Parallele Programmierung mit C++
SS 2019

Andreas F. Borchert
Universität Ulm

26. Juli 2019

Inhalte:

- Einführung in die Prozessalgebra CSP
- Architekturen paralleler Systeme
- Parallelisierungstechniken:
  - Threads
  - OpenMP
  - MPI
  - Koprozessoren mit CUDA, OpenCL und OpenACC
- Design-Pattern für die Organisation der Parallelisierung, der Algorithmen, der sie unterstützenden Verwaltung, der Kommunikation und Synchronisierung

Warum parallelisieren?

Ohne Parallelisierung ist keine signifikante Leistungssteigerung mehr zu erwarten:

- Moore's law: Etwa alle zwei Jahre verdoppelt sich die Zahl der integrierten Transistoren, die sich auf einem integrierten Schaltkreis zu vertretbaren Kosten unterbringen lassen.
- Das bedeutet jedoch nicht, dass im gleichen Maße auch die Rechenleistung bzw. die Taktraten eines Prozessors steigen.
- Ein Teil der zusätzlichen Transistoren sind einem verbesserten Caching gewidmet und insbesondere der Unterstützung mehrerer Kerne (*multicore processors*).
- Dual-Core- und Quad-Core-Maschinen sind bereits weit verbreitet. Es gibt aber auch Prozessoren mit 56 Kernen und 112 Threads (Intel Xeon Platinum 9282), Koprozessoren mit 72 Kernen (Intel Xeon Phi) und GPUs mit 5.120 Kernen (Nvidia Quadro GV100).

Warum parallelisieren? 4

Ohne Parallelisierung wird keine Skalierbarkeit erreicht:

- Google verteilt seine Webdienste über 15 Zentren. Eines dieser Zentren hatte einem älteren Bericht des *The Register* zufolge 45 Container mit jeweils 1.160 Rechnern.
- Der Supercomputer Sunway TaihuLight ist ausgestattet mit 160 sogenannten Supernodes, die jeweils aus 256 Sunway-RISC-Prozessoren umfassen, die jeweils aus vier Verwaltungsprozessoren mit je 64 Kernen bestehen. Insgesamt sind das 10.649.600 Kerne. Konkret verwendet werden hier C, C++ und Fortran in Kombination mit MPI, OpenMP und OpenACC. (Siehe Haohuan Fu et al, *The Sunway TaihuLight supercomputer: system and applications*.)
- Das bwUniCluster besteht im wesentlichen aus 512 Knoten mit jeweils zwei Intel-Xeon-Prozessoren mit je acht Kernen. Die Knoten sind miteinander über InfiniBand vernetzt (hier kommt wiederum MPI zum Einsatz).

- Traditionell ging häufig eine Parallelisierung von einer idealisierten Welt aus, bei der n Prozesse getrennt arbeiteten und miteinander ohne nennenswerten Aufwand sich synchronisieren und miteinander kommunizieren konnten.
- Die Realität sieht jedoch inzwischen sehr viel komplexer aus:
  - Algorithmen können nicht mehr ohne tiefe Kenntnisse der zugrundeliegenden Architektur sinnvoll parallelisiert werden.
  - Der Zeitaufwand für Synchronisierung, Kommunikation und Speicherzugriffe ist von wesentlicher Bedeutung.
  - Parallelisierungen finden typischerweise auf drei Ebenen statt: Cluster, Prozessor-Kerne und Koprozessoren.
  - Bei einigen Koprozessoren wie insbesondere GPUs operieren die Kerne nicht unabhängig voneinander.

Warum ist die Architektur relevant?

- Es gehört zu den Errungenschaften in der Informatik, dass Software-Anwendungen weitgehend plattform-unabhängig entwickelt werden können.
- Dies wird erreicht durch geeignete Programmiersprachen, Bibliotheken und Standards, die genügend weit von der konkreten Maschine und dem Betriebssystem abstrahieren.
- Leider lässt sich dieser Erfolg nicht ohne weiteres in den Bereich des *High Performance Computing* übertragen.
- Entsprechend muss die Gestaltung eines parallelen Algorithmus und die Wahl und Konfiguration einer geeigneten zugrundeliegenden Architektur Hand in Hand gehen.
- Ziel ist nicht mehr eine höchstmögliche Portabilität, sondern ein möglichst hoher Grad an Effizienz bei der Ausführung auf einer ausgewählten Plattform.

Fokus der Vorlesung

- Der Schwerpunkt liegt bei den Techniken zur Parallelisierung von Anwendungen, bei denen Kommunikation und Synchronisierung eine wichtige Rolle spielen.
- Ziel ist es, die Grundlagen zu erlernen, die es erlauben, geeignete Architekturen für parallelisierbare Problemstellungen auszuwählen und dazu passende Algorithmen zu entwickeln.
- CSP dient im Rahmen der Vorlesung als Instrument zur formalen Beschreibung von Prozessen, die sich synchronisieren und miteinander kommunizieren.

- Für OpenMP, MPI, CUDA, OpenCL und OpenACC ist die Auswahl nicht sehr groß. Neben C++ kommen hier nur noch Fortran oder C in Frage und hier bietet C++ im Vergleich dann doch die moderneren Sprachkonzepte.
- Bei rechenintensiven Anwendungen spielen Cache-Optimierungen eine nicht zu unterschätzende Rolle. Diese können durchgeführt werden, wenn die Programmiersprache es erlaubt, das Layout von Datenstrukturen im Speicher genau festzulegen. Bei C++ ist dies problemlos möglich, bei der Mehrzahl der Alternativen ist das nicht vorgesehen.
- Ebenfalls von Bedeutung ist das Vorhandensein von Übersetzern mit fortgeschrittenen Optimierungstechniken (wie etwa *loop unrolling* und *instruction scheduling*), um das Pipelining moderner Architekturen auszureizen. Der Aufwand ist dafür polynomial und damit nur einmalig beim Übersetzen zumutbar – jedoch nicht zur Laufzeit (wie etwa mit einem JIT-Übersetzer bei Java).

- Grundkenntnisse in Informatik. Insbesondere sollte keine Scheu davor bestehen, etwas zu programmieren.
- Für diejenigen, die bislang weder C noch C++ kennen, gibt es in den Übungen zu Beginn eine kleine Einführung dazu.
- Freude daran, etwas auch an einem Rechner auszuprobieren und genügend Ausdauer, dass nicht beim ersten Fehlversuch aufgegeben wird.

Syllabus: Struktur

- Jede Woche gibt es zwei Vorlesungsstunden an jedem Donnerstag von 16-18 Uhr in der Helmholtzstraße 18, Raum E.60.
- Die Übungen sind am Freitag von 14-16 Uhr in der Helmholtzstraße 18, Raum E.44.
- Da die Vorlesungstermine am 30. Mai und am 20. Juni wegen Feiertagen ausfallen, werden diese in anderen Wochen nachgeholt, indem der Übungstermin am Freitag auch für die Vorlesung genutzt wird.
- Freitags-Vorlesungen finden in der Helmholtzstraße 18 im Raum 1.20 statt.
- An den „Brückentagen“ am 31. Mai und 21. Juni finden keine Übungen statt.
- Webseite: `https://www.uni-ulm.de/mawi/mawi-numerik/lehrenumerik/sommersemester-2019/vorlesung-parallele-programmierung-mit-c/`

- Wir haben keinen Übungsleiter, keine Tutoren und auch keine Korrekteure.
- Lösungen zu Übungsaufgaben werden auf unseren Rechnern mit einem speziellen Werkzeug eingereicht. Details zu dem Verfahren werden zusammen mit der ersten Übungsaufgabe vorgestellt.
- Zu Beginn dienen die Übungen auch dazu, C++ einzuführen.

- Die Prüfungen erfolgen mündlich zu Terminen, die jeweils mit mir zu vereinbaren sind.
- Für die mündliche Prüfungen wird keine Vorleistung benötigt.
- Terminlich bin ich flexibel. Abgesehen von meinen Urlaubszeiten bin ich jederzeit bereit zu prüfen. Nennen Sie eine Woche, ich gebe Ihnen dann gerne einen Termin innerhalb dieser Woche.

- Die Vorlesungsfolien und einige zusätzliche Materialien werden auf der Webseite der Vorlesung zur Verfügung gestellt werden.
- Verschiedene Dokumente wie beispielsweise Tony Hoares Buch *Communicating Sequential Processes* oder Spezifikationen zu Threads, OpenMP und MPI stehen frei zum Herunterladen zur Verfügung.

- C. A. R. Hoare, *Communicating Sequential Processes*, ISBN 0131532898
- Timothy G. Mattson et al, *Patterns for Parallel Programming*, ISBN 0321228111
- Anthony Williams, *C++ Concurrency in Action*, ISBN 1-933988-77-0
- Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*, ISBN 0071232656
- Nicholas Wilt, *The CUDA Handbook*, ISBN 0-321-80946-7
- Matthew Scarpino, *OpenCL in Action*, ISBN 1-61729-017-3

Syllabus: Sprechstunden

- Sie sind eingeladen, mich jederzeit per E-Mail zu kontaktieren:
  E-Mail: `andreas.borchert@uni-ulm.de`
- Meine reguläre Sprechstunde ist am Mittwoch 10:00-11:30 Uhr. Zu finden bin ich in der Helmholtzstraße 20, Zimmer 1.23.
- Zu anderen Zeiten können Sie auch gerne vorbeischauen, aber es ist dann nicht immer garantiert, daß ich Zeit habe. Gegebenenfalls lohnt sich vorher ein Telefonanruf: 23572.

Syllabus: Nicht verzweifeln! 16

- Bevor Sie bei der Lösung einer Übungsaufgabe völlig verzweifeln, sollten Sie mir Ihren aktuellen Stand per E-Mail zukommen lassen. Dann werde ich versuchen, Ihnen zu helfen.
- Das kann auch am Wochenende funktionieren.

Syllabus: Feedback

- Feedback ist ausdrücklich erwünscht.
- Noch ist die Vorlesung nicht fertig. Das bedeutet auch, dass ich auf Ihre Anregungen eingehen kann und auch Punkte mehr berücksichtigen kann, die Ihnen wichtig sind.

C. A. R. Hoare 1994

C. A. R. Hoare entwickelte 1978–1985 die erste Prozessalgebra CSP (*Communicating Sequential Processes*). Obwohl es inzwischen einige alternative Prozessalgebren gibt, ist sie nach wie vor die bedeutendste geblieben.

Eine Prozessalgebra erlaubt die formale Beschreibung paralleler Systeme und die Ableitung ihrer innewohnenden Eigenschaften.

Die folgenden Ausführungen lehnen sich recht eng an das Buch von Hoare an.

Ereignisklassen

Noebse, Wikimedia Commons

- Die Interaktionen zwischen Prozessen erfolgt über Ereignisse.
- Jedes Ereignis gehört einer Ereignisklasse an.
- Die für einen Prozess relevanten Ereignisklassen sind endlich und fest vorgegeben.
- Ereignisklassen bilden das Alphabet eines Prozesses und dienen der Abstrahierung.
- Der abgebildete Zeitungsautomat hat beispielsweise die Ereignisklassen *muenze5*, *muenze10*, *muenze20*, *muenze50*, *muenze100*, *zeitung_ausgeben* und *rueckgabe*.

- Ein konkretes für die Betrachtung relevantes Ereignis gehört einer Ereignisklasse an.
- Das Eintreten eines Ereignisses benötigt keine Zeit. Wenn längere Ereignisse modelliert werden sollen, kann dies erfolgen durch zwei verschiedene Ereignisklassen, die den Anfang und das Ende eines längeren Ereignisses markieren.
- Es gibt keinen Initiator eines Ereignisses und keine Kausalität.
- Prozesse werden dadurch charakterisiert, in welcher Reihenfolge sie an welchen Ereignissen teilnehmen.

- Ereignisklassen werden mit Kleinbuchstaben beschrieben. (Beispiele: *muenze5*, *a*, *b*, *c*.)
- Für die Namen der Prozesse werden Großbuchstaben verwendet. (Beispiele: *ZA*, *P*, *Q*, *R*.)
- Für Mengen von Ereignisklassen werden einzelne Großbuchstaben aus dem Anfang des Alphabets gewählt: *A*, *B*, *C*.
- Für Prozessvariablen werden die Großbuchstaben am Ende des Alphabets gewählt: *X*, *Y*, *Z*.
- Das Alphabet eines Prozesses *P* wird mit αP bezeichnet. Beispiel:

$$\begin{aligned} \alpha ZA \quad = \quad & \{\mathit{muenze5},\ \mathit{muenze10},\ \mathit{muenze20},\ \mathit{muenze50},\\ & \mathit{muenze100},\ \mathit{zeitung\_ausgeben},\ \mathit{rueckgabe}\} \end{aligned}$$

$$(x \to P)$$

- Gegeben sei eine Ereignisklasse x und ein Prozess P. Dann beschreibt $(x \to P)$ einen Prozess, der zuerst an einem Ereignis der Klasse x teilnimmt und sich dann wie P verhält.
- Ein Zeitungsautomat, der eine Ein-Euro-Münze entgegennimmt und dann den Dienst einstellt, kann so beschrieben werden:

  $$(\mathit{muenze100} \to \mathit{STOP}_{\alpha ZM})$$

- $\mathit{STOP}_{\alpha P}$ repräsentiert einen Prozess mit dem Alphabet αP, der an keinen Ereignissen aus diesem Alphabet mehr teilnimmt.
- Ein Zeitungsautomat, der eine Ein-Euro-Münze entgegennimmt, eine Zeitung ausgibt und dann den Dienst einstellt:

  $$(\mathit{muenze100} \to (\mathit{zeitung\_ausgeben} \to \mathit{STOP}_{\alpha ZM}))$$

- Der Operator $\to$ ist rechts-assoziativ und erwartet links eine Ereignisklasse und rechts einen Prozess. Entsprechend dürften in diesen Beispielen auch die Klammern wegfallen.

