werden:




Eine Vereinfachung der Berechnung kann noch erreicht werden, wenn *m* eine Potenz von 2 ist, womit die *mod*-Operation durch eine einfache *Shift-Right*-Operation ersetzt werden kann, welche die „*herausgeschobenen*“ untersten Bits als Rest liefert.

Wegen der Bedeutung der Qualität der Zufallszahlengeneratoren für die MC-Methoden wurde unter anderem eine spezielle Library für die Erzeugung von Zufallszahlensequenzen für parallele Prozesse entwickelt, die SPRNG (Scalable Pseudorandom Number Generator). Sie besitzt mehrere unterschiedliche Generatoren und Features zur Minimierung der Interprozess-Übereinstimmungen der erzeugten Sequenzen und besitzt eine Interface zum MPI.

# 4. Partitioning und Divide-and-Conquer Strategien

## 4.1 Partitioning

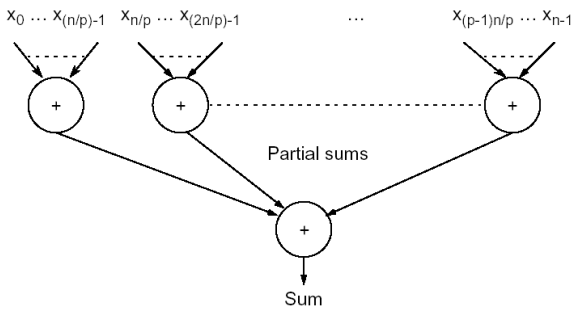
### 4.1.1 Partitioning Strategien

Dabei wird die Aufgabe in separate Teile aufgeteilt welche dann jedes für sich bearbeitet wird. Idealerweise können die einzelnen Teile unabhängig voneinander bearbeitet werden, ohne irgendwelche Interaktionen zwischen den einzelnen Teilen. In den meisten Fällen ist jedoch zumindest eine Zusammenführung der Teilergebnis zu einem Gesamtergebnis notwendig.

Die Partitionierung kann

- auf die Daten -> *data partitioning* oder *domain decomposition*
- aber auch auf die Funktionen eines Programms -> *functional decomposition*

angewendet werden. Da parallel bearbeitbare Funktionen in Programmen relativ selten sind, ist die Hauptstrategie bei der parallelen Programmierung die Partitionierung von Daten. Z.B. die Aufsummierung einer Reihe von Zahlen durch parallele Teilsummenbildung der Slaves und anschließender Aufsummierung der Teilsummen durch den Master:
Da alle Slaves mit den Zahlensequenzen versorgt werden müssen, die sie aufsummieren sollen, ist der Kommunikationsaufwand in einem Message-Passing System relativ hoch. Deshalb wäre in diesem Fall ein Shared-Memory System durchaus geeigneter.




***Analyse:***

Die sequentielle Berechnung benötigt *n-1* Additionen mit einer Zeitkomplexität von $O(n)$. In der parallelen Implementierung mit *p* Slaves (SPMD-Modell, also der Mastercode ist in einem der Slaves implementiert) ergibt sich für

(1) den Datenversand $t_{comm_1} = p(t_{startup} + (n/p)t_{data})$, bzw. bei Verwendung der *scatter*-Funktion $t_{comm_1} = t_{startup} + nt_{data}$, insgesamt aber trotzdem eine Zeitkomplexität $O(n)$,

(2) die Teilsummenberechnung $t_{comp_1} = n/p - 1$,

(3) Rückgabe der berechneten Teilsummen $t_{comm_2} = p(t_{startup} + t_{data})$, bzw. bei Verwendung der *gather*- und *reduce*-Funktion $t_{comm_2} = t_{startup} + pt_{data}$,

(4) die Aufsummierung der *p* Teilsummen $t_{comp_2} = p - 1$.

Also folgt für die gesamte Ausführungszeit $t_P = O(n)$. Es ist ein geringer Speedup bei gleichbleibender Anzahl von Prozessoren zu erwarten, wir die Kommunikation vernachlässigt so tendiert der Speedup gegen *p* für große *n*. Bei kleinen *n* ist der Speedup jedoch sehr gering und verschlechtert sich sogar mit steigendem *p*.
Ebenso lässt das Computation/communication ratio bei fixierter Anzahl von Prozessoren keine großen Möglichkeit für eine Verbesserung erwarten.

### 4.1.2 Divide and Conquer

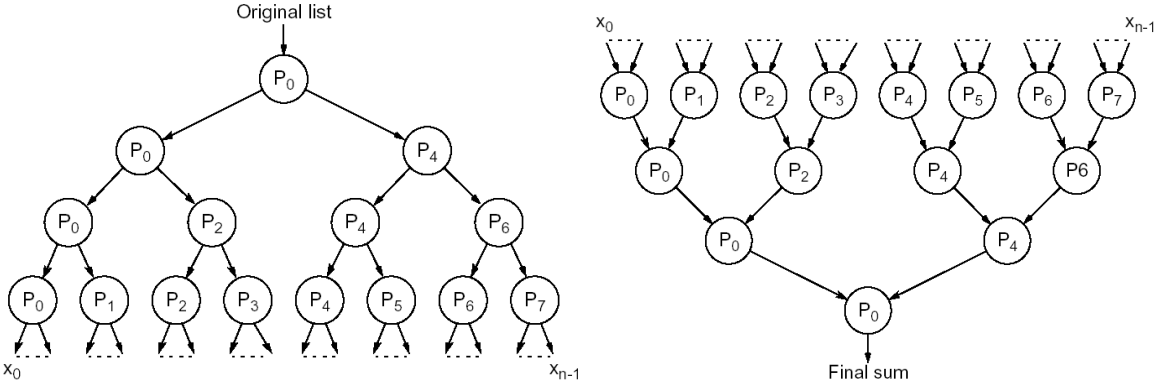
Dabei wird eine Problem in Teilprobleme zerlegt, welche die selbe Form haben wie das ursprüngliche Problem. Es wird mit Hilfe der Rekursion diese Zerlegung solange durchgeführt, bis keine weitere Teilung mehr möglich ist. Die so vereinfachten Teilaufgaben werden dann bearbeitet und die Ergebnisse kombiniert und wieder zu einer größeren Teilaufgabe vereint, solange bis die Ebene des Ausgangsproblems erreicht ist und somit alle Teilergebnisse zu einem Gesamtergebnis zusammengeführt wurden.

Diese Methode kann z.B. für globale Operationen auf Listen wie die Sortierung der Liste (Mergesort od. Quicksort), das Auffinden eines max./min. Wertes oder das Suchen in Listen angewandt werden. Werden bei jeder Zerlegung genau zwei Teilprobleme generiert, so entsteht beim rekursiven Divide-and-Conquer Ansatz ein binärer Baum, der bei der Zerlegung nach unten verfolgt werden und bei der Zusammenführung der Ergebnisse nach oben.

***Parallele Implementierung:***

In einer sequentiellen Implementierung kann immer nur ein Knoten des Baumes gleichzeitig abgearbeitet werden, bei der parallelen Implementierung können aber verschiedene Teile des Baumes simultan abgearbeitet werden. Eine adhoc-Lösung wäre nun, für jeden Knoten des

Baumes einen Prozessor vorzusehen, was aber äußerst ineffizient wäre, da jeder Prozessor nur in einer Baumebene aktiv wäre und somit für eine Zerlegung in $2^m$ Teile (Blätter) genau $2^{m+1}-1$ Prozessoren notwendig wären. Eine effizientere Lösung ist die Wiederverwendung von Prozessoren in verschiedenen Baumebenen, womit nur mehr $2^m$ Prozessoren benötigt werden, z.B.:




Diese Baumkonstruktion lässt sich hervorragend mit einer Hypercube-Architektur realisieren. Prozessoren kommunizieren mit jenen Prozessoren, die sich in ihrer binären Adresse in genau einem Bit unterscheiden, beginnend beim MSB für die Zerlegungsphase verwendet und beginnend beim LSB für die Kombinierungsphase.

***Analyse:***

Für das Bsp. im Kurstext, also die Aufsummierung von Listenelementen, wird vorausgesetzt, dass die Anzahl der Daten *n* eine Potenz von 2 ist und es wird die Setup-Zeit $t_{startup}$ der Kommunikation vernachlässigt. Für die Zerlegungsphase wird angenommen, dass im wesentlichen nur Kommunikation stattfindet und erst in der Kombinierungsphase zusätzlich zu Kommunikation auch noch die Berechnungen der Teilsummen.

**Kommunikation**: Aufgrund der Baumstruktur ergeben sich bei p Prozessen für die Zerlegungsphase *log p* Schritte, also folgt für

$$t_{comm_1} = \frac{n}{2}t_{data} + \frac{n}{4}t_{data} + ... + \frac{n}{p}t_{data} = \frac{n(p-1)}{p}t_{data}$$

Die Kombinierungsphase ist ähnlich, nur mit dem Unterschied, dass nur ein Datum (die Teilsumme) je Nachricht versendet wird.

$$t_{comm_2} = (\log p)t_{data}$$

Somit folgt für die Zeitkomplexität der gesamten Kommunikationszeit $t_{comm} = O(n)$ für eine fixe Anzahl von Prozessoren.

**Berechnung**: Am Ende der Zerlegungsphase werden n/p Zahlen addiert, danach in jeder Ebene der Kombinierungsphase nur mehr eine Addition der Teilsummen:

$$t_{comp} = \frac{n}{p} + \log p = O(n)$$ für ein fixe Anzahl von Prozessoren.

**Speedup Faktor**: Der max. erreichbare Speedup wäre *p*, wenn alle Prozessoren ihre Teilsumme berechnen, allerdings wird der praktisch erreichbare Speedup wegen der Zerlegungs- und Kombinierungsphase darunter liegen.

### 4.1.3 M-ärer Divide-and-Conquer

Wird ein Aufgabe bei der Zerlegungsphase in *m* Teilaufgaben zerlegt, so erhält man einen *m*-ären Baum, also ein Knoten hat maximal *m* Kindknoten. Mit wachsendem *m* ist auch eine

höhere Parallelisierung zu erwarten, da eben mehr Teile simultan bearbeitet werden können. Ein Bsp. wäre die Aufteilung eines Bildes in der Bildverarbeitung in Quadranten, also einen Quadtree.

## 4.2 Beispiele für Partitioning und Divide-and-Conquer

### 4.2.1 Sortierung mittels Bucket Sort

Bucket Sort basiert nicht auf einer „compare and exchange"-Methode ist aber vom Prinzip her eine Partitioning-Methode. Bucket Sort funktioniert nur dann wirklich gut, wenn die zu sortierenden Zahlen annähernd gleichmäßig über einen bekannten Intervall verteilt sind.

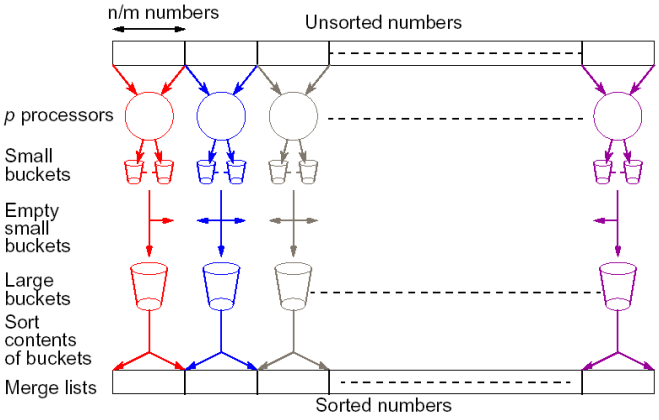
#### *Sequentieller Algorithmus:*

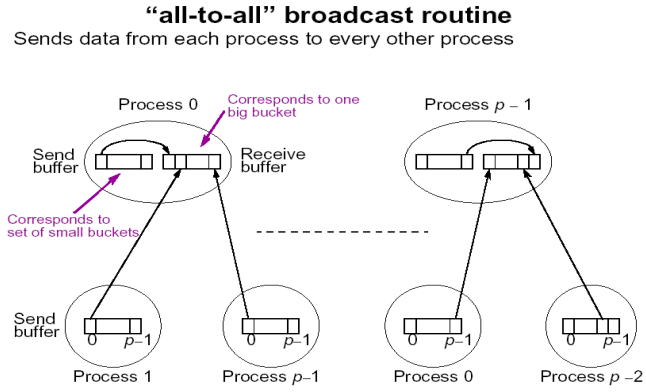
Das Intervall *a* wird in *m* Regionen (Buckets) unterteilt und die zu sortierenden Zahlen werden in die entsprechenden Buckets verteilt (z.B. wird die Zahl durch *a/m* dividiert und das Resultat entspricht damit dem Ziel-Bucket – also in $O(n)$). Bei gleichmäßig verteilten *n* Werten über das Intervall *a* befinden sich danach *n/m* Werte in einem Bucket. Danach werden die Werte in den Buckets mit z.B. Quicksort od. Mergesort (in $O((n/m)\log(n/m))$) sortiert und danach der sortierte Inhalt aller Buckets zu einer sortierten Liste zusammengehängt. Unter Vernachlässigung des letzten Schrittes folgt für $t_S = n + m(n/m\log(n/m)) = O(n\log(n/m))$. Wird für $n = km$ mit $k$ konstant angenommen so folgt sogar $O(n)$ was wesentlich besser ist als die untere Grenze $\Omega(n\log n)$ der „compare and exchange"-Algorithmen ist. *(Anm.: Laut [Ottm02] S.115ff funktioniert der Bucket Sort allerdings etwas anders!)*

#### *Paralleler Algorithmus:*

Eine einfache Parallelisierung besteht bereits darin, jedem Bucket einen Prozessor zuzuordnen. Damit erreicht man bereits eine Reduzierung der Zeitkomplexität auf $O(n/p\log(n/p))$ bei $p=m$ Prozessoren. Der Nachteil ist hier, dass jeder Prozessor alle Zahlen der unsortierten Liste prüfen muss, um die passenden Zahlen für sein Bucket zu finden.