- Eine Uhr, bei der nur das Ticken relevant ist ($\alpha UHR = \{tick\}$), könnte so beschrieben werden:

$$UHR = (tick \rightarrow UHR)$$

- Der Prozess *UHR* ist dann eine Lösung dieser Gleichung.
- So eine Gleichung erlaubt eine iterative Expansion:

$$\begin{aligned} UHR &= (tick \rightarrow UHR) \\ &= (tick \rightarrow tick \rightarrow UHR) \\ &= (tick \rightarrow tick \rightarrow tick \rightarrow UHR) \\ &= \ldots \end{aligned}$$

- Dies gelingt jedoch nur, wenn die rechte Seite der Gleichung mit einer Präfixnotation beginnt. Hingegen würde $X = X$ nichts über X aussagen.
- Eine Prozessbeschreibung, die mit einer Präfixnotation beginnt, wird *geschützt* genannt.

$$\mu X : A.F(X)$$

- Wenn $F(X)$ ein geschütze Prozessbeschreibung ist, die X enthält, und $\alpha X = A$, dann hat $X = F(X)$ eine eindeutige Lösung für das Alphabet A.
- Dies kann auch kürzer als $\mu X : A.F(X)$ geschrieben werden, wobei in dieser Notation X eine lokal gebundene Variable ist.
- Beispiel für die tickende Uhr:

  $$\mu X : \{tick\} . (tick \rightarrow X)$$

- Beispiel für einen Zeitungsautomaten, der nur Ein-Euro-Münzen akzeptiert:

  $$\begin{aligned} \mu X : \quad & \{muenze100, zeitung\_ausgeben\} . \\ & (muenze100 \rightarrow zeitung\_ausgeben \rightarrow X) \end{aligned}$$

- Die Angabe des Alphabets A kann entfallen, wenn es offensichtlich ist.

$$(x \to P \mid y \to Q)$$

- Der Operator „|“ ermöglicht eine freie Auswahl. Je nachdem ob zuerst ein Ereignis der Klasse x oder y eintritt, geht es mit P bzw. Q weiter.
- Die Alphabete müssen bei diesem Konstrukt zueinander passen: $\alpha P = \alpha Q$. Und die genannten Ereignisklassen müssen in dem gemeinsamen Alphabet enthalten sein: $\{x, y\} \subseteq \alpha P$.
- Ein Zeitungsautomat, der entweder zwei 50-Cent-Stücke oder eine Ein-Euro-Münze akzeptiert, bevor er eine Zeitung ausgibt:

$$\begin{aligned} \mu X : \quad & \{\mathit{muenze50}, \mathit{muenze100}, \mathit{zeitung\_ausgeben}\} . \\ & (\mathit{muenze100} \to \mathit{zeitung\_ausgeben} \to X \\ \mid \quad & \mathit{muenze50} \to \mathit{muenze50} \to \mathit{zeitung\_ausgeben} \to X) \end{aligned}$$

- Es können beliebig viele Alternativen gegeben werden, aber all die angegebenen Präfixe müssen sich voneinander unterscheiden.

- Es sind auch mehrere wechselseitig rekursive Gleichungen möglich. Dann sollte aber die rechte Seite jeweils geschützt sein und jede linke Seite nur ein einziges Mal vorkommen.
- Ein Fahrzeug FZ hat eine Automatik mit den Einstellungen d (vorwärts), r (rückwärts) und p (Parken) und kann entsprechend vorwärts (vf) oder rückwärts (rf) fahren:

$$\begin{aligned} FZ &= (d \to FZV \mid r \to FZR) \\ FZV &= (vf \to FZV \mid p \to FZ \mid r \to FZR) \\ FZR &= (rf \to FZR \mid p \to FZ \mid d \to FZV) \end{aligned}$$

Ablauf eines Prozesses

- Ein Ablauf (*trace*) eines Prozesses ist eine endliche Sequenz von Symbolen aus dem Alphabet, das die Ereignisklassen repräsentiert, an denen ein Prozess bis zu einem Zeitpunkt teilgenommen hat.
- Ein Ablauf wird in winkligen Klammern $\langle \ldots \rangle$ notiert.
- $\langle x, y \rangle$ besteht aus zwei Ereignisklassen: Zuerst x, dann y. $\langle x \rangle$ besteht nur aus dem Ereignis x. $\langle \rangle$ ist die leere Sequenz.
- Beispiel: $\langle d, vf, vf, vf, r, rf, p \rangle$ ist ein möglicher Ablauf von FZ.

$$traces(P)$$

- Sei P ein Prozess, dann liefert $traces(P)$ die Menge aller möglichen Abläufe.
- Beispiel: $traces\,((muenze100 \rightarrow STOP)) = \{\langle\rangle, \langle muenze100\rangle\}$
- Beispiel:
  $traces\,(\mu X : (tick \rightarrow X)) = \{\langle\rangle, \langle tick\rangle, \langle tick, tick\rangle, \langle tick, tick, tick\rangle, \ldots\}$

Operationen für Abläufe

- Seien s und t zwei Abläufe, dann ist $s \hat{} t$ eine zusammengesetzte Ablaufsequenz von s und t. Beispiel: $\langle a, b \rangle \hat{} \langle c, a \rangle = \langle a, b, c, a \rangle$
- Die Relation $s \leq t$ gilt genau dann, wenn $\exists u : s \hat{} u = t$
- n aufeinanderfolgende Kopien einer Sequenz sind so definiert:

$$\begin{aligned} t^0 &= \langle \rangle \\ t^{n+1} &= t \hat{} t^n \end{aligned}$$

- Sei A ein Alphabet und t ein Ablauf, dann ist $t \upharpoonright A$ der Ablauf, bei dem alle Elemente $a \notin A$ aus t entfernt worden sind. Beispiel: $\langle a, b, c, a, c \rangle \upharpoonright \{a, b\} = \langle a, b, a \rangle$

- $STOP_A$ mit $\alpha STOP_A = A$ ist ein Prozess, der an keinem Ereignis teilnimmt.
- Entsprechend gilt: $traces(STOP_A) = \{\langle\rangle\}$.
- $STOP$ repräsentiert den Deadlock.
- RUN_A mit $\alpha RUN_A = A$ ist ein Prozess, der an allen Ereignissen des Alphabets A in beliebiger Reihenfolge teilnimmt.
- Entsprechend gilt: $traces(RUN_A) = A^*$, wobei A^* die Menge aller endlichen Sequenzen aus dem Alphabet A ist.
- Beispiel: $\mu X : \{tick\}.\,(tick \rightarrow X) = RUN_{\{tick\}}$
- $SKIP_A$ mit $\alpha SKIP_A = A \cup \{\checkmark\}$ verhält sich analog zu $STOP_A$ ist aber nicht als Deadlock zu verstehen, sondern als erfolgreiches Ende, das durch das Ereignis $\checkmark$ repräsentiert wird. Das Ende-Ereignis darf ansonsten nicht verwendet werden.

$$P \parallel Q$$

- P und Q sind zwei Prozesse, bei denen wir zunächst aus Gründen der Einfachheit ausgehen, dass $\alpha P = \alpha Q = A$.
- Dann ergibt $P \parallel Q$ einen Prozess, bei dem Ereignisse aus dem gemeinsamen Alphabet A genau dann stattfinden, wenn sie simultan bei den Prozessen P und Q stattfinden.
- Beide Prozesse laufen parallel, aber die Ereignisse finden synchron statt.
- Beispiel:

$$\begin{aligned} ZA &= (muenze50 \rightarrow zeitung\_ausgeben \rightarrow ZA \mid \\ &\quad\ muenze100 \rightarrow zeitung\_ausgeben \rightarrow ZA) \\ K &= (muenze20 \rightarrow zeitung\_ausgeben \rightarrow K \mid \\ &\quad\ muenze50 \rightarrow zeitung\_ausgeben \rightarrow K) \\ ZA \parallel K &= \mu X : (muenze50 \rightarrow zeitung\_ausgeben \rightarrow X) \end{aligned}$$

Unter der Voraussetzung, dass $\alpha P = \alpha Q$ gilt, gelten folgende Gesetze:

- Der Operator „||“ ist kommutativ: $P \parallel Q = Q \parallel P$
- Es gilt das Assoziativgesetz: $(P \parallel Q) \parallel R = P \parallel (Q \parallel R)$
- Ein Deadlock ist viral: $P \parallel STOP = STOP$
- Eine Kombination mit RUN hat keinen Effekt: $P \parallel RUN = P$
- Wenn beide Prozesse kein gemeinsames Folge-Ereignis finden $(a \neq b)$, haben wir einen Deadlock: $(a \rightarrow P) \parallel (b \rightarrow Q) = STOP$
- $traces(P \parallel Q) = traces(P) \cap traces(Q)$

$P \parallel Q$ ist auch zulässig, falls $\alpha P \neq \alpha Q$. Dann gilt:

- $\alpha(P \parallel Q) = \alpha P \cup \alpha Q$
- Ereignisse aus $\alpha P \setminus \alpha Q$ werden nur von P verfolgt und Ereignisse aus $\alpha Q \setminus \alpha P$ nur von Q.
- Nur Ereignisse aus $\alpha P \cap \alpha Q$ betreffen beide Prozesse.
- Es gelten weiterhin die Gesetze der Kommutativität und der Assoziativität.
-

$$\begin{aligned} traces(P \parallel Q) \;=\; \{t \mid\; & (t \upharpoonright \alpha P) \in traces(P) \;\wedge \\ & (t \upharpoonright \alpha Q) \in traces(Q) \;\wedge \\ & t \in (\alpha P \cup \alpha Q)^*\} \end{aligned}$$

Es gelte

$$\alpha ZA = \alpha K = \{muenze20, muenze50, muenze100, zeitung\_ausgeben\}$$

und es sei gegeben:

$$\begin{aligned} ZA &= (muenze50 \rightarrow zeitung\_ausgeben \rightarrow ZA \mid \\ &\quad\ muenze100 \rightarrow zeitung\_ausgeben \rightarrow ZA) \\ K &= (muenze20 \rightarrow zeitung\_ausgeben \rightarrow K \mid \\ &\quad\ muenze50 \rightarrow zeitung\_ausgeben \rightarrow K) \end{aligned}$$

Dann gilt: $traces(ZA \parallel K) = \{t | t \leq \langle muenze50, zeitung\_ausgeben \rangle^n\}$

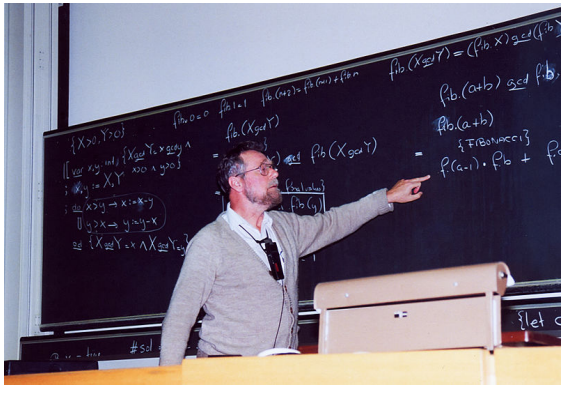
Wenn jedoch *ZA* und *K* unterschiedliche Alphabete haben, dann werden Ereignisse wie *muenze20* oder *muenze100* nur von einem der beiden parallel laufenden Prozesse wahrgenommen:

$$\begin{aligned} \alpha ZA &= \{muenze50, muenze100, zeitung\_ausgeben\} \\ \alpha K &= \{muenze20, muenze50, zeitung\_ausgeben\} \end{aligned}$$

Wenn dann eine Ablaufsequenz mit *muenze20* beginnt, dann muss zwingend *muenze100* folgen, damit beide Prozesse, *ZA* und *K* das Ereignis *zeitung_ausgeben* gemeinsam wahrnehmen können. Entsprechend gilt

$$\begin{aligned} ZA \,||\, K = \; & \mu X : (muenze50 \rightarrow zeitung\_ausgeben \rightarrow X \mid \\ & muenze20 \rightarrow muenze100 \rightarrow zeitung\_ausgeben \rightarrow X \mid \\ & muenze100 \rightarrow muenze20 \rightarrow zeitung\_ausgeben \rightarrow X) \end{aligned}$$

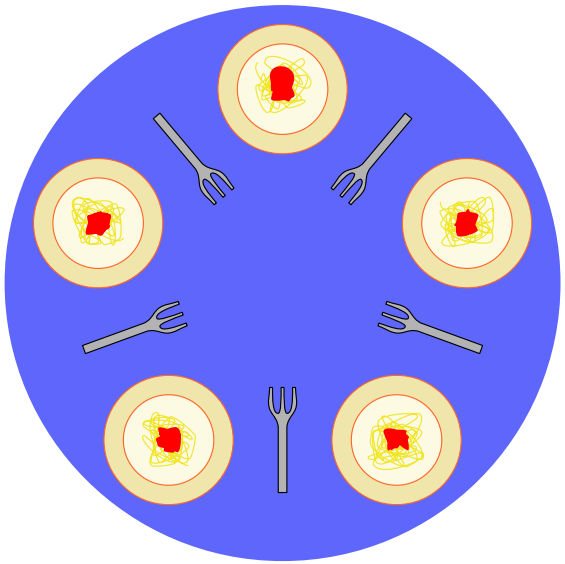
Beispiel: Philosophenproblem



Edsger Dijkstra 1994

Edsger Dijkstra stellte in dem Aufsatz *Hierarchical ordering of sequential processes* in der *Acta Informatica* 1971 Semaphore als Mittel der Synchronisierung von Prozessen vor und demonstrierte diese Technik an dem von ihm aufgestellten Philosophenproblem, das die Gefahren eines Deadlocks und des Aushungerns sehr schön demonstrierte.

Beispiel: Philosophenproblem



Gegeben seien fünf Philosophen $P_0, \ldots, P_4$, die an einem Tisch sitzen und die auf ihrem Teller befindlichen Spaghetti verzehren möchten. Zum Essen werden jeweils zwei Gabeln benötigt, die bereits neben den Tellern liegen. Jedoch stehen insgesamt nur fünf Gabeln $G_0, \ldots, G_4$ zur Verfügung.