Eine verbesserte Parallelisierung kann erreicht werden, wenn die Liste der unsortierten Zahlen bereits in p Regionen aufgeteilt wird, wobei jedem Prozessor eine Region zugeteilt wird. Jeder Prozessor verteilt nun die Zahlen der ihm zugewiesenen Region vorerst in kleine Buckets die das gesamte Intervall repräsentieren. Danach wird der Inhalt jedes kleinen Buckets an den entsprechenden Prozessor gesandt (kleines Bucket *i* an Prozessor *i*) und dann erst in die eigentlichen Buckets einsortiert. Dies erfordert einen Datentransfer von jedem Prozessor zu jedem anderen Prozessor, also ein „all-to-all"-Transfer. Es findet also bezogen auf die Daten ein Transfer der Zeilenelemente eines Quellarrays auf die Spaltenelemente eines Zielarrays statt -> Transponierung einer Matrix.



***Analyse:***

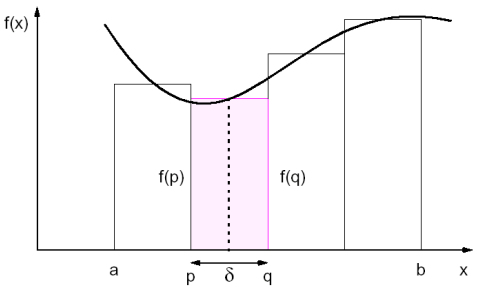
Zu Beginn werden die gesamten Daten an alle Prozessoren mittels *broadcast* oder *scatter* gesendet und die einzelnen Prozessoren führen für sich jeweils die Gruppierung der Ausgangsdaten durch. Die Ablauf des parallelisierten Bucket Sort lässt sich dann grob in 4 Phasen aufteilen:

1. *Bearbeitung u. Kommunikation*: Die Gruppierung der Daten kann in konstanter Zeit erfolgen und wird deshalb vernachlässigt. Für die Kommunikationszeit des broadcast/scatter folgt: $t_{comm_1} = t_{startup} + nt_{data}$
2. *Bearbeitung*: Verteilung von *n/p* Zahlen in *p* kleine Buckets: $t_{comp_2} = n/p$
3. *Kommunikation*: p-1 kleine Buckets werden von jedem Prozessor an die anderen p-1 Prozessoren verteilt, wobei bei gleichmäßiger Verteilung der Zahlen jedes kleine Bucket $n/p^2$ Zahlen enthält: $t_{comm_3} = p(p-1)(t_{startup} + (n/p^2)t_{data})$ , falls die Kommunikation nicht überlappend durchgeführt werden kann, andernfalls ergibt sich $t_{comm_3} = (p-1)(t_{startup} + (n/p^2)t_{data})$ *(falls ein all-to-all Kommando zur Verfügung steht).*
4. *Bearbeitung*: Die n/p Zahlen in den großen Buckets werden parallel sortiert: $t_{comp_4} = (n/p)\log(n/p)$

Die gesamte Ausführungszeit und Speedup Faktor folgen leicht aus obigen Teilergebnissen, ebenso wie die Computation/Communication ratio. Die Analyse gilt unter der Annahme, dass die zu sortierenden Zahlen gleichmäßig über den gesamten Werteintervall verteilt sind. Ist dies nicht der Fall, so dominiert die Sortierung innerhalb der Buckets die Gesamtzeit, was im worst case (alle Zahlen in einem Bucket) bedeutet, dass die Zeitkomplexität von dem verwendeten „compare-and-exchange"-Algortihmus erreicht wird.

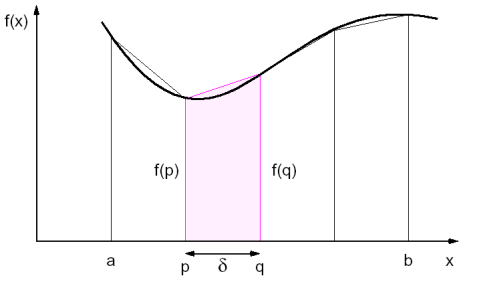
## 4.2.2 Numerisches Integrieren

Für die Berechnung eines bestimmten Integrals der Form $I = \int_a^b f(x)dx$ kann das Intervall *a-b* in mehrere Abschnitte unterteilt werden, wobei jeder Teilabschnitt z.B. durch Näherung mittels Rechteck von einem separaten Prozess berechnet werden kann -> *Quadratur*-Methode.



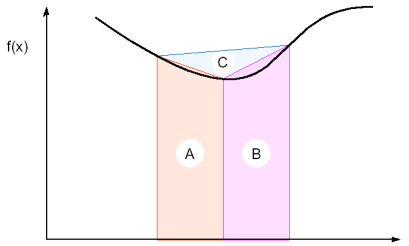

***Static Assignment:***

Eine weitere aber nicht unbedingt bessere Methode ist die *Trapezoid*-Methode, wobei sich dann die Fläche der einzelnen Regionen mit der Formel $1/2(f(p)+f(q))\delta$ berechnen lässt. Falls im Vorhinein die Breite $\delta$ des Subintervalls bekannt ist, um die erforderliche Genauigkeit zu erreichen, können die Prozesse für die Berechnung der einzelnen Regionen statisch zugeteilt werden. Bei *p* Prozessoren ist die Größe der Region für jeden Prozessor *(b-a)/p*. Da die Berechnung immer in der selben Form ist, kann eine SPMD-Modell verwendet werden.

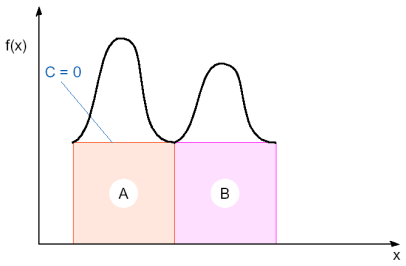



***Adaptive Quadratur:***

Ist die Größe $\delta$ des Subintervalls für die Erreichung der geforderten Genauigkeit nicht bekannt, so müssen Methoden verwendet werden, die iterativ $\delta$ solange verkleinern, bis die gewünschte Genauigkeit erreicht wird.



Das Intervall $\delta$ wird z.B. wie in nebenstehender Abbildung in 3 Bereiche A, B, und C aufgeteilt und dann solange verkleinert bis der größere der beiden Bereiche A und B „ausreichend“ gleich groß wie die Summe der beiden verbleibenden Bereiche ist oder bis der Bereich C „ausreichend“ klein ist. Bei der Wahl der Terminierungsbedingung ist jedoch auch die Kurvenform von f(x) zu berücksichtigen, da starke lokale Varianzen im Kurvenverlauf durchaus zu falschen Terminierungen führen können. Eine andere Methode wäre die sukzessive Halbierung von $\delta$ (erzeugt einen binären Baum) bis zwei aufeinanderfolgende Näherungen „ausreichend“ Nahe (wertmäßig) beieinander liegen.




In jedem Fall sind dynamische Methoden bzw. Lastverteilungs-Methoden für die Prozesszuweisung erforderlich, um eine möglichst effiziente Nutzung des parallelen Systems zu gewährleisten.

### 4.2.3 N-Body Problem

Das N-Body Problem beschäftigt sich mit der Bestimmung von Kräften zwischen Körpern im astronomischen aber auch im molekularen Bereich.

#### *N-Body Gravitations-Problem:*

Die Aufgabe besteht darin, für Körper im Raum (z.B. Planeten), welche gravitativen Kräften ausgesetzt sind, Position und Bewegung gemäß den Newton’schen Gesetzen zu ermitteln. Die Gravitationskraft zwischen zwei Körpern mit dem Massen $m_a$ und $m_b$ ist dabei gegeben durch $F = \frac{Gm_a m_b}{r^2}$, mit *G* als Gravitationskonstante und *r* als Abstand zwischen den Körpern. Aufgrund dieser Kraft werden die Körper gemäß Newtons zweitem Gesetz beschleunigt gegeben durch $F = ma$ mit der Beschleunigung $a = \frac{dv}{dt}$. Für eine Computer-Simulation kann die Berechnung natürlich nur für diskrete Zeitwerte in möglichst kleinen Intervallen durchgeführt werden um so eine möglichst gute Näherung zu erreichen. Bei einem Intervall $\Delta t$ folgt dann zum Zeitpunkt *t+1* für $F = \frac{m(v_{t+1} - v_t)}{\Delta t}$ und damit für $v_{t+1} = v_t + \frac{F\Delta t}{m}$, woraus sich für die Positionsänderung $x_{t+1} = x_t + v\Delta t$ ergibt (bezogen auf eine Dimension). Nachdem sich die Körper in die neue Position bewegt haben, ändert sich damit auch die Kraft und damit die Beschleunigung und die Berechnung muss wiederholt werden. Da die Geschwindigkeit über das Zeitintervall $\Delta t$ nicht konstant bleibt, ist das Ergebnis immer nur eine Näherung. Eine Verbesserung dieser Situation bringt die Berechnung von Geschwindigkeit und Position zu abwechselnden Zeitpunkten („leap-frog“ Methode), also z.B. Berechnung der Positionen zu den Zeitpunkten *t, t+1, t+2* usw. und jene der Geschwindigkeit zu den Zeitpunkten *t+0.5, t+1.5, t+2.5* usw. Für Berechnungen im 3D-Raum ist entsprechend für alle 3 Vektoren die Berechnung durchzuführen.

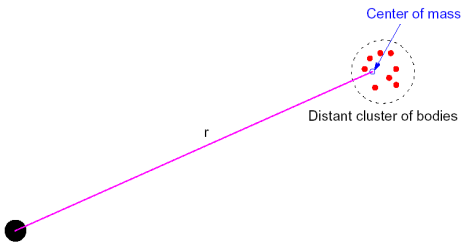
#### *Sequentieller Code:*

```
for (t = 0; t < tmax; t++)            /* for each time period */
   for (i = 0; i < N; i++) {          /* for each body */
      F = Force_routine(i);           /* compute force on ith body */
      v[i]new = v[i] + F * dt / m;    /* compute new velocity */
      x[i]new = x[i] + v[i]new * dt;  /* and new position */
   }
```

```
    for (i = 0; i < N; i++) {          /* for each body */
        x[i] = x[i]new;                /* update velocity & position*/
        v[i] = v[i]new;
    }
}
```

***Paralleler Code:***

Die Parallelisierung der sequentiellen Lösung könnte durch einfache Partitionierung erfolgen, wobei jeder Prozessor für eine Gruppe von Körpern zuständig ist und die berechneten Kräfte mit den anderen Prozessoren kommuniziert werden. Die Zeitkomplexität dieses Ansatzes bei $N$ Körpern beträgt jedoch für eine Iteration $O(N^2)$ *(jeder der N Körper wird von den anderen N-1 Körpern beeinflusst).*




Deshalb wird ein anderer Ansatz verwendet, der davon ausgeht, dass ein Cluster von entfernten Körpern näherungsweise als ein einziger Körper betrachtet werden kann. Der *Barnes-Hut-Algorithmus* ist ein Divide-and-Conquer Algorithmus, der diesen Ansatz aufgreift. Es wird dabei der gesamte Raum als ein Kubus aufgefasst, der alle zu betrachtenden Körper enthält. Dieser Kubus wird nun in 8 Subkuben unterteilt, wobei jene Subkuben, die keine Körper mehr enthalten, entfernt werden. Die verbleibenden Subkuben, die mehr als einen Körper enthalten, werden wieder je in 8 kleiner Kuben unterteilt, solange, bis jeder Subkubus nur mehr einen Körper enthält. Durch diesen Teilungsprozess wird also ein oktärer Baum generiert, dessen Blätter jeweils Zellen mit einem Körper repräsentieren. *(Bei einem 2D-Raum erhält man entsprechend einen quartären Baum).* Nach der Baumkonstruktion ist die Gesamtmasse und die Schwerpunktkoordinaten jedes Subkubus in den jeweiligen Knoten des Baumes gespeichert. Die Kraft die auf jeden Körper wirkt, kann nun durch einfaches Durchlaufen des Baumes von der Wurzel beginnend abwärts ermittelt werden. Bei jedem Knoten wird geprüft, ob die Cluster-Näherung angewandt werden kann, indem berechnet wird, ob $r \geq \frac{d}{\theta}$, mit $d$ als Seitenlänge des Subkubus und typischerweise $\theta \leq 1.0$ als Konstante. Wenn ja wird gestoppt, ansonsten der Baum weiter nach unten durchlaufen. Nachdem für alle Körpern eine neue Position und Geschwindigkeit berechnet wurde, wird der komplette Ablauf inkl. Baumaufbau für jedes Zeitintervall neu gestartet. Baumaufbau und Berechnung aller Kräfte in einem Durchlauf benötigen je $O(N \log N)$ was auch für die gesamte Ausführung gilt.

Der oktäre Baum wird i.A. stark unbalanciert sein, deshalb ist auch eine recht einfache statische Aufteilungsstrategie, wie oben beschrieben, zur Lastverteilung nicht besonders zielführend. Bessere Ergebnisse bez. Lastverteilung erzielt man mit der „orthogonal recursive bisection"-Methode. Dabei wird der Raum zuerst in 2 Hälften mit annähernd gleich vielen Körpern geteilt. Die beiden neuen Teilräume werden nun wieder nach den gleichen Bedingungen geteilt, solange, bis genau so viele Teilräume entstanden sind, wie Prozessoren für die Bearbeitung vorhanden sind. Damit sind, im Gegensatz zum *Barnes-Hut* Algorithmus, keine Prozessoren „arbeitslos"

# 8. Programmierung mit Shared Memory

## 8.1 Shared Memory Multiprozessoren

siehe auch 1.3.1
Neben den in Kap. 1.3.1 erwähnten Möglichkeiten für die Programmierung von SM-Multiprozessor Systemen gibt es noch folgende Alternativen:

- Verwendung einer bestehenden sequentiellen Programmiersprache zusammen mit speziellen Compiler-Anweisungen für die Parallelisierung
- Verwendung eines spez. Compilers zur Erzeugung parallelen Codes aus sequentiellem Code
- Verwendung spezieller Library-Routinen zusammen mit einer bestehenden sequentiellen Programmiersprache (PVM, MPI)
- Verwendung sog. „heavyweight"-Prozessen (siehe UNIX)
- Verwendung von Threads

## 8.2 Konstrukte zur Spezifizierung von Parallelität

### 8.2.1 Erzeugung paralleler Prozesse

Wahrscheinlich die ersten Konstrukte zur Erzeugung paralleler Prozesse sind die *FORK-JOIN* Befehle, welche als Erweiterung zu Fortran und UNIX eingesetzt wurden. Mit *FORK* wurde ein neuer Pfad für einen parallelen Prozess erzeugt. Wenn alle parallelen Prozesse auf ihren *JOIN* Befehl aufgelaufen sind, wird die Abarbeitung wieder in sequentieller Weise weitergeführt. *FORK/JOIN* ist im Wesentlichen das selbe wie das *spawn/exit* Konstrukt in Message Passing Systemen.

#### *UNIX Heavyweight Prozesse:*

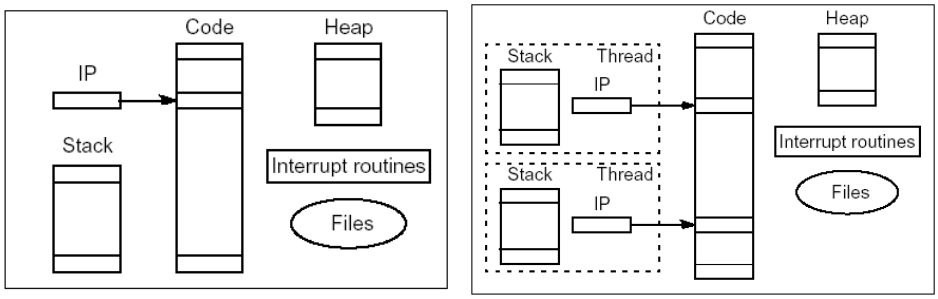
Der UNIX Systemaufruf *fork()* erzeugt einen neuen Prozess (Kind-Prozess) der eine exakte Kopie des aufrufenden Prozesses ist, außer dass er seine eigene eindeutige Prozess-ID hat. Ebenso hat er seine eigene Kopie der Variablen des Eltern-Prozesses mit den Werten, die diese Variablen zum Zeitpunkt des *fork()*-Aufrufes hatten. Bei erfolgreichem *fork()* liefert dieser an den Elternprozess die Prozess-ID des erzeugten Kind-Prozesses zurück und an den Kind-Prozess den Wert 0. Somit kann im weiteren Code zwischen Kind- und Elternprozess unterschieden werden. Es kann also ein SPMD-Programmmodell verwendet werden. Über die Systemaufrufe *wait()*, *waitpid()* im Elternprozess bzw. *exit()* im Kind-Prozess können die Prozesse wieder zusammengeführt werden.

Der mit *fork()* erzeugte Prozess ist also ein komplett separates Programm mit eigenen Variablen, Stack und Speicherallozierung.

### 8.2.2 Thread

Da die Erzeugung kompletter Prozesse sehr zeit- und speicherraubend ist, gibt es mit den Threads eine effiziente Alternative, parallele Abarbeitungspfade zu generieren. Threads haben zwar ihren eigenen Stack und IP, teilen sich jedoch sonst den Speicherbereich und globale Variablen mit dem Prozess in dem sie eingebettet und von dem sie generiert worden sind. Ein weiterer Vorteil von Threads ist die effizientere Synchronisierbarkeit über globale Variable im Gegensatz zu zeitraubenden Systemaufrufen bei der Prozesssynchronisierung. Multithreading wird z.T. in modernen OS wie WinXP od. SUN Solaris eingesetzt um effizient gleichzeitige Aktivitäten (z.B. I/O's) in den Griff zu bekommen oder als wirkungsvoller Mechanismus für das latency-hiding in Message-Passing-Systemen um z.B. schnell auf den nächsten Thread weiterzuschalten, wenn der aktuelle geraden auf eine Nachricht wartet. Mit

der POSIX Pthread-Library gibt es mittlerweile einen bereits von vielen Systemen unterstützten Standard.




*Note: Wenn ein Thread blockiert, blockiert auch der zugehörige Prozess ausser es handelt sich um einen Kernel-Thread.*

### *Ausführen eines Pthread Thread:*

Unter Pthread wird das Hauptprogramm selbst als Thread angesehen. Weitere Threads können mit dem Befehl *pthread_create()* erzeugt werden und mit *pthread_join()* kann auf die Terminierung eines anderen Thread gewartet werden. Bei der Erzeugung können dem neuen Thread über einen *void** Daten übergeben werden und dieser kann über den selben Mechanismus bei Terminierung Daten an den mit *pthread_join()* wartenden Thread Daten zurückgeben.

### *Detached Threads:*

Ist es für den erzeugenden Thread belanglos, wann der erzeugte Thread terminiert, so muss ersterer nicht mit *pthread_join()* auf letzteren warten. Der erzeugte Thread wird dann als *detached* bezeichnet.

### *Thread Pools:*

Hier sind die selben Konfiguration wie bie Prozessen möglich, also

- Master-Slave Konfigurationen
- Thread Pools
- Inter-Thread-Kommunikation über shared memory oder über *signals*

### *Ausführungsreihenfolge der Befehle:*

In einem Single-Prozessor System hängt diese bei parallelen Pfäden von der Scheduler-Strategie des OS ab, obwohl ein Thread typischerweise bis zum Ende durchläuft falls er nicht blockiert. Bei Multiptozessor-Systemen können parallele Prozesse/Threads auf verschiedenen Prozessoren laufen und sich je nach Prozessorleistung oder sontigen lokalen Systemeigenschaften zeitlich überlappen. Darüberhinaus kann auch durch den Compiler die Ausführungsreihenfolge geändert werden, um den Code für die entsprechende Prozessorarchitektur zu optimieren. Ebenso ist es bei modernen Superskalar Prozessoren üblich, zur Verbesserung der Ausführungsgeschwindigkeit den Maschinencode „out-of-order“ auszuführen.

### *Thread-Safe Funktionen:*

Wenn Routinen/Funktionen von mehreren Threads simultan aufgerufen werden können ohne dass die Korrektheit des Ergebnisses beeinträchtigt wird, spricht man von Thread-Safe Routinen/Funktionen. Die meisten Funktionen der POSIX Library sind bereits Thread-Safe implementiert. Um diesem Aspekt der Thread-Safety aus dem Weg zu gehen, kann man Routinen/Funktionen z.B. mit *critical sections* oder Eintritts-*Semaphoren* einfach so gestalten, dass immer nur ein Aufrufer zugelassen wird.

# 8.3 Sharing Data

## 8.3.1 Erzeugung von Shared Data

Thread besitzen bereits implizit mit den globalen Variablen des Hauptprogramms einen Shared Data Bereich. Bei Prozessen muss mit verschiedenen Systemaufrufen shared memory alloziert und an den virtuellen Speicherbereich des jeweiligen Prozesses angefügt werden:

- `shmget()` – erzeugt eine Shared Memory Segment und gibt eine ID an den Aufrufer zurück.
- `shmat()` – das über die ID identifizierte Shared Memory Segment wird an den Datenbereich des aufrufenden Prozesses angefügt und gibt die Startadresse des Datensegmentes zurück.

## 8.3.2 Zugriff auf Shared Data

Der Zugriff auf Shared Data bzw. Shared Resourcen i.A. kann durch z.B. geschützte Codebereiche, sog. *critical sections* realisiert werden, wobei für den gegenseitigen Zugriffsausschluss Mechanismen wie *mutexes* (mutual exclusions) verwendet werden.

### *Locks:*

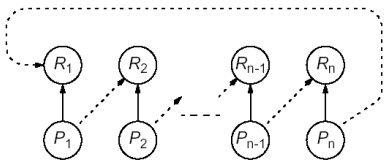
Der einfachste Mechanismus zur Realisierung eines Mutex ist die Verwendung eines *lock*, eine 1-bit Variable. Wartet ein Prozess in einer Schleife auf die Freigabe eines locks so wird dies als *busy waiting* auf einen *spin lock* bezeichnet. Da *busy waiting* eine äußerst ineffiziente Methode ist, ist es besser, spez. *wait*-Funktionen des OS zu verwenden (falls vorhanden), die den wartenden Prozess „schlafen legen" bis der *lock* frei wird. Damit werden nicht unnötig CPU-Resourcen verschwendet. Wenn mehrere Prozesse auf das Freiwerden eines *locks* warten, muss eine Priorisierungsmöglichkeit gegeben sein, nach der *lock* einem der wartenden Prozesse zugeteilt wird.

Um zu Verhindern, dass aufgrund von race conditions der lock von mehreren Prozessen gleichzeitig belegt wird, muss die Abfrage des *lock state* und die Veränderung dessen *atomar* erfolgen. Dafür bieten die meisten Prozessoren spez. unteilbare Maschinenbefehle, sog. *test-and-set*-Befehle, es ist aber auch eine Realisierung atomarer Zugriffe ohne HW-Unterstützung möglich (siehe SmartCards OS).

Die Pthread-Library stellt dafür spezielle *pthread_mutex* Befehle zur Verfügung, um solche *locks* bzw. *mutexes* zu realisieren.

### *Deadlock:*

Ein *deadlock* kann auftreten, wenn z.B. zwei Prozesse die jeweils vom anderen Prozess gehaltene Resource belegen wollen. Ein *deadlock* kann auch auf circuläre Art zustande kommen *(siehe nebenstehende Abbildung)* -> *deadly embrace*. Ein *deadlock* zwischen zwei Prozessen kann verhindert werden, indem beide Prozesse in der selben Reihenfolge auf die Resourcen zugreifen oder indem mit *try_lock*-Funktionen gearbeitet wird, die eine Resource nur belegen wenn sie frei ist und ansonsten mit einer Fehlermeldung zurückkehren.




### *Semaphoren:*

Sind positive Ganzzahlvariablen die mit speziellen atomaren Funktionen inkrementiert od. dekrementiert werden können. Im ursprünglichen Konzept von Dijkstra wurden diese Funktionen mit *P* (derkrementieren wenn Sem. > 0 -> belegen) und *V* (inkrementieren -> freigeben) benannt, in aktuellen Implementierungen werden diese meist mit *down(s)* und *up(s)* bezeichnet. Ist die Semaphore>0, so ist sie frei und kann von einem Prozess mit *down(s)*

belegt werden. Sie muss von diesem Prozess auch wieder mit *up(s)* freigegeben werden. Mutexes können mit binären Semaphoren realisiert werden, die nur die Werte 0 und 1 annehmen können. Die up und down Funktionen beinhalten auch einen Prozess-Scheduling Mechanismus der blockierte Prozesse schlafen legt und diese beim Freiwerden der Semaphore auch wieder weckt. Für *up/down* gibt es verschiedenste Variante, wie z.B. eine Timeout-Variante, so dass der schlafende Prozess spätestens nach einem Timeout wieder geweckt wird unabhängig davon, ob die Semaphore frei ist.

#### *Monitor:*

Semaphore, Zugriffs- und Verwaltungsfunktionen werden in einer Struktur bzw. einem Objekt zusammengefasst. Somit kann über das Objekt die korrekte Verwendung der Semaphore gesteuert werden und damit fallen Programmierfehler wie ein vergessenes Release nach einer Belegung weg. Solch ein Monitor-Konzept existiert z.B. in Java.