Entsprechend kann die zwischen den Philosophen P_i und $P_{i+1 \bmod 5}$ liegende Gabel G_i nur von einem der beiden genommen werden. Und der Philosoph P_i kann erst dann essen, wenn es ihm gelungen ist, die Gabeln $G_{i-1 \bmod 5}$ und G_i zu ergattern.

Beispiel: Philosophenproblem

Ein erster Lösungsansatz:

$$
\begin{aligned}
DP &= P_0 \parallel \ldots \parallel P_4 \parallel G_0 \parallel \ldots \parallel G_4 \\
G_i &= (G_i\_genommen\_von\_P_i \rightarrow \\
&\quad G_i\_hingelegt\_von\_P_i \rightarrow G_i \mid \\
&\quad G_i\_genommen\_von\_P_{i+1 \bmod 5} \rightarrow \\
&\quad G_i\_hingelegt\_von\_P_{i+1 \bmod 5} \rightarrow G_i) \\
P_i &= (G_{i-1 \bmod 5}\_genommen\_von\_P_i \rightarrow \\
&\quad G_i\_genommen\_von\_P_i \rightarrow \\
&\quad essen_i \rightarrow \\
&\quad G_{i-1 \bmod 5}\_hingelegt\_von\_P_i \rightarrow \\
&\quad G_i\_hingelegt\_von\_P_i \rightarrow P_i)
\end{aligned}
$$

αP_i und αG_i seien hier jeweils durch die in den jeweiligen Definitionen explizit genannten Ereignisse gegeben.

Beispiel: Philosophenproblem

Wenn alle Philosophen gleichzeitig loslegen mit dem Aufnehmen der jeweils linken Gabel, haben wir einen Deadlock:

$$t = \langle G_4\_genommen\_von\_P_0, G_0\_genommen\_von\_P_1, G_1\_genommen\_von\_P_2, G_2\_genommen\_von\_P_3, G_3\_genommen\_von\_P_4 \rangle \in traces(DP)$$

$$\nexists\, u \in traces(DP) : u > t$$

Wenn P_0 und P_2 immer schneller als die anderen sind, werden P_1, P_3 und P_4 nie zum Essen kommen:

$$\langle \{ G_4\_genommen\_von\_P_0, G_1\_genommen\_von\_P_2, G_0\_genommen\_von\_P_0, G_2\_genommen\_von\_P_2, essen\_0, essen\_2, G_4\_hingelegt\_von\_P_0, G_1\_hingelegt\_von\_P_2, G_0\_hingelegt\_von\_P_0, G_2\_hingelegt\_von\_P_2 \}^n \rangle \subset traces(DP)$$

Beispiel: Philosophenproblem

Das Deadlock-Problem kann mit einer Ergänzung von Carel S. Scholten vermieden werden. Diese Ergänzung sieht vor, dass sich die Philosophen nur mit Hilfe eines Dieners zu Tisch setzen und wieder aufstehen dürfen und dass dieser Diener die Anweisung erhält, nur maximal vier der fünf Philosophen sich gleichzeitig setzen zu lassen:

$$
\begin{aligned}
DP &= P_0 \parallel \ldots \parallel P_4 \parallel G_0 \parallel \ldots \parallel G_4 \parallel D_0 \\
P_i &= (hinsetzen_i \rightarrow G_{i-1 \bmod 5}\_genommen\_von\_P_i \rightarrow \\
&\quad\; G_i\_genommen\_von\_P_i \rightarrow essen_i \rightarrow \\
&\quad\; G_{i-1 \bmod 5}\_hingelegt\_von\_P_i \rightarrow \\
&\quad\; G_i\_hingelegt\_von\_P_i \rightarrow aufstehen_i \rightarrow P_i) \\
D_0 &= x : H \rightarrow D_1 \\
D_i &= (x : H \rightarrow D_{i+1} \mid y : A \rightarrow D_{i-1}) \text{ für } i \in \{1,2,3\} \\
D_4 &= y : A \rightarrow D_3
\end{aligned}
$$

Dabei seien $H = \bigcup_{i=0}^{4} \{hinsetzen_i\}$ und $A = \bigcup_{i=0}^{4} \{aufstehen_i\}$

$$P \sqcap Q$$

- P und Q mit $\alpha P = \alpha Q = \alpha P \sqcap Q$.
- Dann ergibt $P \sqcap Q$ einen Prozess, der sich nichtdeterministisch für einen der beiden Prozesse P oder Q entscheidet.
- Diese Notation ist kein Hinweis auf die Implementierung. Es bleibt vollkommen offen, ob sich $P \sqcap Q$ immer für P, immer für Q oder nach unbekannten Kriterien oder Zufällen für P oder Q entscheidet.
- Beispiel eines Automaten, der sich nichtdeterministisch dafür entscheidet, ob er Tee oder Kaffee ausliefert:

$$\begin{aligned} GA &= (muenze \rightarrow (tee \rightarrow GA) \sqcap (kaffee \rightarrow GA)) \\ AT &= (muenze \rightarrow (tee \rightarrow AT \mid kaffee \rightarrow AT)) \\ KT &= (muenze \rightarrow kaffee \rightarrow KT) \end{aligned}$$

  Hier ist AT flexibel genug, damit bei $GA \parallel AT$ ein Deadlock ausgeschlossen ist. Bei $GA \parallel KT$ besteht die Gefahr eines Deadlocks, wenn sich GA nichtdeterministisch für die Ausgabe eines Tees entscheidet.

$P \,[\!]\, Q$

- P und Q mit $\alpha P = \alpha Q = \alpha(P \,[\!]\, Q)$.
- Anders als bei $P \sqcap Q$ hat die Umgebung eine Einflussmöglichkeit. Wenn das nächste Ereignis aus $\alpha(P \,[\!]\, Q)$ nur von P oder nur von Q akzeptiert werden kann, dann fällt die Entscheidung für den entsprechenden Zweig.
- Wenn jedoch das nächste Ereignis sowohl von P als auch Q akzeptiert werden kann, dann fällt die Entscheidung genauso nichtdeterministisch wie bei $P \sqcap Q$.
- $c \to P \,[\!]\, d \to Q \;=\; (c \to P \mid d \to Q)$, falls $c \neq d$.
- $c \to P \,[\!]\, d \to Q \;=\; (c \to P) \sqcap (d \to Q)$, falls $c = d$.
- $\mathit{traces}(P \,[\!]\, Q) = \mathit{traces}(P \sqcap Q) = \mathit{traces}(P) \cup \mathit{traces}(Q)$

$$P \,|||\, Q$$

- P und Q mit $\alpha P = \alpha(Q = \alpha P \,|||\, Q)$.
- Sowohl P als auch Q werden parallel verfolgt, aber jedes Ereignis aus $\alpha P \,|||\, Q)$ kann nur von einem der beiden Prozesse wahrgenommen werden. Wenn nur einer der beiden Prozesse das Ereignis akzeptieren kann, dann wird es von diesem akzeptiert. Andernfalls, wenn beide ein Ereignis akzeptieren können, dann fällt die Entscheidung nichtdeterministisch.

Beispiel: Philosophenproblem

Damit vereinfacht sich die Umsetzung des Philosophenproblems:

$$
\begin{aligned}
DP &= (P_0 \,|||\, \ldots \,|||\, P_4) \,||\, (G_0 \,|||\, \ldots \,|||\, G_4) \\
G_i &= (G_i\_genommen \rightarrow G_i\_hingelegt \rightarrow G_i) \\
P_i &= (G_{i-1 \bmod 5}\_genommen \rightarrow \\
&\quad G_i\_genommen \rightarrow \\
&\quad essen \rightarrow \\
&\quad G_{i-1 \bmod 5}\_hingelegt \rightarrow \\
&\quad G_i\_hingelegt \rightarrow P_i)
\end{aligned}
$$

Beispiel: Philosophenproblem

Ebenso vereinfacht sich die Fassung mit dem Diener:

$$
\begin{aligned}
DP &= (P_0 \,|||\, \ldots \,|||\, P_4) \,||\, (G_0 \,|||\, \ldots \,|||\, G_4) \,||\, (D \,|||\, D \,|||\, D \,|||\, D) \\
G_i &= (G_i\_genommen \to G_i\_hingelegt \to G_i) \\
P_i &= (hinsetzen \to G_{i-1 \bmod 5}\_genommen \to G_i\_genommen \to \\
&\quad\; essen \to \\
&\quad\; G_{i-1 \bmod 5}\_hingelegt \to G_i\_hingelegt \to aufstehen \to P_i) \\
D &= (hinsetzen \to aufstehen \to D)
\end{aligned}
$$

$$c.v$$

- Kommunikation wird mit Ereignissen der Form $c.v$ modelliert. Hierbei repräsentiert c den Kommunikationskanal und v den Inhalt der Nachricht über den Kanal.
- Zu jedem Kommunikationskanal c und jedem Prozess P gibt es das zugehörige Alphabet, das P über den Kanal c kommunizieren kann: $\alpha c(P) = \{v | c.v \in \alpha P\}$
- Ferner lässt sich definieren: $channel(c.v) = c, message(c.v) = v$

$$c!v \rightarrow P$$

- Sei c ein Kommunikationskanal, $c.v \in \alpha c(P)$, dann kann das Versenden einer Nachricht mit dem Operator „!“ explizit notiert werden: $(c!v \rightarrow P) = (c.v \rightarrow P)$
- Beim Empfang wird dann mit einer Variablen x gearbeitet, die an die Übertragungsereignis gebunden wird:
  $(c?x \rightarrow P(x)) = (y : \{y | channel(y) = c\} \rightarrow P(message(y)))$
- Ein Prozess *CopyBit* mit zwei Kommunikationskanälen *in* und *out*, der die Bits $\alpha in(CopyBit) = \alpha out(CopyBit) = \{0, 1\}$ überträgt:
  $CopyBit = (in?x \rightarrow out!x \rightarrow CopyBit)$
- Normalerweise gilt für zwei Prozesse P und Q, die parallel über einen Kanal kommunizieren $(P \parallel Q)$, dass $\alpha c(P) = \alpha c(Q)$.

- Ereignisse finden in CSP immer synchron statt – entsprechend erfolgt auch die Kommunikation synchron.
- Durch die Einführung von zwischenliegenden Puffern lässt sich eine asynchrone Kommunikation modellieren.
- Gegeben seien $P(c)$ und $Q(c)$, die über einen Kommunikationskanal c miteinander verbunden werden können: $P(c) \parallel Q(c)$.
- Wenn jedoch die synchrone Kopplung zwischen $P(c)$ und $Q(c)$ nicht gewünscht wird, lässt sich dies durch das Einfügen eines Puffers vermeiden:

$$Puffer(in, out) = \mu X : (in?x \rightarrow out!x \rightarrow X)$$

Wir haben dann zwei Kommunikationskanäle c_0 (zwischen P und dem *Puffer*) und c_1 (zwischen dem *Puffer* und Q):

$$P(c0) \parallel Puffer(c0, c1) \parallel Q(c1)$$

- Auch die Pufferkapazität lässt sich modellieren, indem mehrere Puffer hintereinander geschaltet werden.
- Ein Kommunikationskanal zwischen P und Q mit der Kapazität n lässt sich mit Hilfe der Kommunikationskanäle $c_0, \ldots, c_n$ modellieren:

$$P(c0) \parallel \mathit{Puffer}(c0, c1) \parallel \ldots \parallel \mathit{Puffer}(c_{n-1}, c_n) \parallel Q(c_n)$$

- Wenn Prozesse miteinander verknüpft werden, wird eine Abstrahierung möglich, die sich nur auf die Außenansicht beschränkt und die Betrachtung der Ereignisse weglässt, die nur intern stattfindet.
- Beispiel:
  $Puffer_3(in, out) = Puffer(in, c_1) \parallel Puffer(c_1, c_2) \parallel Puffer(c_2, out)$ ist ein FIFO-Puffer mit der Kapazität 3, bei dem in der weiteren Betrachtung die internen Kanäle c_1 und c_2 nicht weiter interessant sind und wir uns auf die äußeren Kanäle *in* und *out* beschränken können.

- Eine Abstraktion kann dann zu einer Divergenz führen, wenn es die Möglichkeit zu einer ungebundenen internen Sequenz von Ereignissen gibt.
- Beispiel: Folgende Pipeline besteht aus den beiden Prozessen P und Q, die auch untereinander die Nachricht *Hallo* kommunizieren:

$$
\begin{aligned}
DivPuffer(in, out) &= P(in, c) \parallel Q(c, out) \\
P(in, out) &= \mu X : (in?x \rightarrow out!x \rightarrow X \mid out!Hallo \rightarrow X) \\
Q(in, out) &= \mu X : (in?x \rightarrow out!x \rightarrow X \mid in?Hallo \rightarrow X)
\end{aligned}
$$

Rechenintensive parallele Anwendungen können nicht sinnvoll ohne Kenntnis der zugrundeliegenden Architektur erstellt werden.

Deswegen ist die Wahl einer geeigneten Architektur bzw. die Anpassung eines Algorithmus an eine Architektur von entscheidender Bedeutung für die effiziente Nutzung vorhandener Ressourcen.

Die Aufnahme von Steve Jurvetson (CC-AT-2.0, Wikimedia Commons) zeigt den 1965–1976 entwickelten Parallelrechner ILLIAC 4.

- Die sequentielle Arbeitsweise eines Prozessors kann durch verschiedene Parallelisierungstechniken beschleunigt werden (z.B. durch Pipelining oder die Möglichkeit, mehrere Instruktionen gleichzeitig auszuführen).
- Einzelne Operationen lassen sich auf größere Datenmengen gleichzeitig anwenden (z.B. für eine Vektoraddition).
- Die Anwendung wird aufgeteilt in unabhängig voneinander rechnende Teile, die miteinander kommunizieren (über gemeinsamen Speicher und/oder Nachrichten) und auf Mehrprozessorsystemen, Multicomputern oder mehreren unabhängigen Rechnern verteilt sind.

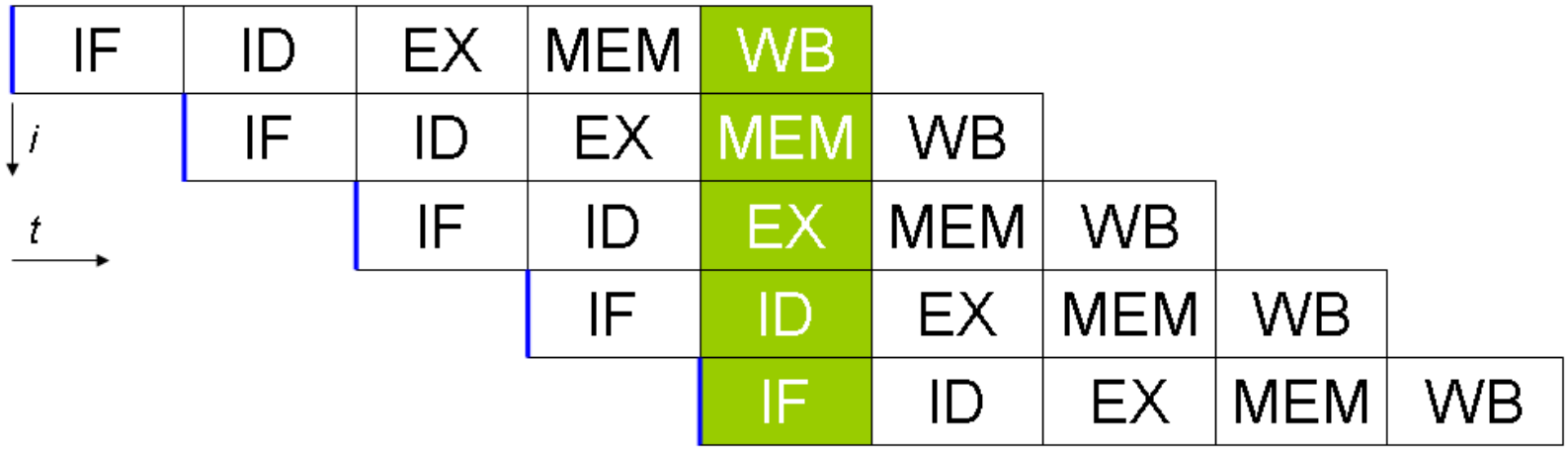
- Eine Anwendung wird durch eine Parallelisierung nicht in jedem Fall schneller.
- Es entstehen Kosten, die sowohl von der verwendeten Architektur als auch dem zum Einsatz kommenden Algorithmus abhängen.
- Dazu gehören:
  - Konfiguration
  - Kommunikation
  - Synchronisierung
  - Terminierung
- Interessant ist auch immer die Frage, wie die Kosten skalieren, wenn der Umfang der zu lösenden Aufgabe und/oder die zur Verfügung stehenden Ressourcen wachsen.

- Moderne Prozessoren arbeiten nach dem Fließbandprinzip: Über das Fließband kommen laufend neue Instruktionen hinzu und jede Instruktion wird nacheinander von verschiedenen Fließbandarbeitern bearbeitet.
- Dies parallelisiert die Ausführung, da unter günstigen Umständen alle Fließbandarbeiter gleichzeitig etwas tun können.
- Eine der ersten Pipelining-Architekturen war die IBM 7094 aus der Mitte der 60er-Jahre mit zwei Stationen am Fließband. Die UltraSPARC-IV-Architektur hat 14 Stationen.
- Die RISC-Architekturen (RISC = *reduced instruction set computer*) wurden speziell entwickelt, um das Potential für Pipelining zu vergrößern.
- Bei der Pentium-Architektur werden im Rahmen des Pipelinings die Instruktionen zuerst intern in RISC-Instruktionen konvertiert, so dass die x86-Architektur ebenfalls von diesem Potential profitieren kann.

Um zu verstehen, was alles innerhalb einer Pipeline zu erledigen ist, hilft ein Blick auf die möglichen Typen von Instruktionen:

- Operationen, die nur auf Registern angewendet werden und die das Ergebnis in einem Register ablegen.
- Instruktionen mit Speicherzugriff. Hier wird eine Speicheradresse berechnet und dann erfolgt entweder eine Lese- oder eine Schreiboperation.
- Sprünge.

Zerlegung in Arbeitsschritte



Eine einfache Aufteilung sieht folgende einzelne Schritte vor:

- Instruktion vom Speicher laden (IF)
- Instruktion dekodieren (ID)
- Instruktion ausführen, beispielsweise eine arithmetische Operation oder die Berechnung einer Speicheradresse (EX)
- Lese- oder Schreibzugriff auf den Speicher (MEM)
- Abspeichern des Ergebnisses in Registern (WB)

Das Diagramm wurde von Stéphane Tsacas geklaut.

- Bedingte Sprünge sind ein Problem für das Pipelining, da unklar ist, wie gesprungen wird, bevor es zur Ausführungsphase kommt.
- RISC-Maschinen führen typischerweise die Instruktion unmittelbar nach einem bedingten Sprung immer mit aus, selbst wenn der Sprung genommen wird. Dies mildert etwas den negativen Effekt für die Pipeline.
- Im übrigen gibt es die Technik der *branch prediction*, bei der ein Ergebnis angenommen wird und dann das Fließband auf den Verdacht hin weiterarbeitet, dass die Vorhersage zutrifft. Im Falle eines Misserfolgs muss dann u.U. recht viel rückgängig gemacht werden.
- Das ist machbar, solange nur Register verändert werden. Manche Architekturen verfolgen die Alternativen sogar parallel und haben für jedes abstrakte Register mehrere implementierte Register, die die Werte für die einzelnen Fälle enthalten.
- Die Vorhersage wird vom Übersetzer generiert. Typisch ist beispielsweise, dass bei Schleifen eine Fortsetzung der Schleife vorhergesagt wird.

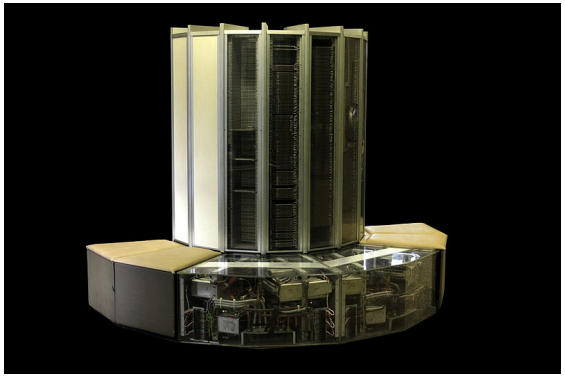
- Das Pipelining lässt sich dadurch noch weiter verbessern, wenn aus dem Speicher benötigte Werte frühzeitig angefragt werden.
- Moderne Prozessoren besitzen Caches, die einen schnellen Zugriff ermöglichen, deren Kapazität aber sehr begrenzt ist (dazu später mehr).
- Ebenfalls bieten moderne Prozessoren die Möglichkeit, das Laden von Werten aus dem Hauptspeicher frühzeitig zu beantragen – nach Möglichkeit so früh, dass sie rechtzeitig vorliegen, wenn sie dann benötigt werden (*software prefetch*).
- Speicherzugriffsmuster mit konstanten Abständen werden erkannt und führen zum Laden auf Verdacht hin (*hardware prefetch*).

Flynn schlug 1972 folgende Klassifizierung vor in Abhängigkeit der Zahl der Instruktions- und Datenströme:

Instruktionen	Daten	Bezeichnung
1	1	SISD (Single Instruction Single Data)
1	> 1	SIMD (Single Instruction Multiple Data)
> 1	1	MISD (Multiple Instruction Single Data)
> 1	> 1	MIMD (Multiple Instruction Multiple Data)

SISD entspricht der klassischen von-Neumann-Maschine, SIMD sind z.B. vektorisierte Rechner, MISD wurde wohl nie umgesetzt und MIMD entspricht z.B. Mehrprozessormaschinen oder Clustern. Als Klassifizierungsschema ist dies jedoch zu grob.

- Die klassische Variante der SIMD sind die Array-Prozessoren.
- Eine Vielzahl von Prozessoren steht zur Verfügung mit zugehörigem Speicher, die diesen in einer Initialisierungsphase laden.
- Dann wird die gleiche Instruktion an alle Prozessoren verteilt, die jeder Prozessor auf seinen Daten ausführt.
- Die Idee geht auf S. H. Unger 1958 zurück und wurde mit dem ILLIAC IV zum ersten Mal umgesetzt.
- Die heutigen GPUs übernehmen teilweise diesen Ansatz.



Bei Vektor-Prozessoren steht zwar nur ein Prozessor zur Verfügung, aber dieser ist dank dem Pipelining in der Lage, pro Taktzyklus eine Operation auf einem Vektor umzusetzen. Diese Technik wurde zuerst von der Cray-1 im Jahr 1974 umgesetzt und auch bei späteren Cray-Modellen verfolgt.

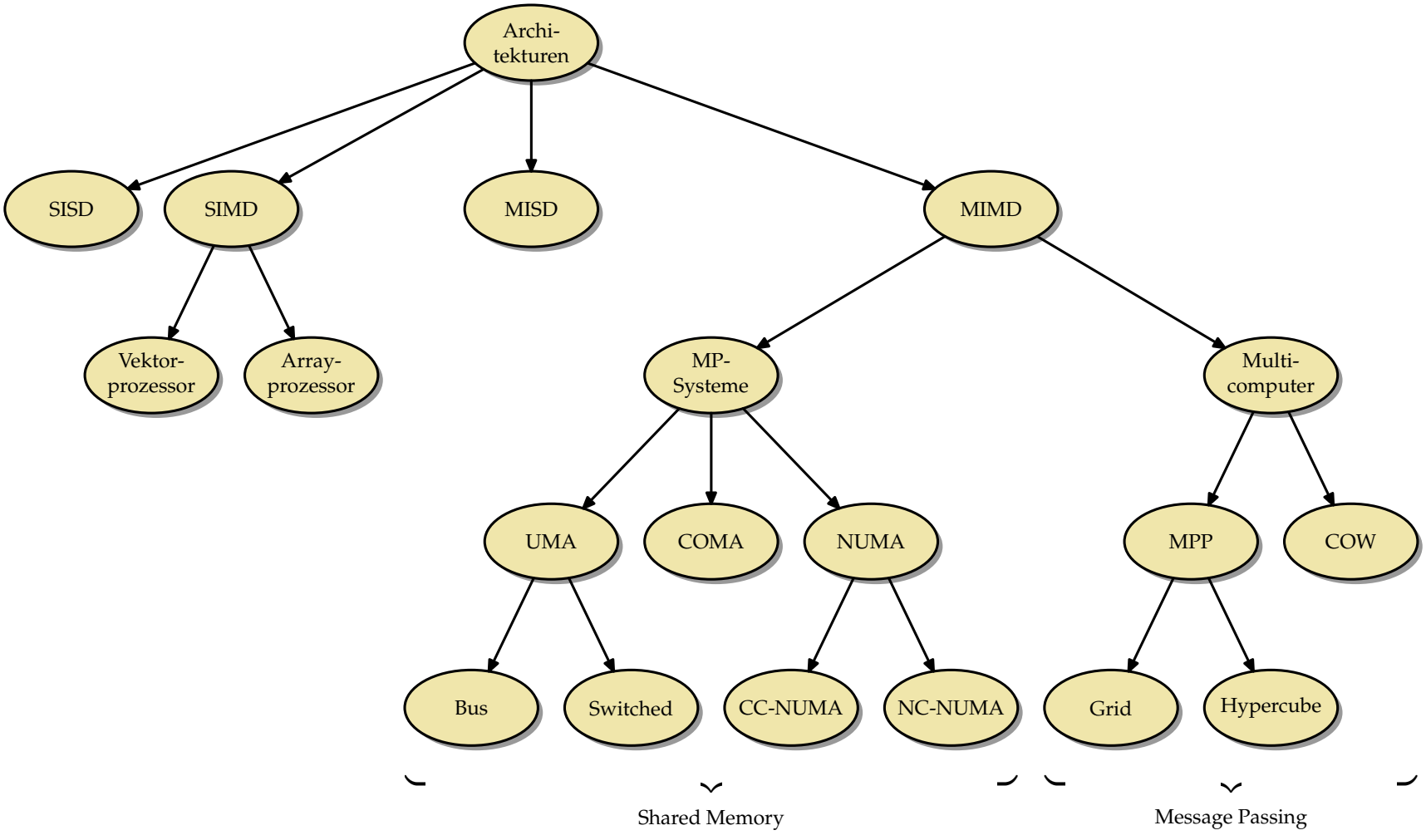
Die MMX- und SSE-Instruktionen des Pentium 4 setzen ebenfalls dieses Modell um.

Die von Rama (Wikimedia Commons, Cc-by-sa-2.0-fr) gefertigte Aufnahme zeigt eine an der EPFL in Lausanne ausgestellte Cray-1.

Hier wird unterschieden, ob die Kommunikation über gemeinsamen Speicher oder ein gemeinsames Netzwerk erfolgt:

- Multiprozessor-Systeme (MP-Systeme) erlauben jedem Prozessor den Zugriff auf den gesamten zur Verfügung stehenden Speicher. Der Speicher kann auf gleichförmige Weise allen Prozessoren zur Verfügung stehen (UMA = *uniform memory access*) oder auf die einzelnen Prozessoren oder Gruppen davon verteilt sein (NUMA = *non-uniform memory access*).
- Multicomputer sind über spezielle Topologien vernetzte Rechnersysteme, bei denen die einzelnen Komponenten ihren eigenen Speicher haben. Üblich ist hier der Zusammenschluss von Standardkomponenten (COW = *cluster of workstations*) oder spezialisierter Architekturen und Bauweisen im großen Maßstab (MPP = *massive parallel processors*).