#### *Condition Variable:*

Damit kann z.B. die Abarbeitung einer *critical section* an eine Bedingung in Form einer *condition variable* geknüpft werden. Typische Funktionen auf solche condition variable sind

- `wait(cond_var)` - warten bis Bedingung eintritt
- `signal(cond_var)` - signalisieren dass Bedingung eingetreten ist
- `status(cond_var)` - gibt die Anzahl der Prozesse zurück, die auf die Bedingung warten

Die Pthread-Lib z.B. stellt *condition variable* zur Verfügung, die mit einem bestimmten Mutex verbunden sind.

Bei der Signalisierung ist zu beachten, dass Signal vom OS nicht gespeichert werden, d.h. wenn das Auftreten einer Bedingung signalisiert wird bevor ein anderer Prozess darauf wartet und dieser erst nach der Signalisierung das *wait* aufruft, kommt er erst mit der nächsten Signalisierung wieder raus!

#### *Barrieren:*

Für die Synchronisierung von Prozessen/Threads werden sog. Barrieren eingesetzt, also Synchronisierungspunkte im Code die nur dann überwunden werden können, wenn alle Synchronisierungspartner ebenfalls diese Barrieren erreicht haben. Dabei muss durch die Implementierung der Barriere berücksichtigt werden, dass manche Prozesse/Threads eine Barriere schon das zweite mal betreten können wobei andere noch immer das erste mal in der Barriere sind. Dies kann mit einem Design, das Eintritts- und Austrittsphase separat behandelt, realisiert werden.

Die Pthread-Lib stellt keine eigenen Funktionen für Barrieren zur Verfügungen, diese müssen als aus eine Kombination von *conditon variable* und *mutex* realisiert werden.

## 8.4 Parallele Programmiersprachen und Konstrukte

### 8.4.1 Sprachen

Es wurden über die Jahre viele spezialisierte Sprachen für die parallele Programmierung entwickelt, aber nur wenige haben eine nennenswerte Akzeptanz erfahren. Einzig HPF (High Performance Fortran) ist noch immer in einem nennenswerten Ausmaß im Einsatz. Jüngere Entwicklungen wie z.B. UPC (Unified Parallel C – www.gwu.edu/~upc) scheinen etwas mehr Chancen für eine breitere Akzeptanz zu besitzen, vorallem wenn ein größeres Konsortium von Wirtschaft und Wissenschaft als Lobby dahinte steht so wie bei UPC.

## 8.4.2 Sprachkonstrukte

### *Shared Data:*

Es sollte ein Konstrukt für die Deklaration von *shared* Variablen vorhanden sein wie z.B. das `shared` keyword in UPC. Gleichzeitig müssen dann auch Mechanismen für den sicheren Zugriff auf `shared` Daten implementiert sein. UPC bietet auch noch die Möglichkeit Arrays zu deklarieren, dessen Elemente über verschiedene Threads verteilt sind.

### *`par` Construct:*

Damit kann spezifiziert werden, dass Statements/Aufrufe/Prozesse parallel abgearbeitet werden sollen, also *Instruction-Level* Parallelität:

```
par { S1; S2; ... Sn; }
```

### *`forall` Construct:*

Damit besteht die Möglichkeit, mehrere ähnliche Prozesse gleichzeitig zu starten:

```
forall (i=0; i<p; i++) { a[i]=0; }
```

startet `p` Prozesse die alle die Statements `a[i]=0` parallel ausführen, wobei jeder Prozess einen anderen Wert für `i` erhält, also würden die Arrayelemente `a[0]...a[p-1]` alle „*gleichzeitig*“ auf 0 gesetzt werden.

## 8.4.3 Analyse der Abhängigkeiten

Eine der Schlüsselaufgaben in der parallelen Programmierung ist die Identifizierung jener Prozesse, die zusammen ausgeführt werden können, wo also keine Abhängigkeiten untereinander bestehen. Da dies aber nicht immer offensichltich ist, wird ein Algorithmus benötigt, um solche Abhängigkeiten zu erkennen, der günstigerweise auch von einem parallelisierenden Compiler verwendet werden kann.

### *Bernstein's Bedingungen:*

Bernstein definierte einen Satz von Bedingungen die ausreichend sind um festzustellen, ob zwei Prozesse gleichzeitig ausgeführt werden können. Diese Bedingungen beziehen sich auf die Speicherbereiche der Variablen, die von den parallelen Prozessen während der Ausführung gelesen und geschrieben werden. Dabei gibt es zwei Sets von Speicherbereichen,

- $I_i$ ist ein Satz von Speicherplätzen, die vom Prozess $P_i$ gelesen werden
- $O_j$ sind jene Speicherplätze, die vom Prozess $P_j$ verändert werden

Für zwei parallel ausgeführte Prozesse muss nun gelten, dass

$$I_1 \cap O_2 = \phi \quad \text{und} \quad I_2 \cap O_1 = \phi ,$$

mit $\phi$ als leeres Set. Ausserdem muss das Ausgabeset der Prozesse verschieden sein, also

$$O_1 \cap O_2 = \phi .$$

Diese Bedingungen können auf Prozesse mit beliebiger Komplexität angewandt werden genauso wie zur Identifizierung möglicher Parallelitäten auf Befehlsebene.

Eine Möglichkeit um Abhängigkeiten in Schleifen zu beseitigen und sie damit parallelisierbar zu machen ist die Aufteilung in mehrere kleinere Schleifen die voneinander unabhängig sind -> *loop decomposing*.

## 8.6 Performance Issues

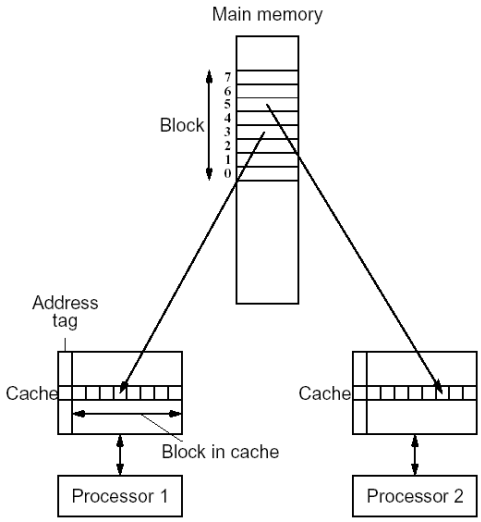
### 8.6.1 Zugriff auf shared Daten

Gerade bei modernen Prozessoren mit mehreren Cache-Levels ergibt sich das Problem der Datenkohärenz und –konsistenz in einem Shared-Memory-System. Um sicher zu stellen, dass bei der Änderung von Daten im Cache eines Prozessors nachfolgende Referenzierungen dieser Daten im Shared-Memory durch einen anderen Prozessor auch die aktuellen Daten betreffen, werden sog. Cache-Kohärenzprotokolle eingesetzt (siehe 1709 KE4, S.31ff). Diese verwenden meist eine der folgenden Strategien:

- *Write-update*-Strategie: Beim Verändern einer Kopie in einem Cache müssen alle Kopien in anderen Caches ebenfalls verändert werden.
- *Write-invalidate*-Strategie (häufiger bei SMP): Vor dem Verändern einer Kopie in einem Cache müssen alle Kopien in anderen Caches als „*invalid*“ markiert werden. Eine *invalid* markierte Kopie wird dann automtisch mit dem Inhalt aus dem Hauptspeicher aufgefrischt, wenn sie vom Prozessor referenziert wird.

Die Einhaltung der Datenkohärenz bedeutet natürlich eine gewisse Performanceeinbuße des Systems. Allerdings könnte man wieder den Umstand nutzen, dass immer ganze Cachelines in den Cache geladen werden und nicht einzelne Bytes. Damit bekommt man allerdings eine starke Abhängigkeit von der jeweiligen Prozessorarchitektur in das SW-System.

Durch das Cachelining kann es aber auch zu unnötigen Performanceeinbussen aufgrund des Cache-Kohärenzprotokolls kommen. Zwei Prozessoren könnten in verschiedenen Bereichen des gleichen Cacheblockes arbeiten. Wenn nun einer seinen Teil des Blockes verändert wird automatisch die betreffende Cacheline im Cache des anderen Prozessors als invalid markiert und dieser muss bei einem neuen Zugriff auf seinen Teil des Blocks wieder die Cacheline mit den Daten aus dem Hauptspeicher auffrischen, obwohl beide Prozessoren eigentlich an verschiedenen, non-shared Daten gearbeiten haben -> *false sharing*. Eine mögliche Lösung dafür wäre, dass der Compiler Daten, die nur von einem Prozessor geändert werden, bzw. alle non-shared Daten, in verschiedenen Blöcken abzulegen, was natürlich nicht immer einfach möglich ist. Bei der Programmierung sollte also die Caching Charakteristik möglichst ausgenutzt werden und *false sharing* soweit als möglich reduziert werden ohne aber dabei zu große HW-Abhängigkeiten in den Code zu bringen.




# 10. Sortier-Algorithmen

## 10.1 Allgemeines

### 10.1.1 Sortierung

Viele parallele Sortier-Algorithmen und parallele Implementierungen von sequentiellen Sortier-Algorithmen sind synchronisierte Algorithmen, in dem eine Gruppe von Aktionen erst beendet sein muss bevor die nächsten Aktionen beginnen können.

### 10.1.2 Speedup Potential

$O(n \log n)$ ist die untere Grenze für die Zeitkomplexität der compare-and-exchange Sortier-Algorithmen wie z.B. Quicksort oder Mergesort, wenn keinen besonderen Voraussetzungen an die zu sortierenden Elemente gestellt werden. Damit ergibt sich bei p Prozessoren eine optimale parallele Zeitkomplexität von $\frac{O(n \log n)}{p} = O(\log n)$ *für* $p = n$.

## 10.2 Compare-and-Exchange Sortier-Algorithmen

### 10.2.1 Compare and Exchange

Es werden dabei jeweils zwei Werte miteinander verglichen (compare) und gegebenenfalls vertauscht (exchange) um die Sortierrichtung (auf-/absteigend) herzustellen. In einem Message Passing System lässt sich das recht gut realisieren, in dem zwei Prozesse $P_1$ u. $P_2$ jeweils einen der zu vergleichenden Werte z.B. *A* u. *B* halten. Beide Prozesse senden nun dem jeweils anderen Prozess den eigenen Wert und vergleichen dann den eigenen mit dem Wert des anderen Prozess. $P_1$ behält dann z.B. den kleineren der beiden Werte und $P_2$ den größeren (bei aufsteigender Sortierung). *-> um einen Deadlock zu vermeiden, beginnt ein Prozess zuerst mit dem Senden seines Wertes und der andere mit dem Empfangen.*

#### *Bemerkung zur Präzision von duplizierten Berechnungen:*

Beim compare findet normalerweise ein logischer Vergleich der Form A > B statt der ein boolsches Ergebnis liefert. Wenn im MPS Prozessoren mit verschiedenen Architekturen eingesetzt werden, kann dieser Vergleich z.B. bei float-Werten durchaus verschiedene Ergebnisse liefern. Dies ist bei der Programmierung zu bedenken.

#### *Data Partitioning:*

Da üblicherweise ein Prozessor nicht nur ein Wertepaar sondern eine Gruppe von *n/p* Werten bearbeiten muss, werden zw. zwei Prozessoren eben Wertegruppen ausgetauscht. Jeder Prozessor verwaltet eine sortierte Wertegruppe. Die eigene Gruppe wird mit den empfangenen Werten verschmolzen (merge) und ein Prozessor sendet z.B. die höheren Werte zurück an den anderen Prozessor und einer die niedrigeren. Voraussetzung dafür ist, dass beide Prozessoren beim Verschmelzen die selben Rersultate liefern und die Gruppen gleich teilen. Das Verschmelzen benötigt *2(n/p)-1* Schritte und zwei Nachrichtentransfers.

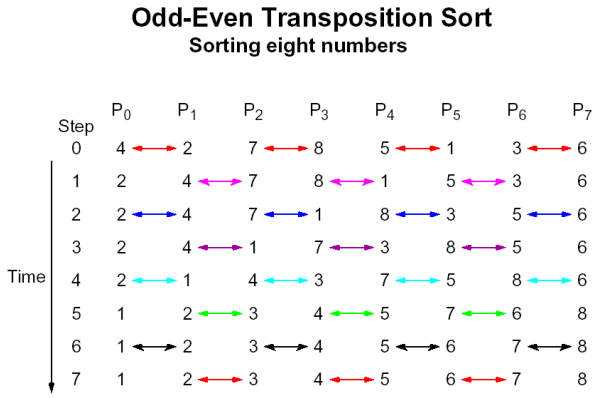
### 10.2.2 Bubble Sort und Odd-Even Transposition Sort

Angenommen es gibt *n* Zahlen $x_0 \ldots x_{n-1}$, so werden zuerst $x_0$ und $x_1$ verglichen und die größere Zahl wandert zu $x_1$, danach $x_1$ und $x_2$ usw. bis die größte der *n* Zahlen $x_{n-1}$ ist. Danach wird der Vorgang wiederholt, allerdings nur bis $x_{n-2}$ usw. Es sind also bei *n* Zahlen *n-1* Vergleiche im ersten Durchlauf, *n-2* im Zweiten usw. notwendig, also insgesamt $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ Durchläufe. Daraus folgt eine Zeitkomplexität von $O(n^2)$ unter der Annahme, das eine einzelne *compare-and-exchange* Operation eine konstante Zeitkomplexität $O(1)$ besitzt.