Architekturen nach Tanenbaum



- Die Theseus gehört mit vier Prozessoren des Typs UltraSPARC IV+ mit jeweils zwei Kernen zu der Familie der Multiprozessorsysteme (MP-Systeme).
- Da der Speicher zentral liegt und alle Prozessoren auf gleiche Weise zugreifen, gehört die Theseus zur Klasse der UMA-Architekturen (*Uniform Memory Access*) und dort zu den Systemen, die Bus-basiert Cache-Kohärenz herstellen (dazu später mehr).
- Die Thales hat zwei Xeon-5650-Prozessoren mit jeweils 6 Kernen, die jeweils zwei Threads unterstützen. Wie bei der Theseus handelt es sich um eine UMA-Architektur, die ebenfalls Bus-basiert Cache-Kohärenz herstellt.
- Die Theon hat einen Xeon-E5-2650-Prozessor mit 12 Kernen, die ebenfalls je zwei Threads unterstützen.

- Die Maschinen im Pool-Raum E.44 sowie die Livingstone haben je einen Intel-i5-3470-Prozessor mit vier Kernen.
- Die Livingstone ist zusätzlich mit einer Nvidia-Quadro-P620-Grafikkarte ausgestattet. Diese hat 2 GB Speicher, vier Multiprozessoren und insgesamt 512 CUDA-Kerne (128 pro Multiprozessor).
- Die Grafikkarte hat eine SIMD-Architektur, die sowohl Elemente der Array- als auch der Vektorrechner vereinigt und auch den Bau von Pipelines ermöglicht.

- Die Schnittstelle für Threads ist eine Abstraktion des Betriebssystems (oder einer virtuellen Maschine), die es ermöglicht, mehrere Ausführungsfäden, jeweils mit eigenem Stack und PC ausgestattet, in einem gemeinsamen Adressraum arbeiten zu lassen.
- Der Einsatz lohnt sich insbesondere auf Mehrprozessormaschinen (bzw. Prozessoren mit mehreren Kernen) mit gemeinsamen Speicher.
- Vielfach wird die Fehleranfälligkeit kritisiert wie etwa von C. A. R. Hoare in *Communicating Sequential Processes*: „In its full generality, multithreading is an incredibly complex and error-prone technique, not to be recommended in any but the smallest programs.“

- Wie die *comp.os.research* FAQ belegt, gab es Threads bereits lange vor der Einführung von Mehrprozessormaschinen:
  „The notion of a thread, as a sequential flow of control, dates back to 1965, at least, with the Berkeley Timesharing System. Only they weren't called threads at that time, but processes [Dijkstra, 1965]. Processes interacted through shared variables, semaphores, and similar means. Max Smith did a prototype threads implementation on Multics around 1970; it used multiple stacks in a single heavyweight process to support background compilations."
  `http://www.serpentine.com/blog/threads-faq/the-history-of-threads/`
- UNIX selbst kannte zunächst nur Prozesse, d.h. jeder Thread hatte seinen eigenen Adressraum.

- Zu den ersten UNIX-Implementierungen, die Threads unterstützten, gehörten der Mach-Microkernel (entwickelt at der Carnegie Mellon University, eingebettet in NeXT, später Mac OS X) und Solaris (zur Unterstützung der von Sun ab 1992 hergestellten Multiprozessormaschinen). Heute unterstützen alle UNIX-Varianten einschließlich Linux Threads.
- 1995 wurde durch die *The Open Group* (einer Standardisierungsgesellschaft für UNIX) mit POSIX 1003.1c-1995 eine standardisierte Threads-Bibliotheksschnittstelle publiziert, die 1996 von der IEEE, dem ANSI und der ISO übernommen wurde.
- Diverse andere Bibliotheken für Threads existierten und existieren (u.a. von Microsoft und von Sun), sind aber nicht portabel und daher inzwischen von geringerer Bedeutung.

- Spezifikation der *Open Group*:
  `https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html`
- Unterstützt
  - das Erzeugen von Threads und das Warten auf ihr Ende,
  - den gegenseitigen Ausschluss (notwendig, um auf gemeinsame Datenstrukturen zuzugreifen),
  - Bedingungsvariablen (*condition variables*), die einem Prozess signalisieren können, dass sich eine Bedingung erfüllt hat, auf die gewartet wurde,
  - Lese- und Schreibsperren, um parallele Lese- und Schreibzugriffe auf gemeinsame Datenstrukturen zu synchronisieren.
- Freie Implementierungen der Schnittstelle für C:
  - GNU Portable Threads:
    `http://www.gnu.org/software/pth/`
  - Native POSIX Thread Library:
    `http://people.redhat.com/drepper/nptl-design.pdf`

- Seit dem C++-Standard ISO 14882-2012 (C++11) werden POSIX-Threads direkt unterstützt, wobei die POSIX-Schnittstelle in eine für C++ geeignete Weise verpackt ist.
- Diese Schnittstelle basiert auf der zuvor entwickelten Boost-Schnittstelle für Threads.
- Die folgende Einführung bezieht sich auf C++11 bzw. auf die späteren Standards C++14 und C++17.

- Der von einem Thread auszuführende Programmtext wird in C++ immer durch ein sogenanntes Funktionsobjekt repräsentiert.
- In C++ sind alle Objekte Funktionsobjekte, die den parameterlosen Funktionsoperator unterstützen.
- Das könnte im einfachsten Falle eine ganz normale parameterlose Funktion sein:

```
void f() {
   // do something
}
```

- Das ist jedoch nicht sehr hilfreich, da wegen der fehlenden Parametrisierung unklar ist, welche Teilaufgabe die Funktion für einen konkreten Thread erfüllen soll.

```
class Task {
   public:
      Task( /* parameters */ );
      void operator()() {
         // do something in dependence of the parameters
      }
   private:
      // parameters of this task
};
```

- Eine Klasse für Funktionsobjekte muss den parameterlosen Funktionsoperator unterstützen, d.h. **void operator**()().
- Im privaten Bereich der *Task*-Klasse können alle Parameter untergebracht werden, die für die Ausführung benötigt werden.
- Der Konstruktor erhält die Parameter einer *Task* und kann diese dann in den privaten Bereich kopieren.
- Dann kann die parameterlose Funktion problemlos auf ihre Parameter zugreifen.

Repräsentierung einer *Task*

fork-and-join.cpp

```
class Task {
   public:
      Task(int id) : id(id) {};
      void operator()() {
         std::cout << "task " << id << " is being worked on" <<
            std::endl;
      }
   private:
      const int id;
};
```

- In diesem einfachen Beispiel wird nur ein einziger Parameter für eine *Task* verwendet: *id*
- Häufig genügt bereits ein Parameter, der die Identität festlegt.
- Für Demonstrationszwecke gibt der Funktionsoperator nur seine eigene *id* aus.
- So ein Funktionsobjekt kann auch ohne Threads erzeugt und benutzt werden:
  Task $t(7)$; $t()$;

fork-and-join.cpp

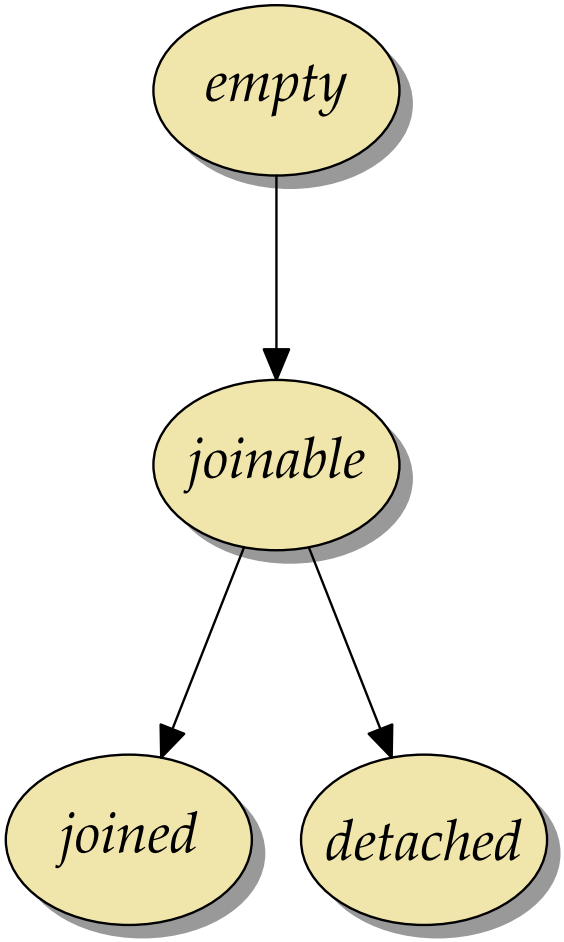
```
#include <iostream>
#include <thread>

// class Task...

int main() {
   // fork off some threads
   std::thread t1(Task(1)); std::thread t2(Task(2));
   std::thread t3(Task(3)); std::thread t4(Task(4));
   // and join them
   std::cout << "Joining..." << std::endl;
   t1.join(); t2.join(); t3.join(); t4.join();
   std::cout << "Done!" << std::endl;
}
```

- Objekte des Typs *std*::*thread* (aus **#include** <thread>) können mit einem Funktionsobjekt initialisiert werden. Die Threads werden sofort aktiv.
- Mit der *join*-Methode wird auf die Beendigung des jeweiligen Threads gewartet.

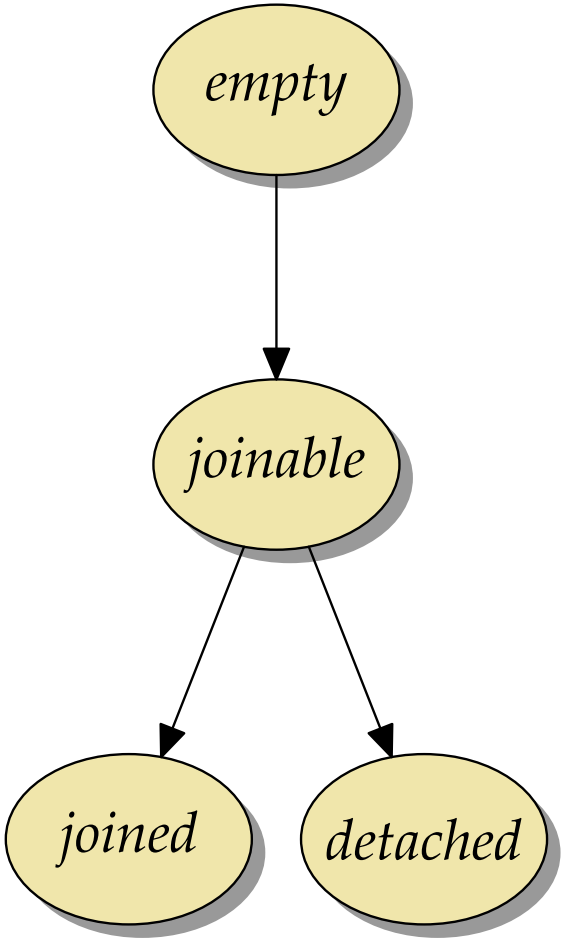
$$P_i = (fork \rightarrow join \rightarrow SKIP)$$

- Beim Fork-And-Join-Pattern werden beliebig viele einzelne Threads erzeugt, die dann unabhängig voneinander arbeiten.
- Entsprechend bestehen die Alphabete nur aus *fork* und *join*.
- Das Pattern eignet sich für Aufgaben, die sich leicht in unabhängig voneinander zu lösende Teilaufgaben zerlegen lassen.
- Die Umsetzung in C++ sieht etwas anders aus mit $\alpha P_i = \{fork_i, join_i\}$ und $\alpha M = \alpha P = \bigcup_{i=1}^{n} \alpha P_i$:

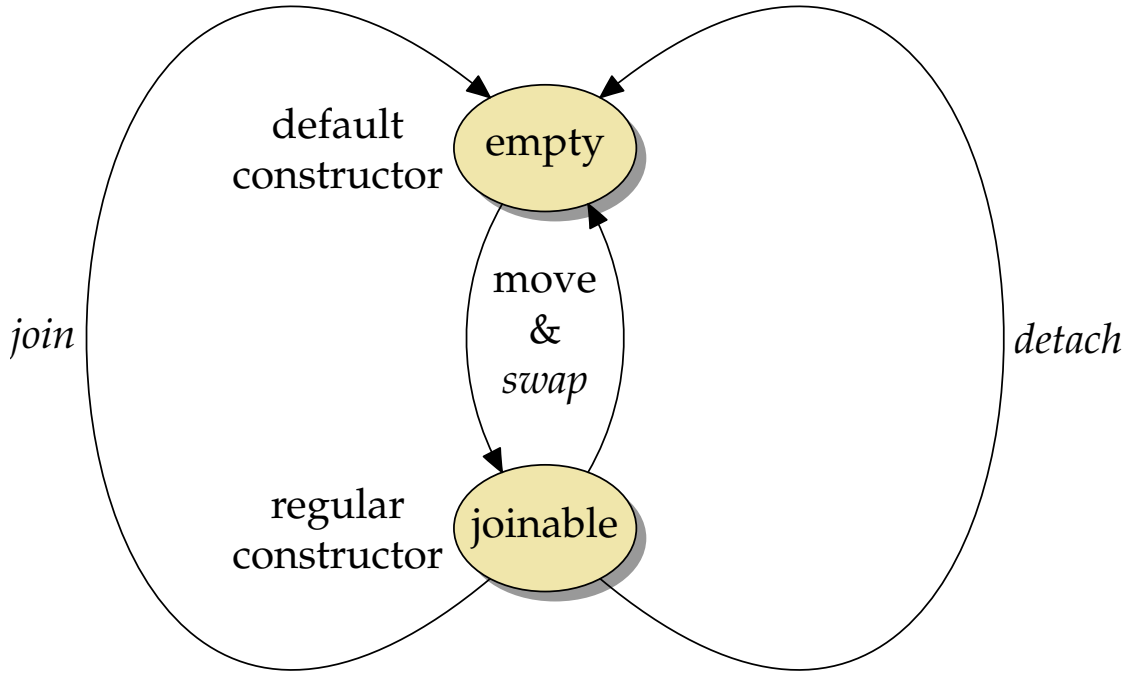
$$P = M \parallel P_1 \parallel \ldots \parallel P_n$$
$$M = (fork_1 \rightarrow \cdots \rightarrow fork_n \rightarrow join_1 \rightarrow \cdots \rightarrow join_n \rightarrow SKIP)$$
$$P_i = (fork_i \rightarrow join_i \rightarrow SKIP)$$



- Objekte des Typs *std*::*thread* sind RAII-Objekte (RAII = *resource acquisition is initialization*), die mit einer Ressource (hier der POSIX-Thread) fest verknüpft sind.
- Wenn ein *std*::*thread*-Objekt mit einem Funktionsobjekt erzeugt wird, dann wird bereits beim Konstruieren der POSIX-Thread erzeugt und das Objekt ist im Zustand *joinable*.
- Mit der *join*-Methode erreicht das Objekt den Zustand *joined*, in der der Thread beendet ist.



- Mit der *detach*-Methode kann ein *std*::*thread*-Objekt dauerhaft von dem zugehörigen Thread getrennt werden. Der Thread läuft dann im Hintergrund weiter. Eine Synchronisierung ist dann nicht mehr möglich.
- Die Zustände *joined* und *detached* sind äquivalent zum Zustand *empty*.
- Mit der *joinable*-Methode kann abgefragt werden, ob sich ein *std*::*thread*-Objekt im *joinable*-Zustand befindet.
- Wenn beim Abbau eines *std*::*thread*-Objekts dieses sich im *joinable*-Zustand befindet, wird vom Destruktor *std*::*terminate* aufgerufen.



- Prinzipiell gibt es nur zwei Zustände: *empty* und *joinable*.
- Mit dem Default-Konstruktor wird ein „leeres“ Objekt erzeugt; mit dem Konstruktor, der ein Funktionsobjekt erhält, wird unmittelbar ein Thread erzeugt, der dieses ausführt.
- Der Move-Konstruktor, das Move-Assignment und die *swap*-Operation sind allesamt äquivalent.

destructing-joinable-thread.cpp

```
int main() {
   {
      std::thread t1(Task(1));
      // t1 will now be destructed in the joinable state
   }
}
```

- *std*::*thread*-Objekte dürfen nicht abgebaut werden, wenn sie noch *joinable* sind. Andernfalls wird *std*::*terminate* aufgerufen:

```
clonmel$ destructing-joinable-thread
terminate called without an active exception
Abort (core dumped)
clonmel$
```

assigning-joinable-thread.cpp

```
int main() {
   {
      std::thread t1(Task(1)); // joinable state
      std::thread t2; // empty state
      t2 = std::thread(Task(2)); // move assignment
      std::thread t3; // empty state
      t3 = std::move(t2); // t3 now joinable, t2 empty
      /* reached */
      t1 = std::move(t3); // not permitted as t1 is joinable
      /* not reached */
   }
}
```

- Auf der linken Seite der Zuweisung darf kein *std*::*thread*-Objekt im Zustand *joinable* sein.

detached-thread.cpp

```
#include <chrono>
#include <iostream>
#include <thread>

class Task {
   public:
      Task(int id) : id(id) {};
      void operator()() {
         std::cout << "task " << id << " starts" << std::endl;
         std::this_thread::sleep_for(std::chrono::seconds(2));
         std::cout << "task " << id << " ends" << std::endl;
      }
   private:
      const int id;
};

int main() {
   {
      std::thread t1(Task(1));
      t1.detach();
   }
   std::this_thread::sleep_for(std::chrono::seconds(1));
   std::cout << "main exits" << std::endl;
}
```

- Ein *std*::*thread*-Objekt kann im Zustand *detached* abgebaut werden.
- Der C++11-Standard verrät nichts darüber, was passiert, wenn solche Threads noch weiterlaufen, wenn *main* endet.
- Da auf einem POSIX-System der Thread nicht länger als der Prozess laufen kann, terminiert dieser, wenn es explizit oder implizit zu *exit* kommt.
- Da nach dem Ende von *main* auch noch die statischen Objekte abgebaut werden, kann es zu undefinierten Effekten kommen, wenn der Thread auf diese noch zugreifen sollte.
- Somit ist *detach* normalerweise nicht empfehlenswert.

detached-thread.cpp

```
std::this_thread::sleep_for(std::chrono::seconds(2));
```

- *std::thread::this_thread* ist ein Namensraum mit einigen Funktionen, die sich implizit auf den aufrufenden Thread beziehen.
- Angebotene Funktionen:

thread::*id get_id*()	liefert die ID des aktuellen Threads (ist implementierungsabhängig)
void *yield*()	signalisiert, dass andere ausführungsbereite Threads eher ausgeführt werden sollten
void *sleep_until*(...)	bis zu einem Zeitpunkt suspendieren
void *sleep_for*(...)	für die genannte Zeitperiode suspendieren

fork-and-join2.cpp

```
// fork off some threads
std::thread threads[10];
for (int i = 0; i < 10; ++i) {
   threads[i] = std::thread(Task(i));
}
```

- Wenn Threads in Datenstrukturen unterzubringen sind (etwa Arrays oder beliebigen Containern), dann werden sie dort normalerweise im leeren Zustand konstruiert.
- Später kann dann mit Hilfe eines Move-Assignments ein Thread zugewiesen wird.
- Hier ist auf der rechten Seite ein temporäres Objekt mit dem Typ *std*::*thread*&&, das ohne weiteres Zutun zur Verwendung des Move-Assignments führt.
- Im Anschluss an die Zuweisung hat die linke Seite den Verweis auf den Thread, während die rechte Seite dann nur noch eine leere Hülle ist.

fork-and-join2.cpp

```
// and join them
std::cout << "Joining..." << std::endl;
for (int i = 0; i < 10; ++i) {
   threads[i].join();
}
```

- Das vereinfacht dann auch das Zusammenführen all der Threads mit der *join*-Methode.

simpson.cpp

```
double simpson(double (*f)(double), double a, double b,
      std::size_t n) {
   assert(n > 0 && a <= b);
   double value = f(a)/2 + f(b)/2;
   double xleft; double x = a;
   for (std::size_t i = 1; i < n; ++i) {
      xleft = x; x = a + i * (b - a) / n;
      value += f(x) + 2 * f((xleft + x)/2);
   }
   value += 2 * f((x + b)/2); value *= (b - a) / n / 3;
   return value;
}
```

- *simpson* setzt die Simpsonregel für das in n gleichlange Teilintervalle aufgeteilte Intervall $[a, b]$ für die Funktion f um:

$$S(f, a, b, n) = \frac{h}{3}\left(\frac{1}{2}f(x_0) + \sum_{k=1}^{n-1} f(x_k) + 2\sum_{k=1}^{n} f(\frac{x_{k-1} + x_k}{2}) + \frac{1}{2}f(x_n)\right)$$

mit $h = \frac{b-a}{n}$ und $x_k = a + k \cdot h$.

simpson.cpp

```
class SimpsonTask {
   public:
      SimpsonTask(double (*f)(double), double a, double b,
                  std::size_t n, double& rp) :
            f(f), a(a), b(b), n(n), rp(rp) {
      }
      void operator()() {
         rp = simpson(f, a, b, n);
      }
   private:
      double (*f)(double);
      double a, b;
      std::size_t n;
      double& rp;
};
```

- Jedem Objekt werden nicht nur die Parameter der *simpson*-Funktion übergeben, sondern auch noch einen Zeiger auf die Variable, wo das Ergebnis abzuspeichern ist.

simpson.cpp

```
double mt_simpson(double (*f)(double), double a, double b,
      std::size_t n, std::size_t nofthreads) {
   // divide the given interval into nofthreads partitions
   assert(n > 0 && a <= b && nofthreads > 0);
   std::size_t nofintervals = n / nofthreads;
   std::size_t remainder = n % nofthreads;
   std::size_t interval = 0;

   std::vector<std::thread> threads(nofthreads);
   std::vector<double> results(nofthreads);

   // fork & join & collect results ...
}
```

- *mt_simpson* ist wie die Funktion *simpson* aufzurufen – nur ein Parameter *nofthreads* ist hinzugekommen, der die Zahl der zur Berechnung zu verwendenden Threads spezifiziert.
- Dann muss die Gesamtaufgabe entsprechend in Teilaufgaben zerlegt werden.

simpson.cpp

```
std::vector<std::thread> threads(nofthreads);
std::vector<double> results(nofthreads);
```

- Variabel lange Arrays auf dem Stack sind in C++ nicht zulässig.
- C++ ist hier restriktiver als C99 bzw. C11 und dies wird sich auch nicht ändern, da dies inzwischen als eine Fehlentwicklung angesehen wird.
- *g++* lässt es zwar zu – dennoch sollte es vermieden werden.
- Stattdessen empfiehlt sich die Verwendung von *std::vector* aus *<vector>*. Beim Konstruktor kann die gewünschte Dimensionierung angegeben werden (in runden Klammern!).
- Die im Vektor enthaltenen Objekte leben dann auf dem Heap.

simpson.cpp

```
double x = a;
for (std::size_t i = 0; i < nofthreads; ++i) {
   std::size_t intervals = nofintervals;
   if (i < remainder) ++intervals;
   interval += intervals;
   double xleft = x; x = a + interval * (b - a) / n;
   threads[i] = std::thread(SimpsonTask(f,
      xleft, x, intervals, results[i]));
}
```

- Für jedes Teilproblem wird ein entsprechendes Funktionsobjekt temporär angelegt, an *std*::*thread* übergeben, womit ein Thread erzeugt wird und schließlich an *threads*[*i*] mit der Verlagerungs-Semantik zugewiesen.
- Hierbei wird auch implizit der Verlagerungs- oder Kopierkonstruktor von *SimpsonThread* verwendet.

simpson.cpp

```
double sum = 0;
for (std::size_t i = 0; i < nofthreads; ++i) {
   threads[i].join();
   sum += results[i];
}
return sum;
```

- Wie gehabt erfolgt die Synchronisierung mit der *join*-Methode.
- Danach kann das entsprechende Ergebnis abgeholt und aggregiert werden.

simpson.cpp

```
cmdname = *argv++; --argc;
if (argc > 0) {
   std::istringstream arg(*argv++); --argc;
   if (!(arg >> N) || N <= 0) usage();
}
if (argc > 0) {
   std::istringstream arg(*argv++); --argc;
   if (!(arg >> nofthreads) || nofthreads <= 0) usage();
}
if (argc > 0) usage();
```

- Es ist sinnvoll, die Zahl der zu startenden Threads als Kommandozeilenargument (oder alternativ über eine Umgebungsvariable) zu übergeben, da dieser Parameter von den gegebenen Rahmenbedingungen abhängt (Wahl der Maschine, zumutbare Belastung).
- Zeichenketten können in C++ wie Dateien ausgelesen werden, wenn ein Objekt des Typs *std*::*istringstream* damit initialisiert wird.
- Das Einlesen erfolgt in C++ mit dem überladenen >>-Operator, der als linken Operanden einen Stream erwartet und als rechten eine Variable.

simpson.cpp

```
double sum = mt_simpson(f, a, b, N, nofthreads);
std::cout << std::setprecision(14) << sum << std::endl;
std::cout << std::setprecision(14) << M_PI << std::endl;
```

- Testen Sie Ihr Programm zuerst immer ohne Threads, indem die Funktion zur Lösung eines Teilproblems verwendet wird, um das Gesamtproblem unparallelisiert zu lösen. (Diese Variante ist hier auskommentiert.)
- *std::cout* ist in C++ die Standardausgabe, die mit dem <<-Operator auszugebende Werte erhält.
- *std*::*setprecision*(14) setzt die Zahl der auszugebenden Stellen auf 14. *endl* repräsentiert einen Zeilentrenner.
- Zum Vergleich wird hier *M_PI* ausgegeben, weil zum Testen $f(x) = \frac{4}{1+x^2}$ verwendet wurde, wofür $\int_0^1 f(x)dx = 4 \cdot \arctan(1) = \pi$ gilt.

simpson2.cpp

```
threads[i] = std::thread([=,&results]() {
   results[i] = simpson(f, xleft, x, intervals);
});
```

- Die Lösung vereinfacht sich, wenn die Klasse *SimpsonTask* durch einen Lambda-Ausdruck ersetzt wird.
- Die *capture*, d.h. der Teil in den eckigen Klammern zu Beginn des Lambda-Ausdrucks ([=,&*results*]) spezifiziert, dass per Voreinstellung alle aus der Umgebung benötigten lokalen Variablen kopiert werden, *results* jedoch als Referenz bezogen wird.
- Lambda-Ausdrücke führen zum Erzeugen einer entsprechenden anonymen Klasse, wobei die *capture* festgelegt wird, was wie in ein zu konstruierendes Objekt übernommen wird.

$$
\begin{aligned}
MX &= M \,\|\, (P_1 \,|||\, \ldots \,|||\, P_n) \\
M &= (lock \rightarrow unlock \rightarrow M) \\
P_i &= (lock \rightarrow begin\_critical\_region_i \rightarrow \\
&\quad end\_critical\_region_i \rightarrow unlock \rightarrow \\
&\quad private\_work_i \rightarrow P_i)
\end{aligned}
$$

- n Prozesse $P_1 \ldots P_n$ wollen konkurrierend auf eine Ressource zugreifen, aber zu einem Zeitpunkt soll diese nur einem Prozess zur Verfügung stehen.
- Die Ressource wird von M verwaltet.
- Wegen dem $|||$-Operator wird unter den P_i jeweils ein Prozess nicht-deterministisch ausgewählt, der für *lock* bereit ist.

task-queue.hpp

```
template<typename Task>
class TaskQueue {
   public:
      void submit(Task task) {
         mutex.lock();
         queue.emplace_back(std::move(task));
         mutex.unlock();
      }
      bool fetch(Task& task) {
         mutex.lock();
         bool success;
         if (queue.empty()) {
            success = false;
         } else {
            success = true;
            task = std::move(queue.front());
            queue.pop_front();
         }
         mutex.unlock();
         return success;
      }
   private:
      std::mutex mutex;
      std::deque<Task> queue;
};
```

task-queue.hpp

```
void submit(Task task) {
   mutex.lock();
   queue.emplace_back(std::move(task));
   mutex.unlock();
}
```

- Bei Threads wird ein gegenseitiger Ausschluss über sogenannte Mutex-Variablen (*mutual exclusion*) organisiert.
- Dies wird primär dazu genutzt, um auf gemeinsame Datenstrukturen (wie hier die *TaskQueue*) konkurrierend zuzugreifen.
- Sogenannte „kritische Regionen“ werden dann von *mutex.lock*() und *mutex.unlock* eingeschlossen.
- Das stellt sicher, dass sich nie mehr als ein Thread innerhalb einer kritischen Region befindet.