#### *Paralleler Code – Odd-Even Transposition Sort:*

Eine Parallelisierungsmöglichkeit besteht darin, dass der zweite Durchlauf beginnen kann, bevor der Erste beendet ist unter der Voraussetzung, dass der Erste vom Zweiten nicht überholt wird. Wenn also im ersten Durchlauf das erste Paar bearbeitet wurde kann der zweite Durchlauf bereits mit dem ersten Paar beginnen wenn der erste Durchlauf mit dem zweiten Paar beginnt usw. Eine Pipeline-Architektur wäre also vorteilhaft.

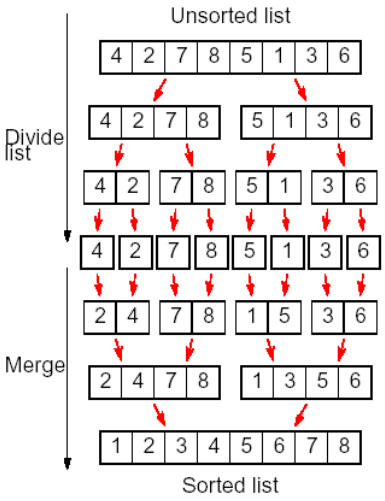
Eine Variation von Bubble Sort welche diese Parallelisierungsmöglichkeit ausnutzt ist die *Odd-Even Transposition Sort*. Dabei gibt es zwei alternierende Phasen, die *odd-* und die *even*-Phase. In der *even*-Phase führen nur Prozesse mit geraden Prozess-IDs den *compare-and-exchange* Vorgang mit ihrem Nachbarprozess durch und in der *odd*-Phase nur die Prozesse mit ungeraden IDs. Auf einem sequentiellen System bringt dieser Algorithmus keinerlei Vorteil gegenüber Bubble Sort aber in der parallelem Implementierung reduziert sich damit die Zeitkomplexität auf $O(n)$ bei $p = n$. *(Im Buch wird zwar angegeben dass dieser Algorithmus optimal für ein Netzwerk mit Bustopologie ist, ich würde aber eher sagen dass ein switched Ethernet geeigneter ist, da immer nur Paare von Prozessoren gleichzeitig miteinander kommunizieren wobei eben diese Kommunikationspaare am selben Switch hängen müssen.)*




Da alle Prozessoren im Prinzip den selben Programmcode benötigen, bietet sich ein SPMD Modell an, wobei die einzelnen Prozessoren anhand ihrer eigenen ID entscheiden, welchen Codeteil sie ausführen.

### 10.2.3 Mergesort

Mergesort ist ein klassische sequentieller Algorithmus der den *divide-and-conquer* Ansatz verwendet. In der *divide*-Phase wird die unsortierte Liste in der Hälfte geteilt und die entstandenen Teilhälften solange weiter in der Hälfte geteilt bis die erhaltenen Teile nur mehr eine Zahl enthalten. In der anschließenden *merge*-Phase werden nun jeweils ein Zahlenpaar zu einer sortierten Liste mit zwei Zahlen verschmolzen, dann werden die erhaltenen Listen wieder paarweise zu Listen mit 4 Zahlen verschmolzen usw. bis letztlich eine Liste mit allen sortierten Zahlen entsteht. Es lässt sich also sehr gut eine Baumstruktur realisieren wie in *Kap. 4.1.2* bereits besprochen.




Ein signifikanter Nachteil der Baumstruktur ist allerdings die schlechte Lastverteilung, da nicht in allen Ebenen des Baumes alle Prozessoren beschäftigt sind.

***Analyse:***

Die sequentielle Zeitkomplexität beträgt $O(n \log n)$. In der parallelen Version sind $2 \log n$ Schritte notwendig, allerdings kann, abhängig von der Anzahl der zu sortierenden Zahlen, in jedem Schritt mehr als eine Operation notwendig sein. Unter der Annahme das die einzelnen Teillisten zu den zugehörigen Prozessoren gesendet werden und das dann das Merging intern durchgeführt wird so ergibt sich für die *divide*-Phase und die *merge*-Phase zusammen für die

*Kommunikation*: $t_{comm} = 2(t_{startup} + \frac{n}{2} t_{data} + t_{startup} + \frac{n}{4} t_{data} + ...)$ also

$t_{comm} \approx 2((\log p) t_{startup} + n t_{data})$ bei $p$ Prozessoren *(log p wegen Baumhöhe)*.

*Computation* ist nur in der *merge*-Phase notwendig, wenn zwei Teillisten zu einer sortierten Liste zusammengefügt werden, wobei durch die Teillisten schrittweise durchgegangen wird und die kleinste Zahl zuerst in die verschmolzene Liste eingefügt wird. Es sind dabei max.

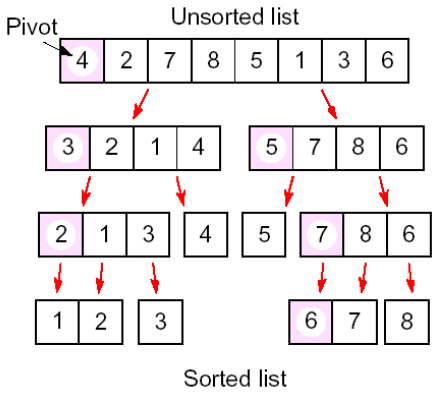
$2n-1$ Schritte notwendig, um zwei Teillisten mit je $n$ Zahlen zu verschmelzen. In obigen Beispiel wären dass zuerst 1 Schritt, dann 3 und dann 7 Schritte. Insgesamt folgt also für

$$t_{comp} = \sum_{i=1}^{\log p} (2^i - 1)$$

Die parallele Zeitkomplexität ist insgesamt also $O(p)$ bei $p$ Prozessoren und $n = p$ Zahlen.

## 10.2.4 Quicksort

Es ist ein Verfahren nach der *divide-and-conquer* Strategie. Es wird dabei ein Element aus der unsortierten Liste als sog. Pivot-Element ausgewählt (z.B. das erste od. das letzte) und dann die Liste in zwei Teillisten zerlegt, wobei in die erste Teilliste alle Elemente eingetragen werden, die kleiner dem Pivot sind, in die zweite Teilliste kommen alle anderen Elemente. Das Pivot selbst kann einer der beiden Teillisten zugeordnet werden od. separiert bleiben. Nun wird mit den Teillisten analog vorgegangen (neues Pivot auswählen und wieder teilen) bis die entstandenen Teillisten nur mehr ein Element enthalten. Danach müssen die Teillisten inkl. der ev. separierten Pivots nur mehr entsprechend zu einer einzigen sortierten Liste zusammengeführt (*merge*) werden.

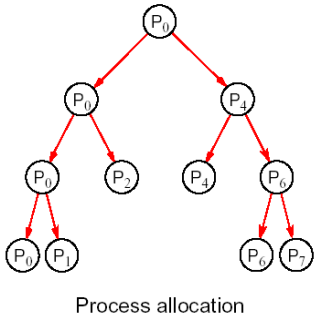



### *Sequentieller Code:*

In der sequentiellen Implementierung wird normalerweise ein rekursiver Algorithmus eingesetzt und die erreichbare durchschnittliche Zeitkomplexität liegt bei $O(n \log n)$ für $n$ Zahlen. Allerdings kann sie bei ungünstiger Wahl des Pivot-Elements (z.B. Pivot ist immer das größte Element) auch bei $O(n^2)$ liegen.

### *Parallelisiertes Quicksort:*

Eine offensichtliche Parallelisierungsmöglichkeit ist, dass eine der beiden entstandenen Teillisten an einen weiteren Prozessor im System zu übergeben. Man erhält also wieder eine Baumstruktur. *(nebenstehende Baumstruktur für obiges Beispiel)*



Process allocation

Das fundamentale Problem mit all diesen Baumstrukturen ist das Problem der ungleichen Lastverteilung. In dem angeführten Beispiel wird die Aufteilung der längsten Liste am Anfang von einem Prozess durchgeführt, je kürzer die Listen werden desto mehr Prozessoren sind beteiligt.

Unter der Annahme einer idealen Wahl des Pivot und dass jede Teilung einer Liste zwei Teillisten gleicher Länge ergeben folgt für die

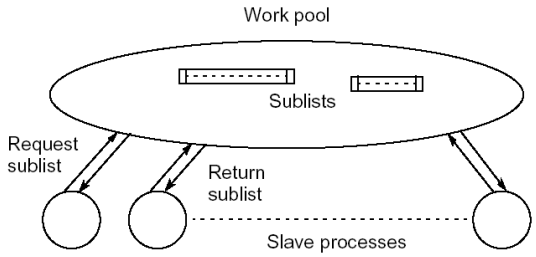
*Kommunikation*: ähnlich wie bei Mergesort $t_{comm} = (t_{startup} + \frac{n}{2} t_{data} + t_{startup} + \frac{n}{4} t_{data} + ...)$ und damit $t_{comm} \approx (\log p) t_{startup} + n t_{data}$ bei $p$ Prozessoren.

*Computation*: Zuerst arbeitet ein Prozessor auf $n$ Elementen, dann 2 auf $n/2$ usw., damit folgt für $t_{comp} = n + n/2 + n/4 + ... \approx 2n$ -> $O(n)$.

Da aber die entstehende Baumstruktur i.A. nicht ideal balanciert ist sondern dessen Form stark von der Wahl des Pivotelements abhängt ist die Pivotwahl natürlich entscheidend für die letztlich erreichbare Zeitkomplexität. Hier gibt es spezielle Auswahlstrategien wie die *3-Median-* od. die *Zufallsstrategie* (siehe [Ottm02] S. 94ff).

***Work-Pool Implemtierung:***

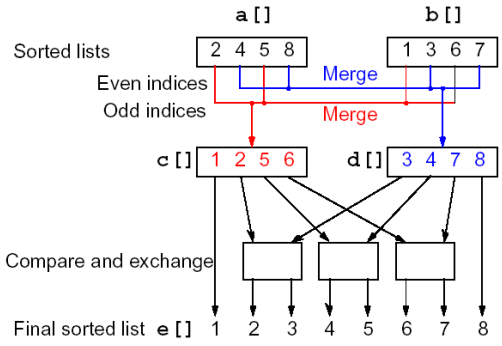
Die Lastverteilung kann auch durch eine dynamische Prozesszuteilung, wie in Kap. 3.2.2 bereits beschrieben, erfolgen. Der Workpool enthält als zuteilbare Aufgaben die Listen bzw. Teillisten die noch weiter zerlegt werden müssen. Mit diesem Ansatz wird zwar das Problem der Idle-Prozessoren nicht eliminiert, allerdings wird dadurch dem Fakt Rechnung getragen, dass längere Listen auch einen höheren Bearbeitungsaufwand erfordern.




## 10.2.5 Odd-Even Mergesort

Batcher stellte 1968 zwei wichtige parallele Sortieralgorithmen vor – den Odd-Even-Mergesort und den Bitonic Mergesort.

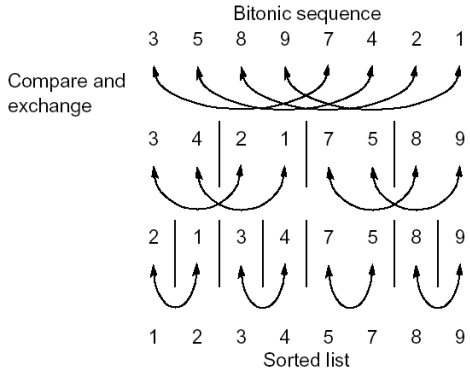
Beim Odd-Even Mergesort werden zwei sortierte Liste zu einer zusammengeführt. Rekursiv angewendet können so alle entstandenen Teillisten zu einer sortierten finalen Liste verschmolzen werden. Es werden dabei die Element mit ungeraden Indizes der beiden Listen *A* und *B* in eine neue Liste *C* einsortiert, ebenso die Elemente mit geraden Indizes in eine neue Liste *D*. Die endgültige sortierte Liste *E* erhält man dann mit $e_{2i} = \min\{c_{i+1}, d_i\}$ und $e_{2i+1} = \max\{c_{i+1}, d_i\}$ für $1 \le i \le n-1$ bei *n* Zahlen. Das erste Element der Liste *E* ist gegeben durch $e_1 = c_1$ (also $a_1$ od. $b_1$) und das letzte Element durch $e_{2n} = d_n$ (also $a_n$ od. $b_n$).




Die erreichbare Zeitkomplexität beträgt $O(\log^2 n)$ bei *n* Prozessoren. In diesem Fall können immer 2 Prozessoren die compare-and-exchange Operationen auf den beiden Listen ausführen.