- Gemeinsame Datenstrukturen, auf die konkurrierend zugegriffen wird, sollten grundsätzlich in eine Klasse verpackt werden, bei der
  - die Daten alle **private** deklariert sind und
  - Zugriffe auf die Daten in den Methoden immer innerhalb einer kritischen Regionen stattfinden.
- Problem: Was passiert, wenn der Aufruf einer Funktion wie z.B. *queue*.*emplace_back*(*std*::*move*(*task*)) zu einer Ausnahmenbehandlung führt? Dann wird die Ausführung von *submit* abrupt beendet, die *mutex* bleibt weiterhin gesperrt, so dass anschließend jeder Aufruf einer der Methoden dieser *TasksQueue* zu einem Deadlock führt.
- Die Lösung besteht in der Verwendung von RAII-Objekten.

```
template<typename Mutex>
class lock_guard {
   public:
      lock_guard(Mutex& mutex) : mutex(mutex) {
         mutex.lock();
      }
      ~lock_guard() {
         mutex.unlock();
      }
      lock_guard(const lock_guard&) = delete;
      lock_guard& operator=(const lock_guard&) = delete;
   private:
      Mutex& mutex;
};
```

- Der Standard bietet hierfür *std*::*lock_guard* an, das (etwas vereinfacht) so aussieht.

RAII-Objekte für kritische Regionen

task-queue.hpp

```
template<typename Task>
class TaskQueue {
   public:
      void submit(Task task) {
         std::lock_guard<std::mutex> lock(mutex);
         queue.emplace_back(std::move(task));
      }
      bool fetch(Task& task) {
         std::lock_guard<std::mutex> lock(mutex);
         if (queue.empty()) {
            return false;
         } else {
            task = std::move(queue.front());
            queue.pop_front();
            return true;
         }
      }
   private:
      std::mutex mutex;
      std::deque<Task> queue;
};
```

philo.cpp

```
int main() {
   constexpr unsigned int PHILOSOPHERS = 5;
   std::thread philosopher[PHILOSOPHERS];
   std::mutex fork[PHILOSOPHERS];
   for (int i = 0; i < PHILOSOPHERS; ++i) {
      philosopher[i] = std::thread(Philosopher(i+1,
         fork[i], fork[(i + PHILOSOPHERS - 1) % PHILOSOPHERS]));
   }
   for (int i = 0; i < PHILOSOPHERS; ++i) {
      philosopher[i].join();
   }
}
```

- Die Philosophen können auf Basis von Threads leicht umgesetzt werden, wobei die Gabeln als Mutex-Variablen repräsentiert werden.

philo.cpp

```
class Philosopher {
   public:
      Philosopher(unsigned int id, std::mutex& left_fork,
            std::mutex& right_fork) :
         id(id), left_fork(left_fork), right_fork(right_fork) {
      }
      void operator()() { /* ... */ }
   private:
      void print_status(const std::string& msg) const {
         std::lock_guard<std::mutex> lock(cout_mutex);
         std::cout << "philosopher [" << id << "]: " <<
            msg << std::endl;
      }
      unsigned int id;
      std::mutex& left_fork;
      std::mutex& right_fork;
};
```

- Dabei erhält jeder Philosoph Referenzen auf die ihm zugeordneten beiden Gabeln.

philo.cpp

```
    void operator()() {
       for (int i = 0; i < 5; ++i) {
          print_status("sits down at the table");
          {
             std::lock_guard<std::mutex> lock1(left_fork);
             print_status("picks up the left fork");
             {
                std::lock_guard<std::mutex> lock2(right_fork);
                print_status("picks up the right fork");
                {
                   print_status("is dining");
                }
             }
             print_status("returns the right fork");
          }
          print_status("returns the left fork");
          print_status("leaves the table");
       }
    }
```

- Alle Philosophen nehmen hier zunächst die linke und dann die rechte Gabel. Bei dieser Vorgehensweise ist ein Deadlock möglich, wenn beispielsweise alle gleichzeitig die linke Gabel nehmen.

$$P = (P_1 \,|||\, P_2) \parallel MX_1 \parallel MX_2$$
$$MX_1 = (lock_1 \rightarrow unlock_1 \rightarrow MX_1)$$
$$MX_2 = (lock_2 \rightarrow unlock_2 \rightarrow MX_2)$$
$$P_1 = (lock_1 \rightarrow think_1 \rightarrow lock_2 \rightarrow unlock_2 \rightarrow unlock_1 \rightarrow P_1)$$
$$P_2 = (lock_2 \rightarrow think_2 \rightarrow lock_1 \rightarrow unlock_1 \rightarrow unlock_2 \rightarrow P_2)$$

- Wenn P_1 und P_2 beide sofort jeweils ihren ersten Lock erhalten, gibt es einen Deadlock:

$$\nexists\, t : t \in traces(P) \wedge t > \langle lock_1, think_1, lock_2, think_2 \rangle$$

- Eine der von Dijkstra vorgeschlagenen Deadlock-Vermeidungsstrategien (siehe EWD625) sieht die Definition einer totalen Ordnung aller MX_i vor, die z.B. durch die Indizes zum Ausdruck kommen kann.
- D.h. wenn MX_i und MX_j gehalten werden sollen, dann ist zuerst MX_i zu belegen, falls $i < j$, ansonsten MX_j.
- Dijkstra betrachtet den Ansatz selbst als unschön, weil damit eine willkürliche Anordnung erzwungen wird. Die Vorgehensweise ist nicht immer praktikabel, aber nicht selten eine sehr einfach umzusetzende Lösung.

philo2.cpp

```
struct Fork {
   unsigned id; std::mutex mutex;
};
/* ... */
int main() {
   constexpr unsigned int PHILOSOPHERS = 5;
   std::thread philosopher[PHILOSOPHERS];
   Fork fork[PHILOSOPHERS];
   for (int i = 0; i < PHILOSOPHERS; ++i) {
      fork[i].id = i;
   }
   for (int i = 0; i < PHILOSOPHERS; ++i) {
      philosopher[i] = std::thread(Philosopher(i+1,
         fork[i], fork[(i + PHILOSOPHERS - 1) % PHILOSOPHERS]));
   }
   for (int i = 0; i < PHILOSOPHERS; ++i) {
      philosopher[i].join();
   }
}
```

- Um das umzusetzen, werden allen Ressourcen (das sind hier die Gabeln) eindeutige Nummern vergeben.

philo2.cpp

```
std::mutex* first; std::mutex* second;
std::string first_text; std::string second_text;
if (left_fork.id < right_fork.id) {
   first = &left_fork.mutex; second = &right_fork.mutex;
   first_text = "left"; second_text = "right";
} else {
   first = &right_fork.mutex; second = &left_fork.mutex;
   first_text = "right"; second_text = "left";
}
{
   std::lock_guard<std::mutex> lock1(*first);
   print_status("picks up the " + first_text + " fork");
   {
      std::lock_guard<std::mutex> lock2(*second);
      print_status("picks up the " + second_text + " fork");
      {
         print_status("is dining");
      }
   }
   print_status("returns the " + second_text + " fork");
}
print_status("returns the " + first_text + " fork");
print_status("leaves the table");
```

- C++ bietet zwei Klassen an, um Locks zu halten: *std*::*lock_guard* und *std*::*unique_lock*.
- Beide Varianten unterstützen die automatische Freigabe durch den jeweiligen Dekonstruktor.
- *std*::*lock_guard* ist eine reine RAII-Klasse: Die Ressource wird nur bei der Konstruktion belegt und die Freigabe erfolgt ausschließlich durch den Abbau.
- *std*::*unique_lock* bietet u.a. die Methoden *lock* und *unlock* an und einige spezielle Optionen bei der Konstruktion:

std::defer_lock	der Mutex wird noch nicht belegt
std::try_to_lock	es wird nicht-blockierend versucht, den Mutex zu belegen
std::adopt_lock	ein bereits belegter Mutex wird übernommen

- *std*::*shared_lock* kam mit C++14 und *std*::*shared_mutex* mit C++17 hinzu.

philo3.cpp

```
{
   std::unique_lock<std::mutex> lock1(left_fork, std::defer_lock);
   std::unique_lock<std::mutex> lock2(right_fork, std::defer_lock);
   std::lock(lock1, lock2);
   print_status("picks up both forks and is dining");
}
```

- C++ bietet mit *std::lock* eine Operation an, die beliebig viele Mutex-Variablen beliebigen Typs akzeptiert, und diese in einer vom System gewählten Reihenfolge belegt, die einen Deadlock vermeidet.
- Normalerweise erwartet *std::lock* Mutex-Variablen. *std::unique_lock* ist eine Verpackung, die wie eine Mutex-Variable verwendet werden kann.
- Zunächst nehmen die beiden *std::unique_lock*-Objekte die Mutex-Variablen jeweils in Beschlag, ohne eine *lock*-Operation auszuführen (*std::defer_lock*). Danach werden nicht die originalen Mutex-Variablen, sondern die *std::unique_lock*-Objekte an *std::lock* übergeben.
- Diese Variante ist umfassend auch gegen Ausnahmenbehandlungen abgesichert.

- Ein Monitor ist eine Klasse, bei der maximal ein Thread eine Methode aufrufen kann.
- Wenn weitere Threads konkurrierend versuchen, eine Methode aufzurufen, werden sie solange blockiert, bis sie alleinigen Zugriff haben (gegenseitiger Ausschluss).
- Der Begriff und die zugehörige Idee gehen auf einen Artikel von 1974 von C. A. R. Hoare zurück.
- Aber manchmal ist es sinnvoll, den Aufruf einer Methode von einer weiteren Bedingung abhängig zu machen,

- Bei Monitoren können Methoden auch mit Bedingungen versehen werden, d.h. eine Methode kommt nur dann zur Ausführung, wenn die Bedingung erfüllt ist.
- Wenn die Bedingung nicht gegeben ist, wird die Ausführung der Methode solange blockiert, bis sie erfüllt ist.
- Eine Bedingung sollte nur von dem internen Zustand eines Objekts abhängen.
- Bedingungsvariablen sind daher private Objekte eines Monitors mit den Methoden *wait*, *notify_one* und *notify_all*.
- Bei *wait* wird der aufrufende Thread solange blockiert, bis ein anderer Thread bei einer Methode des Monitors *notify_one* oder *notify_all* aufruft. (Bei *notify_all* können alle, die darauf gewartet haben, weitermachen, bei *notify_one* nur ein Thread.)
- Eine Notifizierung ohne darauf wartende Threads ist wirkungslos.

```
class Monitor {
   public:
      void some_method() {
         std::unique_lock<std::mutex> lock(mutex);
         while (! /* some condition */) {
            condition.wait(lock);
         }
         // ...
      }
      void other_method() {
         std::unique_lock<std::mutex> lock(mutex);
         // ...
         condition.notify_one();
      }
   private:
      std::mutex mutex;
      std::condition_variable condition;
};
```

- Bei der C++11-Standardbibliothek ist eine Bedingungsvariable immer mit einer Mutex-Variablen verbunden.
- *wait* gibt den Lock frei, wartet auf die Notifizierung, wartet dann erneut auf einen exklusiven Zugang und kehrt dann zurück.

- Die Methoden *notify_one* oder *notify_all* sind wirkunslos, wenn kein Thread auf die entsprechende Bedingung wartet.
- Wenn ein Thread feststellt, dass gewartet werden muss und danach wartet, dann gibt es ein Fenster zwischen der Feststellung und dem Aufruf von *wait*.
- Wenn innerhalb des Fensters *notify_one* oder *notify_all* aufgerufen wird, bleiben diese Aufrufe wirkungslos und beim anschließenden *wait* kann es zu einem Deadlock kommen, da dies auf eine Notifizierung wartet, die nun nicht mehr kommt.
- Damit das Fenster völlig geschlossen wird, muss *wait* als atomare Operation zuerst den Thread in die Warteschlange einreihen und erst dann den Lock freigeben.
- Bei *std::condition_variable* muss der Lock des Typs *std*::*unique_lock*<*std*::*mutex*> sein. Für andere Locks gibt es die u.U. weniger effiziente Alternative *std::condition_variable_any*.

$$
\begin{aligned}
& M \parallel (P_1 \mid\mid\mid P_2 \mid\mid\mid CL) \parallel C \\
M &= (lock \rightarrow unlock \rightarrow M) \\
P_1 &= (lock \rightarrow wait \rightarrow resume \rightarrow \\
& \quad critical\_region_1 \rightarrow unlock \rightarrow P_1) \\
P_2 &= (lock \rightarrow critical\_region_2 \rightarrow \\
& \quad (notify \rightarrow unlock \rightarrow P_2 \mid unlock \rightarrow P_2)) \\
CL &= (unlock_C \rightarrow unlock \rightarrow unlocked_C \rightarrow \\
& \quad lock_C \rightarrow lock \rightarrow locked_C \rightarrow CL) \\
C &= (wait \rightarrow unlock_C \rightarrow unlocked_C \rightarrow notify \rightarrow \\
& \quad lock_C \rightarrow locked_C \rightarrow resume \rightarrow C)
\end{aligned}
$$

- Einfacher Fall mit M für die Mutex-Variable, einem Prozess P_1, der auf eine Bedingungsvariable wartet, einem Prozess P_2, der notifiziert oder es auch sein lässt, und der Bedingungsvariablen C, die hilfsweise CL benötigt, um gemeinsam mit P_1 und P_2 um die Mutexvariable konkurrieren zu können.