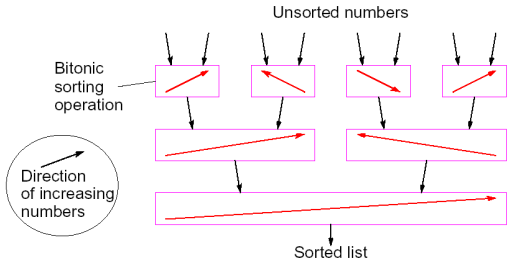
## 10.2.6 Bitonic Mergesort

Die Basis von Bitonic Mergesort ist eine bitonische Sequenz einer Liste, also es gibt zwei Sequenzen der Werte, eine aufsteigende und eine absteigende. Hat man nun eine bitonische Sequenz der Art $a_0 < a_1 < a_2 \ldots < a_{i-1} < a_i > a_{i+1} > \ldots > a_{n-2} > a_{n-1}$ mit $0 \le i < n$ so erhält man nach Ausführung der *compare-and-exchange* Operation auf $a_i$ mit $a_{i+n/2}$ für alle $0 \le i < n/2$ zwei bitonische Sequenzen, wobei in einer Sequenz alle Werte kleiner sind als jene in der zweiten Sequenz. Es wird also die kleinere Zahl jedes Wertepaares in die linke Sequenz verschoben und die größere in die rechte. Rekursiv angewandt erhält man letztlich wieder Teillisten mit ev. nur mehr einem Element die dann zu einer sortierten Liste verschmolzen werden können.



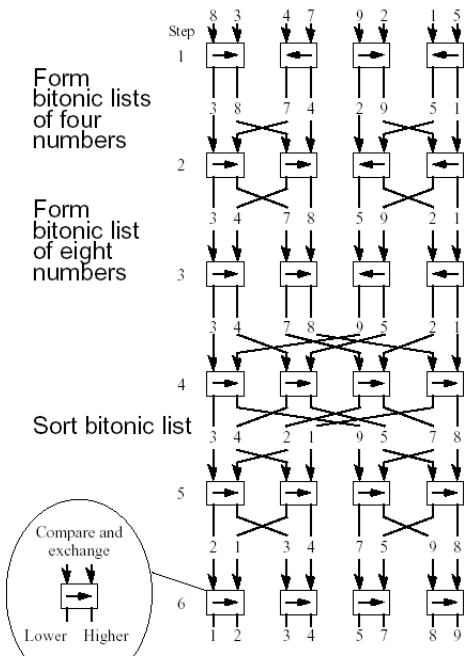

***Sortieren:***

Um eine unsortierte Liste zu sortieren, werden zuerst jeweils Paare benachbarter Werte so aufsteigend bzw. absteigend sortiert, dass zwei benachbarte Wertepaare eine bitonische Sequenz von 4 Werten bilden. Diese 4-Werte Sequenz wird



nun mit der benachbarten Sequenz zu einer bitonischen 8-Werte Sequenz geformt usw. bis man letztlich aller Werte in einer einzigen bitonischen Sequenz sortiert hat. Diese finale bitonische Liste wird dann abschließend in eine monotone Liste sortiert.
Beispiel mit 8 Zahlen in 3 Phasen:

1. *(Step 1):* Wertepaare werden in aufsteigende/absteigende Sequenzen gebracht (4-bit bitonische Sequenzen).
2. *(Step 2/3):* Jede 4-bit Sequenz wird in eine 2-bit Sequenz gesplittet, Sequenzen mit größeren Werten in der Mitte.
   Jede 4-bit bitonische Sequenz wird in eine aufsteigende/absteigende monotone Sequenz sortiert und zu einer 8-bit bitonischen Sequenz verschmolzen.
3. *(Step 4/5/6):* Sortierung der 8-bit bitonischen Sequenz in eine monotone Sequenz.




Bei $n = 2^k$ Zahlen ergeben sich $k$ Phasen mit jeweils max. $k$ Steps. Für die Gesamtanzahl der Schritte ergibt sich

$$Steps = \sum_{i=1}^{k} i = \frac{k(k+1)}{2} = \frac{\log n(\log n + 1)}{2} = O(\log^2 n)$$ mit $n$ Prozessoren. Wie bei allen Sortieralgorithmen können die Daten natürlich partitioniert werden um die Anzahl der benötigten Prozessoren zu reduzieren, was allerdings einen Anstieg der notwendigen internen Bearbeitungsschritte bedeutet. Bitonic Mergesort kann auf eine Mesh- oder auch Hypercube-Topologie appliziert werden. Außerdem lässt er sich auch recht gut in Hardware implementieren.

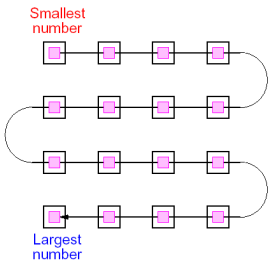
## 10.3 Sortieren auf speziellen Netzwerken

Algorithmen spezialisiert auf bestimmte Netzwerktypen sind heute allerdings nicht mehr so interessant, da dem Programmierer oft die Netzwerktopologie des parallelen Systems gar nicht mehr bekannt ist. Na ja...

### 10.3.1 2D-Sortieren

Werden die Zahlen auf ein Netzwerk mit mesh-Topologie übertragen, so existieren bestimmte Möglichkeiten die Netzwerktopologie für die Sortierung zu nutzen.

Einerseits könnte das Layout einer sortierten Sequenz auf einem Mesh Zeile für Zeile sein, also *schlangenförmig*. Damit können eindimensionale Sortieralgorithmen wie z.B. *Odd-Even Transposition Sort* verwendet werden und man erreicht eine Zeitkomplexität von $O(n)$, was aber noch nicht die aktuelle untere erreichbare Grenze von $2\sqrt{n} - 1$ Schritten, also $O(\sqrt{n})$, für das Mesh darstellt.

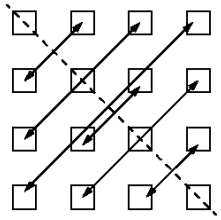



Ein besseres Ergebnis liefert da *shearsort*, womit nur $\sqrt{n}(\log n + 1)$ Schritte ($O(\sqrt{n}\log n)$ für $n$ Zahlen auf einem $\sqrt{n} \times \sqrt{n}$ Mesh benötigt werden. Es werden dabei zuerst die $n$ Zahlen auf das Mesh übertragen. Dann erfolgt in mehreren Phasen die Sortierung, wobei in den

- *ungeraden Phasen* jede Zeile unabhängig von den anderen Zeilen in abwechselnden Richtungen sortiert wird:
  - gerade Zeilen – die kleinste Zahl wird ganz rechts positioniert
  - ungerade Zeilen – die kleinste Zahl wird ganz links positioniert
- *geraden Phasen* jede Spalte unabhängig von den anderen Spalten sortiert wird, wobei das kleinste Element jeder Spalte am oberen Ende jeder Spalte steht.

Nach $\log n + 1$ Phasen sind alle Zahlen sortiert und in Schlangenform links oben beginnend über das Mesh positioniert *(wegen der abwechselnden Richtung bei der Zeilensortierung)*. Für die Sortierung der Zeilen und Spalten kann jeder Sortieralgorithmus verwendet werden.

#### *Verwendung von Transposition:*



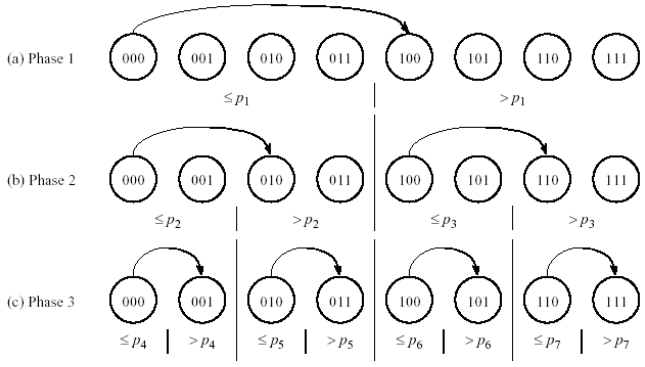
Algorithmen wie shearsort mit abwechselnder Sortierung von Zeilen und Spalten können auch auf die Sortierung nur der Zeilen beschränkt werden, wenn zwischen der Zeilen- und der Spaltensortierung Die Spalten auf Zeilen transponiert werden. Bei einem 2D-Feld (Matrix) $a_{ij}$ mit $0 \le i < n, 0 \le j < n$ werden alle Elemente unterhalb der Diagonale der Matrix mit den Elementen oberhalb vertauscht, also $a_{ij} = a_{ji}$. Wird ein Prozessor je Zeile verwendet, also $\sqrt{n}$ Prozessoren für ein $\sqrt{n} \times \sqrt{n}$ Feld, dann benötigt jeder Prozessor für die Sortierung seiner Zeile $O(\sqrt{n} \log \sqrt{n})$. Die Transposition benötigt $\sqrt{n}(\sqrt{n} - 1)$ Kommunikationen, also $O(n)$. Die Daten werden immer zwischen Paaren von Prozessoren ausgetauscht, wobei jeder Austausch 2 Kommunikationen erfordert. Eine *all-to-all* Kommunikationsroutine reduziert natürlich den Aufwand. Die gesamte Zeitkomplexität der Sortierung im Mesh wird letztlich aber durch die Transposition dominiert und liegt deshalb bei $O(n)$.

### 10.3.2 Quicksort auf einem Hypercube

Das Hypercube Netzwerk bietet durch seine strukturelle Eigenschaft die Möglichkeiten für die Implementierung effizienter *divide-and-conquer* Algorithmen bieten.

#### *Platzierung der kompletten Liste in einem Prozessor:*

Die unsortierte Liste wird zu Beginn in einem Knoten des Hypercube von diesem nach Bestimmung eines Pivot geteilt und eine der beiden Teillisten an einen benachbarten Knoten in der höchsten Dimension übergeben. Dann wird von diesen beiden Knoten mit den beiden Teillisten der Vorgang wiederholt, wobei wieder eine Teilliste an einen benachbarten Knoten in der nächst höheren Dimension übergeben usw. Immer kleiner werdende Teillisten breiten sich so über den Hypercube aus. Wenn alle Knoten mit Listen versorgt sind werden sie mit sequentiellen Algorithmen sortiert und ev. wieder an einen Knoten retourniert der sie dann zu einer einzigen sortierten Liste zusammenfügt.




#### *Initiale Verteilung der Zahlen über alle Prozessoren:*

Alle unsortierten Zahlen werden zu Beginn möglichst gleichmäßig auf alle Knoten eines *d*-dimensionalen Hypercube verteilt. Nun wird ausgenutzt, dass ein d-dimensionaler Hypercube als 2 *(d-1)*-dimensionale Hypercubes angesehen werden kann, wobei auch diese kleineren

Hypercubes wieder aus Subcubes bestehen. Betrachtet man nun zu Beginn den Cube als zwei Subcubes, wobei das MSB in der Knotenadresse im oberen Subcube 1 ist und im unteren 0. Zwei Prozessoren sind nun Partner, wenn sie sich nur im MSB unterscheiden. Die ersten Schritte sind nun

1. Ein Prozessor (z.B. $P_0$) wählt ein Pivot aus und übermittelt dieses mittels Broadcast an alle anderen Knoten.
2. Die Prozessoren im untern Subcube senden nun alle ihre Zahlen > Pivot an die Partner-Knoten im oberen Subcube. Die Knoten im oberen Subcube senden alle Zahlen ≤ Pivot an die Partner-Knoten im unteren Subcube.
3. Jeder Knoten fügt nun die empfangen Listen an die eigene Liste an.

Danach sind alle Zahlen im unteren Subcube kleiner/gleich dem Pivot und im oberen Subcube größer dem Pivot. Schritte 2 und 3 werden rekursiv in den jeweiligen Subcubes wiederholt, wobei wieder ein Prozessor je Subcube dass Pivot wählt und an alle Knoten in seinen beiden Subsubcubes (jeder Subcube besteht ja wieder aus 2 Subsubcubes) sendet usw. Der Ablauf terminiert nach log *d* rekursiven Phasen. Nun werden in jedem Knoten die Werte sequentiell sortiert und die sortierten Teillisten können zu einer einzigen sortierten Liste eingesammelt werden, in dem alle Knoten in numerischer Reihenfolge beginnend mit der Knotenadresse 0 *(bei aufsteigender Sortierung)* besucht werden. *(Wenn die Knoten in Gray-Code durchnummeriert sind, so sind Knoten in numerischer Ordnung auch Nachbarn, bei binärer Codierung nicht immer!)*

#### *Pivotwahl:*

Eine ungünstige Wahl des Pivot kann zur Folge haben, dass große Teile des Cubes im Idle sind. Für die in 10.2.4 bereits erwähnte *Median*-Strategie kann z.B. ein simpler Bubble Sort verwendet werden, um den Median innerhalb der genommenen Samples zu ermitteln.

#### *Hyperquicksort: ??? wo ist hier der Vorteil ???*

Bei dieser Variante werden in jedem Stadium die Zahlen sortiert so dass alle Knoten mit sortierten Werten arbeiten. Dies vereinfacht die Wahl des Pivot und eliminiert die finale Sortierung.

## 10.4 Andere Sortieralgorithmen

### 10.4.1 Rank Sort

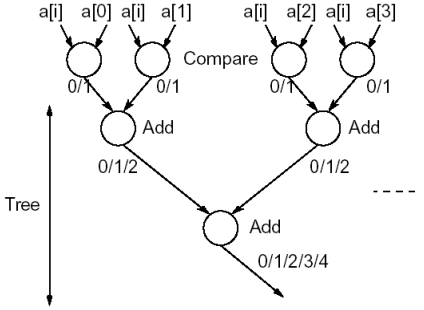
Dabei wird die Anzahl der Werte ermittelt, die kleiner als der aktuell selektierte Wert sind. Diese Anzahl ist die Position (Rang od. rank) des aktuell selektierten Wertes in der sortierten Liste. Dafür ist also jede Zahl mit *n-1* Zahlen zu vergleichen, für alle *n* Zahlen werden also *n(n-1)* Schritte benötigt also eine sequentielle Zeitkomplexität von $O(n^2)$.

#### *Parallelisierung mit n Prozessoren:*

Bei n Prozessoren ist jeder Zahl ein Prozessor zugeordnet und dass Auffinden des Index in der sortierten Liste benötigt $O(n)$ Schritte für jede Zahl und wegen der Parallelität ebenso für die gesamte Sortierung.

#### *Parallelisierung mit $n^2$ Prozessoren:*

Um für *n* Zahlen den Index in einem Schritt zu ermitteln werden *n(n-1)* Prozessoren benötigt. Ein Zähler wird benötigt, um für jede Zahl mitzuzählen, wie viele Zahlen kleiner sind. Das inkrementieren des Zähler erfolgt sequentiell und benötigt max. *n* Schritte. Somit ergibt sich für die gesamte Sortierung *1+n* Schritte also $O(n)$.



Wird für die Inkrementierung des Zählers eine Baumstruktur verwendet so reduziert sich der Aufwand auf $O(\log n)$ mit $n^2$ Prozessoren was aber praktisch keine Bedeutung hat da $n^2$ Prozessoren kaum realisierbar. Da die Prozessoren einen gemeinsamen Zugriff auf die Listenelement benötigen ist ein Shared-Memory System naheliegend. In einem Message-Passing System wäre dafür ein Master-Slave Ansatz passend, wo die Slaves Listenelemente vom Master anfordern können.

Zur Reduzierung der benötigten Prozessoren bietet sich ein Gruppierung der Daten an, so dass bei *m* Werten je Gruppe nur mehr *n/m* Prozessoren benötigt werden. Damit erhöhen sich aber die notwendigen Bearbeitungsschritte in jedem Prozessor um den Faktor *m*.

### 10.4.5 Implementierung von Sortieralgorithmen auf einem Cluster

Effiziente Implementierungen auf einem Cluster verlangen kollektive Operationen wie broadcast, gather, scatter und reduce. Der verteilte Speicher eines Clusters favorisiert natürlich nicht Algorithmen, die einen Zugriff auf weit verteilte Werte verlangen, sondern eher Algorithmen, die lokal arbeiten. Ebenso ist auch der lokal verfügbare Cache zu berücksichtigen wie auch lokale Shared-Memory Systeme bei SMP Clustern wo der Shared-Memory nur von den Prozessoren innerhalb eines SMP Systems benutzt werden kann nicht aber global über den gesamten Cluster.

# 11. Numerische Algorithmen

## 11.2 Implementieren der Matrix Multiplikation

### 11.2.1 Algorithmus

Bei der Multiplikation der *n x l* Matrix ***A*** mit der *l x m* Matrix ***B*** wird jedes Element der *i*-ten Zeile von ***A*** mit dem korrespondierenden Element der *j*-ten Spalte von ***B*** multipliziert und die Produkte aufsummiert sodass für die Ergebnis-Matrix ***C*** gilt

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$

Der Einfachheit halber wird nachfolgend von einer quadratischen *n x n* Matrix ausgegangen.

#### *Sequentieller Code:*

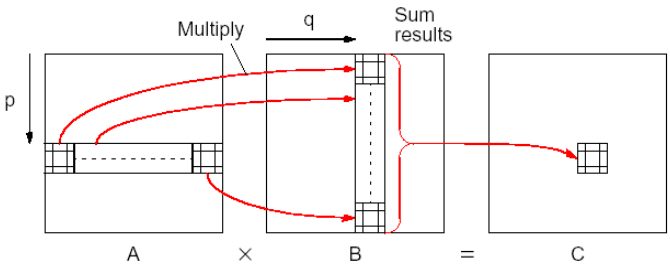
```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        c[i][j] = 0;
        for (k = 0; k < n; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
     }
```

Es werden $n^3$ Multiplikationen und Additionen benötigt und somit folgt $O(n^3)$.

#### *Paralleler Code:*

Es ist offensichtlich dass jede Instanz der innersten Schleife parallel ausgeführt werden kann *(jedes Element von C kann von separatem Prozessor berechnet werden)* und somit ergibt sich mit $p=n$ Prozessoren eine Zeitkomplexität von $O(n^2)$. Bei $p=n^2$ Prozessoren *(jeweils ein Element von A und B werden einem Prozessor zugewiesen)* erreicht man $O(n)$. Mit $p=n^3$ und der Parallelisierung der innersten Schleife erreicht man sogar $O(\log n)$, was aber wegen $O(n^3) \neq n^3 \times O(\log n)$ nicht mehr kostenoptimal ist. Allerdings wurde hier der Kommunikationsaufwand noch nicht berücksichtigt.

**Zerlegung in Teilmatrizen**: Da $p=n$ Prozessoren i.A. nicht praktikabel ist wird die Matrix in Teilmatrizen zerlegt, so dass jeder Prozessor auf einer Gruppe von Daten (Teilmatrix) arbeitet. Jede Teilmatrix kann wie ein einzelnes Element der ursprünglichen Matrix behandelt werden und somit kann die Berechnung in der innersten Schleife analog anstatt mit Matrix-Elementen mit Teilmatrizen formuliert werden -> *Block-Matrix Multiplikation*.




## 11.2.2 Direkte Implementierung

Der einfachste Ansatz ist, für die Berechnung eines jeden Elementes der Ergebnismatrix ***C*** einen Prozessor zu verwenden, also $n^2$ Prozessoren bei einer $n \times n$ Matrix.

### *Analyse:*

*Kommunikation*: Ausgehend von einem Master-Slave Ansatz benötigt jeder Prozessor eine Zeile von ***A*** und eine Spalte von ***B*** für die Berechnungen. Zusätzlich liefert er als Ergebnis ein Element von ***C*** an den Master, woraus folgt dass

$$t_{comm} = n^2(t_{startup} + 2nt_{data}) + n^2(t_{startup} + t_{data})$$

Der dominierende Teil der Kommunikationszeit für das Versenden der Spalten und Zeilen an die Slaves lässt sich durch Verwendung eines *broadcast*-Versandes auf einem *single bus* reduzieren auf

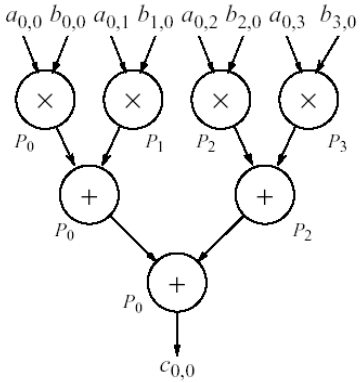
$$t_{comm} = (t_{startup} + n^2 t_{data}) + n^2(t_{startup} + t_{data}) = O(n^2)$$

Damit wird die Zeit für die Rückgabe der Resultate dominierend, da $t_{startup}$ i.A. wesentlich größer ist als $t_{data}$ . Mit einer effizienten Implementierung (z.B. Baumstruktur) einer *gather* Routine kann der Aufwand für die Resultatrückgabe nochmals signifikant reduziert werden.

*Berechnung*: Berechnungen finden nur in den Slaves statt, wobei jeweils $n$ Multiplikationen und Additionen durchgeführt werden, also

$$t_{comp} = 2n\ .$$

Dies kann durch Verwendung einer Baumstruktur noch verbessert werden, so dass letztlich log $n$ Schritte mit $n$ Prozessoren notwendig sind. Damit folgt eine Zeitkomplexität für die Berechnung von $O(\log n)$ bei $n^3$ Prozessoren anstatt $O(n)$ bei $n^2$ Prozessoren.




### *Teilmatrizen:*

Bei Verwendung von $m \times m$ Teilmatrizen wird die ursprüngliche Matrix in $s$ Teilmatrixzeilen und –spalten mit $s = n/m$ aufgeteilt. Also erhält man $s^2$ Teilmatrizen und damit $s^2$ Prozessoren.

*Kommunikation*: Jeder Slave muss nun eine Zeile und eine Spalte von Teilmatrizen empfangen mit je $m^2$ Elementen. Als Resultat wird eine Teilmatrix mit $m^2$ Elementen an den Master gesendet, also folgt für

$$t_{comm} = s^2\{2(t_{startup} + nmt_{data}) + (t_{startup} + m^2 t_{data})\}$$

*(2 mal startup-Zeit im ersten Term ist wahrscheinlich falsch!)*

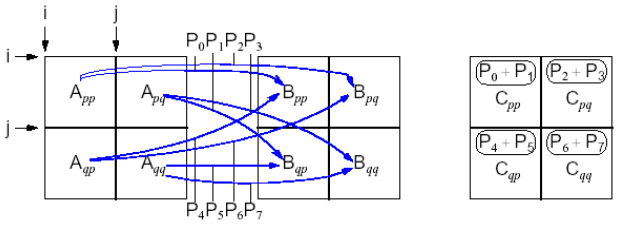
Ein Broadcast würde hier ebenfalls $t_{startup}$ reduzieren. Bei Vergrößerung der Teilmatrizen reduziert sich die Anzahl der benötigten Prozessoren, damit werden größere Datenblöcke je Nachricht übertragen und die Anzahl der Nachrichten reduziert sich. Wird *m* aber zu groß gewählt verliert man Speedup wegen der geringeren Parallelisierung. Es gibt also eine optimale Größe *m* der Teilmatrizen.

*Berechnung*: Jeder Slave führt *s* Teilmatrixmultiplikation und –additionen aus, mit je $m^3$ Multiplikationen und Additionen für eine Teilmatrixmultiplikation und $m^2$ Additionen für eine Teilmatrixaddition. Also folgt für

$$t_{comp} = s(2m^3 + m^2) = O(sm^3) = O(nm^2)$$

### 11.2.3 Rekursive Implementierung

Die beiden *n x n* Matrizen ***A*** und ***B*** (mit *n* als Potenz von 2) werden jeweils in 4 quadratische Teilmatrizen zerlegt. Die entstandenen Teilmatrizen können dann wieder jeweils in weitere Teilmatrizen zerlegt werden, solange, bis man eine Granularität erreicht hat, so dass alle verfügbaren Prozessoren optimal genutzt werden. Letztlich werden dann 8 Teilmatrizenpaare miteinander multipliziert und 4 Resultatpaare zu 4 Teilmatrizen der Ergebnismatrix ***C*** addiert. Es ist also ein klassischer *divide-and-conquer* Ansatz. Die 8 Matrizenmultiplikation je Iteration werden als rekursive Aufrufe implementiert, wobei jeder der 8 Aufrufe parallel auf einem separaten Prozessor ausgeführt werden kann. Es wird also eine Prozessoranzahl benötigt, die eine Potenz von 8 ist.




Mit jedem rekursiven Aufruf wird die Anzahl der zu übergebenden Daten reduziert und lokal gespeichert, was eine ideale Voraussetzung für die Erreichung einer optimalen Performance auf einem Multiprozessorsystem mit Cache Speicher ist, und die Methode ist hervorragend für eine Shared-Memory Architektur geeignet, da der Nachrichtenversand nicht Teil des Algorithmus ist *(-> optimale Realisierung im Speicher mit Pointerstrukturen wie im Begleittext beschrieben).*

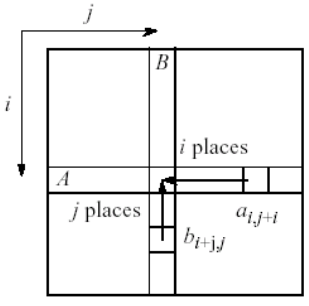
### 11.2.4 Mesh Implementierung

Eine Implementierung der Matrixmultiplikation auf einem Message Passing System wie dem Mesh ist natürlich naheliegend aufgrund der Matrix-Struktur des Mesh. Jeder Knoten des Mesh stellt dabei ein Element od. eine Teilmatrix der Ergebnismatrix dar. Die Mesh-Architektur erlaubt eine simultane Kommunikation benachbarter Knoten im Mesh. Wird ein Mesh auf einer nicht maschenförmigen Netzwerkarchitektur modelliert, so muss auf jeden Fall die Netzwerkcharakteristik im Design des Algorithmus z.B. bei Nachrichtengröße o.ä. berücksichtigt werden.

#### *Cannon's Algorithmus:*

Es wird ein Torus verwendet um bei den Schiebeoperationen den automatischen „wrap around" zu nutzen. Der Algorithmus funktioniert wie folgt *(die Matrixelemente sind i.A. Teilmatrizen)*:

1. Am Beginn hat der Prozessor $P_{i,j}$ die Element $a_{i,j}$ und $b_{i,j}$ $(0 \le i < n, 0 \le j < n)$.
2. Dann werden die Elemente „*aligned*" positioniert, d.h., dass die *i*-te Zeile von ***A*** um *i* Positionen nach links verschoben wird und der *j*-te Spalte von ***B*** um *j* Positionen



nach oben. Damit sind die Element $a_{i,j+i}$ und $b_{i+j,j}$ an der Position des Prozessors $P_{i,j}$. Das Aligning ist notwendig, damit in den nachfolgenden Iteration (in 4.) auch wirklich die Elemente aus A mit ihren korrespondierenden Elementen aus B „*zusammentreffen*“ -> ähnlich wie bei systolischem Array.

3. Jeder Prozessor $P_{i,j}$ multipliziert nun seine Elemente.
4. Danach wird die *i*-te Zeile von ***A*** um eine Position nach links geschoben und die *j*-te Spalte von ***B*** um eine Position nach oben.
5. Jeder Prozessor $P_{i,j}$ multipliziert die neuen Elemente und addiert das Ergebnis zu seiner Teilsumme hinzu.
6. Schritt 4 und 5 werden solange wiederholt, bis alle *n* Elemente einer Zeile/Spalte miteinander multipliziert und die Produkte aufsummiert wurden. Es sind also *n-1* Schiebeoperationen für *n* Zeilen/Spalten notwendig.

Nach Beendigung des Algorithmus repräsentiert das Mesh die Ergebnismatrix ***C***.

### *Analyse:*

*Kommunikation*: Bei $s^2$ Teilmatrizen der Größe *m x m* werden für das initiale „*alignment*“ max. *s-1* Schiebeoperationen jeweils für ***A*** und ***B*** benötigt. Danach werden wieder *s-1* Schieboperationen für ***A*** und ***B*** während der Algorithmusabarbeitung benötigt. Jede Schiebeoperation umfasst $m^2$ Elemente, also folgt für

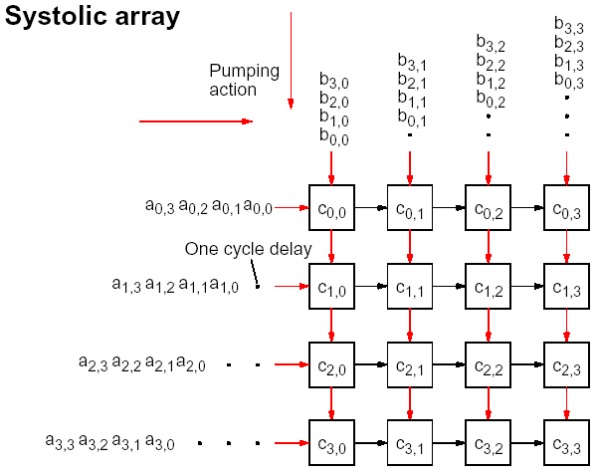
$$t_{comm} = 4(s-1)(t_{startup} + m^2 t_{data}) = O(sm^2) = O(nm) \text{ mit } s = n/m.$$

*Berechnung*: Jede Multiplikation einer Teilmatrix erfordert je $m^3$ Multiplikationen und Additionen, damit ergibt sich mit *s* Wiederholungen

$$t_{comp} = 2sm^3 = 2m^2 n = O(m^2 n).$$

### *2D Pipeline – Systolisches Array:*

Bei einem systolischen Array wird Information in regelmäßigen Intervallen durch das Array „*gepumpt*“ (systolisch kommt aus dem medizinischen Bereich). In dem hier betrachteten 2D systolischen Array wird Information einmal von links nach rechts und von oben nach unten gepumpt, wobei dann die Information die internen Knoten passiert, in denen die Berechnungen stattfinden.




Bei der Multiplikation der beiden Matrizen ***A*** und ***B*** werden nun die Element von ***A*** links beginnend nach rechts durch das Array geschoben, jene von ***B*** von oben nach unten. Nachdem alle Elemente durchgepumpt wurden, stehen in den einzelnen Knoten die Resultate der einzelnen Elemente von ***C***. Es gibt zwar keinen gemeinsamen Takt, trotzdem erfolgt der Ablauf aber synchronisiert zeitlich gesteuert über die `send()` und `recv()` Aufrufe. Wichtig ist dabei, wie in der Abbildung angedeutet, dass jene Knoten, die unter bzw. rechts vom Startknoten $c_{0,0}$ liegen, ihre Elemente entsprechend verzögert erhalten, so dass die Berechnung erst mit dem ersten zusammengehörigen Elementepaar beginnt.

### *Analyse:*

Ähnlich wie bei Cannon’s Algorithmus.

***Matrix-Vektor Multiplikation:***

Für Matrix-Vektor Multiplikation können die selben Methoden wie für die „normale“ Matrix Multiplikation verwendet werden mit dem Unterschied, dass die Matrix ***B*** eben nur aus einer Spalte besteht. *-> Lösung z.B. mit 1D systolischem Array*

### 11.2.5 Andere Methoden zur Matrix Multiplikation

Die Zeitkomplexität von $O(n^3)$ für die sequentielle Matrix Multiplikation lässt sich durch die Verwendung der von Strassen entwickelten Methode leicht verbessern. Man erreicht dadurch $O(n^{2,81})$ bzw. mittlerweile mit einer leicht verbesserten Variante $O(n^{2,376})$. Allerdings ist die Methode um einiges komplexer als die bisher vorgestellten und eine Verwendung erst bei wirklich großen Matrizen sinnvoll.

## 11.3 Lösen eines linearen Gleichungssystems

### 11.3.1 Lineare Gleichungen

Ein lineares Gleichungssystem kann als Matrix-Vektor Multiplikation der Form $\boldsymbol{Ax = b}$ aufgefasst werden mit $\boldsymbol{x}$ und $\boldsymbol{b}$ als Spaltenvektor, wobei $\boldsymbol{A}$ und $\boldsymbol{b}$ gegeben sind und $\boldsymbol{x}$ gesucht ist. In einigen Anwendungen können einige Elemente von $\boldsymbol{A}$ gleich 0 sein. Wenn dies mehrere betrifft, können möglicherweise Methoden verwendet werden, die weniger aufwändige Berechnungen erlauben od. den Speicher effizienter nutzen. Deshalb wird unterschieden zwischen „*dense*“ *(dicht besetzte)* Matrizen mit kaum Elementen gleich 0 und den „*sparse*“ *(dünn besetzte)* Matrizen mit sehr vielen Elementen gleich 0. In der weiteren Betrachtung dieses Abschnitts werden *dense* Matrizen vorausgesetzt. Methoden für *sparse* Matrizen folgen in 11.4.

### 11.3.2 Gauss’sche Eliminierung

Das Ziel der Gauss’schen Eliminierung ist es, die erweiterte Koeffizientenmatrix eines linearen Gleichungssystems in die erweiterte Koeffizientenmatrix eines gestaffelten Systems (Dreieckform) umzuwandeln um dann mittels Rückwärtssubstitution das gestaffelte System zu lösen. Die Vorgangsweise:

1. Es wird die am weitesten links stehende Spalte bestimmt, die von Null verschiedene Werte enthält.
2. Ist die oberste Zahl der in Schritt 1 gefundenen Spalte eine 0, dann wird die erste Zeile mit einer geeigneten anderen Zeile vertauscht.
3. Ist $\alpha$ das erste Element der in Schritt 1 gefundenen Spalte, dann wird die erste Zeile durch $\alpha$ dividiert, um ein führende Eins zu erzeugen.
4. Es wird ein passendes Vielfaches der ersten Zeile zu den übrigen Zeilen addiert, um unterhalb der führenden Eins Nullen zu erzeugen.
5. Die ersten 4 Schritte werden auf den Teil der Matrix angewendet, den man erhält, wenn man die erste Zeile streicht. Das Verfahren wird solange wiederholt, bis man die erweiterte Koeffizientenmatrix eines gestaffelten Systems erhalten hat.
6. Das gestaffelte System wird durch Rückwärtssubstitution gelöst.

Diese Prozedur zeigt allerdings keine besonders gute Stabilität auf digitalen Systemen. Wenn nämlich das zu eliminierende Element $a_{i,i}$ gleich 0 oder sehr klein ist, so kann auf digitalen Systemen dass für die Eliminierung geeignete Vielfache von $a_{j,i}$ – nämlich $-a_{j,i}/a_{i,i}$ nicht mehr berechnet werden. Es muss dann ein sog. *partielles Pivot* verwendet werden, in dem die betroffenen *i*-te Zeile mit jener darunter liegenden Zeile ausgetauscht wird, in der in der *i*-ten Spalte das absolut größte Element ist. Diese numerische Stabilität muss in jedem Schritt der Prozedur geprüft werden und kann nicht vor Beginn der Gauss’schen Eliminierung erfolgen.

***Sequentieller Code:***

```
for (i=0; i<n-1; i++)/* for each row, except last */
     for (j=i+1; j<n; j++) {    /* step through subsequent rows */
          m = a[j][i]/a[i][i]; /* compute multiplier */
          for (k=i; k<n; k++)  /* modify last n-i-1 elements of
                                  row j */
               a[j][k] = a[j][k] - a[i][k]*m;
          b[j] = b[j] - b[i]*m;/* modify right side */
     }
```

Die Zeitkomplexität beträgt $O(n^3)$.

## 11.3.3 Parallele Implementierung

Offensichtlich ist die innere Schleife des sequentiellen Codes, in der die Zeile *j* modifiziert wird, unabhängig vom restlichen Teil und bietet sich daher für eine Parallelisierung an. Eine mögliche Parallelisierung wäre die Aufteilung der n Gleichungen auf n Prozessoren. Jeder Prozessor $P_{i+1}$ bis $P_{n-1}$ muss zuerst die Elemente der *i*-ten Zeile erhalten bevor er mit seinen Berechnungen beginnt. Der Ablauf beginnt mit $i = 0$:

1. Prozessor $P_i$ sendet seine Zeilenelemente (inkl. $b_i$) (mittels *Broadcast*) an die anderen *n-i-1* Prozessoren.
2. Die Prozessoren $P_{i+1}$ bis $P_{n-1}$ berechnen parallel ihre Multiplikatoren und modifizieren ihre *n-i+1* Elemente (Element des Spaltenvektors *b* ist auch dabei).
3. *i* wird inkrementiert und Schritte 1 bis 3 wiederholt bis $i = n-1$.

***Analyse:***

*Kommunikation*: Unter der Annahme, dass der *broadcast* in einem Schritt durchgeführt werden kann, sind n-1 *broadcast* Nachrichten notwendig, die aber sequentiell ausgeführt werden müssen, da nach jedem *broadcast* die Zeilen modifiziert werden. Die *i*-te *broadcast* Nachricht enthält n-i+1 Elemente also für die gesamte Kommunikation

$$t_{comm} = \sum_{i=0}^{n-2} (t_{startup} + (n-i+1)t_{data}) = \left( (n-1)t_{startup} + \left( \frac{(n+2)(n+1)}{2} - 3 \right) t_{data} \right)$$

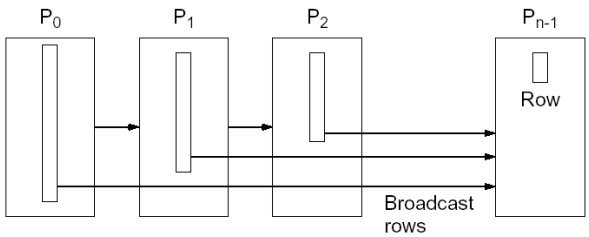
und damit eine Zeitkomplexität von $O(n^2)$. Für große *n* kann $t_{data}$ in der Anfangsphase die Kommunikationszeit dominieren. In der Praxis benötigt ein broadcast i.A. natürlich mehr als einen Schritt, abhängig von der zugrunde liegenden Systemarchitektur.

*Berechnung*: Jeder Prozessor $P_j$ unterhalb des broadcast Prozessors $P_i$ berechnet seinen Multiplikator und modifiziert seine *n-j+2* (wegen $j = i+1$)Elemente. Wird die Berechnung des Multiplikators vernachlässigt so ergibt sich für die gesamte Berechnung

$$t_{comp} = \sum_{j=1}^{n-1} (n-j+2) = \frac{(n+3)(n+2)}{2} - 3 = O(n^2).$$

Allerdings ist die Effizienz sehr gering, da alle Prozessoren oberhalb der *i*-ten Zeile nicht mehr in den Berechnungen eingebunden sind.

***Pipeline Konfiguration:***



***Partitioning:***

Um die Anzahl der benötigten Prozessoren zu verringern, kann die Matrix in Zeilengruppen aufgeteilt werden. Jeder der *p* Prozessoren erhält *n/p* Zeilen zur Berechnung -> *strip partitioning*. Allerdings ist damit das Problem der ungleichen Lastverteilung nicht gelöst. Dafür gibt es einen modifizierten Ansatz, das sog. *cyclic-striped partitioning*. Dabei werden einem Prozessor Zeilen aus verschiedenen Matrixbereichen zugeteilt, also z.B. $P_0$ erhält die Zeilen *0, n/p, 2n/p, ...* und $P_1$ die Zeilen *1, n/p+1, 2n/p+1 ...* usw.

## 11.4 Iterative Methoden

Dabei geht es um Methoden zur Lösung linearer Gleichungssysteme und partieller Differentialgleichungen mittels iterativer Verfahren.

-> siehe Kurstext S.357ff