- Wenn notwendig, können auch eigene Klassen für Locks definiert werden.
- Die Template-Klasse *std::lock_guard* akzeptiert eine beliebige Lock-Klasse, die mindestens folgende Methoden unterstützt:

  void *lock*() blockiere, bis der Lock reserviert ist
  void *unlock*() gib den Lock wieder frei

- Typhierarchien und virtuelle Methoden werden hierfür nicht benötigt, da hier statischer Polymorphismus vorliegt, bei dem mit Hilfe von Templates alles zur Übersetzzeit erzeugt und festgelegt wird.
- In einigen Fällen (wie etwa die Übergabe an *std::lock*) wird auch noch folgende Methode benötigt:

  bool *try_lock*() versuche, nicht-blockierend den Lock zu reservieren

Locks für Ressourcen mit einer Kapazität

resource-lock.hpp

```
class ResourceLock {
   public:
      ResourceLock(unsigned int capacity) : capacity(capacity), used(0) {
      }
      void lock() {
         std::unique_lock<std::mutex> lock(mutex);
         while (used == capacity) {
            released.wait(lock);
         }
         ++used;
      }
      void unlock() {
         std::unique_lock<std::mutex> lock(mutex);
         assert(used > 0);
         --used;
         released.notify_one();
      }
   private:
      const unsigned int capacity;
      unsigned int used;
      std::mutex mutex;
      std::condition_variable released;
};
```

philo4.cpp

```
constexpr unsigned int PHILOSOPHERS = 5;
ResourceLock rlock(PHILOSOPHERS-1);
std::thread philosopher[PHILOSOPHERS];
std::mutex fork[PHILOSOPHERS];
for (int i = 0; i < PHILOSOPHERS; ++i) {
   philosopher[i] = std::thread(Philosopher(i+1,
      fork[i], fork[(i + PHILOSOPHERS - 1) % PHILOSOPHERS], rlock));
}
```

- Mit Hilfe eines *ResourceLock* lässt sich das Philosophenproblem mit Hilfe eines Dieners lösen.
- Bei n Philosophen lassen die Diener zu, dass sich $n-1$ Philosophen hinsetzen.

Philosophen mit Diener

philo4.cpp

```
void operator()() {
   for (int i = 0; i < 5; ++i) {
      print_status("comes to the table");
      {
         std::lock_guard<ResourceLock> lock(rlock);
         print_status("got permission to sit down at the table");
         {
            std::lock_guard<std::mutex> lock1(left_fork);
            print_status("picks up the left fork");
            {
               std::lock_guard<std::mutex> lock2(right_fork);
               print_status("picks up the right fork");
               {
                  print_status("is dining");
               }
            }
            print_status("returns the right fork");
         }
         print_status("returns the left fork");
      }
      print_status("leaves the table");
   }
}
```

thread-overhead.cpp

```
#include <chrono>
#include <iostream>
#include <ratio>
#include <thread>

int main() {
   unsigned int counter = 0;
   unsigned int nof_threads = 1<<15;

   auto start = std::chrono::high_resolution_clock::now();
      for (unsigned int i = 0; i < nof_threads; ++i) {
         auto t = std::thread([&]() { ++counter; });
         t.join();
      }
   auto finish = std::chrono::high_resolution_clock::now();
   std::chrono::duration<double, std::micro> elapsed = finish - start;
   std::cout << "avg time per thread creation = " <<
      elapsed.count() / nof_threads << " us" << std::endl;
}
```

```
theon$ g++ -std=c++11 -O3 -o thread-overhead thread-overhead.cpp
theon$ thread-overhead
avg time per thread creation = 30.3621 us
theon$
```

- Das Erzeugen und Abbauen von Threads ist nicht sehr billig. Auf Theon kostet dies ca. 30 μs pro Thread.
- Es stellt sich daher die Frage, ob dies vermieden werden kann.
- Über *std*::*thread*::*hardware_concurrency*() lässt sich ermitteln, wieviele Threads auf der aktuellen Hardware gleichzeitig operieren können.
- Entsprechend erscheint es naheliegend, entsprechend viele Threads zu Beginn anzulegen und diese dann mit den anfallenden Aufgaben zu beschäftigen.
- Das Master/Worker-Pattern ist hier hilfreich.

$$
\begin{aligned}
P &= M \,||\, (W \,|||\, \ldots \,|||\, W) \\
M &= (job \rightarrow \cdots \rightarrow job \rightarrow SKIP) \\
W &= (job \rightarrow W) \,[]\, SKIP
\end{aligned}
$$

- Das Master/Worker-Pattern operiert mit mindestens einem Master-Thread und beliebig vielen Worker-Threads, die anfallende Jobs nacheinander abarbeiten.

Das Master/Worker-Pattern operiert mit mindestens einem Master-Thread, beliebig vielen Worker-Threads und einer Warteschlange für anfallende Aufgaben, auf die alle Parteien konkurrierend zugreifen:

- Die anfallenden Aufgaben werden analog wie beim Fork-And-Join-Pattern in Teilaufgaben zerlegt und durch den Master-Thread an das Ende der Warteschlange angefügt.
- Jeder Worker-Thread holt in einer immerwährenden Schleife immer eine Aufgabe aus der Warteschlange und arbeitet diese ab. Wenn die Warteschlange leer ist, wird darauf gewartet, dass die Warteschlange sich wieder füllt.
- Die Warteschlange muss konkurrierende Zugriffe durch den Master und die Worker zulassen und das Warten auf eine sich wieder füllende Warteschlange zulassen.

Aufgaben können wie bei Threads durch Funktionsobjekte repräsentiert werden:

- Da Funktionsobjekte alle möglichen Typen haben können bzw. bei Lambda-Ausdrücken diese ohnehin anonym sind, lohnt sich die Verwendung der Template-Klasse *std*::*function*, die beliebige Funktionsobjekte mit einer einheitlichen Parameterliste und einem vorgegebenen Return-Typ verwalten kann.
- Beispiel: *std*::*function*<**double**(**int**, **float**)> repräsentiert beliebige Funktionsobjekte, mit zwei Parametern (**int** und **float**) und dem Return-Typ **double**.
- Zu Beginn versuchen wir es ganz einfach ohne Parameter und mit dem Return-Typ **void**: *std*::*function*<**void**()>.
- Beispiel:

$$
\begin{aligned}
&\mathit{std}::\mathit{function}<\textbf{void}()>\ \mathit{task} = [=,\&\mathit{result}]()\ \{\\
&\quad \mathit{result}[i] = \mathit{simpson}(f,\ a,\ b,\ n);\\
&\};
\end{aligned}
$$

tpool1-simpson.cpp

```
struct ThreadPool {
   public:
      using Job = std::function<void()>;
      // ...
   private:
      unsigned int nof_threads;
      bool finished;
      std::vector<std::thread> threads;
      std::mutex mutex;
      std::condition_variable cv;
      std::deque<Job> jobs;
      // ...
};
```

- Ein Thread-Pool verknüpft eine Warteschlange für Aufgaben mit Zugriffsmethoden mit einem Array von Threads für die Worker. Die **bool**-Variable *finished* dient dem Abbau des Thread-Pools.

tpool1-simpson.cpp

```
ThreadPool(unsigned int nof_threads) :
      nof_threads(nof_threads), finished(false),
      threads(nof_threads) {
   for (auto& t: threads) {
      t = std::thread([=]() { process_jobs(); });
   }
}
```

- Die private Methode *process_jobs* repräsentiert den Algorithmus eines Workers. Die Capture [=] umfasst hier **this**, das implizit benötigt wird, um die Methode aufzurufen, da *process_jobs*(); zu **this**−>*process_jobs*(); expandiert wird.

tpool1-simpson.cpp

```
~ThreadPool() {
   {
      std::unique_lock<std::mutex> lock(mutex);
      finished = true;
   }
   cv.notify_all();
   for (auto& t: threads) {
      t.join();
   }
}
```

- Beim Abbau wird *finished* auf **true** gesetzt und danach werden alle Worker benachrichtigt, dass die Arbeit einzustellen ist, sobald alle Aufträge abgearbeitet sind. Mit *join* werden dann alle Threads ordnungsgemäß beendet.
- Der Abbau eines Thread-Pools bietet somit auch immer eine Möglichkeit der Synchronisierung.

tpool1-simpson.cpp

```
void submit(Job job) {
   std::unique_lock<std::mutex> lock(mutex);
   jobs.emplace_back(std::move(job));
   cv.notify_one();
}
```

- Die Methode *submit* nimmt einen Auftrag für die Worker entgegen und fügt an das Ende der Warteschlange an. Mit *std::move* wird ein unnötiges Klonen des Funktionsobjekts vermieden.
- Wenn Worker auf neue Aufträge warten, wird durch *cv.notify_one*() einer davon aufgeweckt.

tpool1-simpson.cpp

```
void process_jobs() {
   for(;;) {
      Job job;
      /* fetch job */
      {
         std::unique_lock<std::mutex> lock(mutex);
         while (jobs.empty() && !finished) {
            cv.wait(lock);
         }
         if (jobs.empty()) break;
         job = std::move(jobs.front());
         jobs.pop_front();
      }
      /* execute job */
      job();
   }
}
```

- Worker arbeiten kontinuierlich Aufträge aus der Warteschlange ab. Wenn die Warteschlange leer ist, wird gewartet. Wenn zudem *finished* gesetzt ist, beendet der Worker seine Tätigkeit.

Einfache Anwendung des Thread-Pools

tpool1-simpson.cpp

```
int main() {
   unsigned int nofintervals = 1<<12;
   unsigned int jobsize = 1<<6;
   unsigned int noftasks = nofintervals / jobsize;
   std::vector<double> results(noftasks);

   {
      ThreadPool tpool(std::thread::hardware_concurrency());

      auto f = [](double x) { return 4 / (1 + x*x); };
      double a = 0; double b = 1;

      // submit simpson for each interval
      for (int i = 0; i < noftasks; ++i) {
         tpool.submit([=,&results]() {
            double ai = a + i * (b - a) / noftasks;
            double bi = a + (i + 1) * (b - a) / noftasks;
            results[i] = simpson(f, ai, bi, jobsize);
         });
      }
   }

   // sum up the results
   double sum = 0;
   for (auto res: results) {
      sum += res;
   }
   std::cout << std::setprecision(14) << sum << std::endl;
   std::cout << std::setprecision(14) << M_PI << std::endl;
}
```

- In dem vorgestellten Beispiel war eine einfache Synchronisierung nur mit dem gesamten Thread-Pool möglich.
- Es gab keine Vorkehrung zur Synchronisierung mit der Fertigstellung eines Auftrags.
- Prinzipiell ist eine Synchronisierung immer möglich mit Hilfe von Bedingungsvariablen.
- Die C++-Standardbibliothek offeriert hier eine fertige Lösung mit *std*::*promise* und *std*::*future*.

- C++ bietet mit *std::promise* und *std::future* eine einfache Kommunikations- und Synchronisierungsmöglichkeit an.
- Mit *std*::*promise*<*T*> *p*; wird ein „Versprechen“ gegeben, irgendwann einmal mit *p.set_value*(*value*) einen Wert zu setzen.
- Mit *std*::*future*<*T*> *f* = *p.get_future*() darf daraus genau einmal(!) ein zugehöriges *std::future*-Objekt abgeleitet werden.
- Typischerweise „gehören“ dann die beiden Objekte unterschiedlichen Threads.
- Mit *f.get*() wartet der Aufrufer darauf, dass der Wert mit *p.set_value*(*value*) gesetzt wird und liefert dann diesen zurück.
- Normale Zuweisungen sind weder für *std::promise*- noch *std::future*-Objekte zulässig. Ein Paar dient dazu, einen Wert genau einmal synchronisert weiterzugeben.

simpson-fp.cpp

```
template<typename T, typename F>
T mt_simpson(F&& f, T a, T b, unsigned int n, unsigned int nofthreads) {
   // divide the given interval into nofthreads partitions
   assert(n > 0 && a <= b && nofthreads > 0);
   unsigned int nofintervals = n / nofthreads;
   unsigned int remainder = n % nofthreads;
   unsigned int interval = 0;
   std::future<T> results[nofthreads];
   // ... fork ...
   // join threads and sum up their results
   T sum = 0;
   for (unsigned int i = 0; i < nofthreads; ++i) {
      sum += results[i].get();
   }
   return sum;
}
```

- Das Array der hier deklarierten *std::future*-Objekte ist zu Beginn noch „leer“, d.h. die Objekte sind noch nicht mit *std::promise*-Objekten verbunden.
- Am Ende werden die Ergebnisse eingesammelt. Die Synchronisierung erfolgt hier implizit bei der *get*-Methode.

simpson-fp.cpp

```
// fork off the individual threads
T x = a;
for (unsigned int i = 0; i < nofthreads; ++i) {
   unsigned int intervals = nofintervals;
   if (i < remainder) ++intervals;
   interval += intervals;
   T xleft = x; x = a + interval * (b - a) / n;
   std::promise<T> promise;
   results[i] = promise.get_future();
   auto t = std::thread([=, promise = std::move(promise)]() mutable {
      promise.set_value_at_thread_exit(simpson(f, xleft, x,
         intervals));
   });
   t.detach(); // no longer joinable, t object can be destructed
}
```

- In der Schleife werden lokal temporäre *std::promise*-Objekte erzeugt und daraus jeweils das zugehörige *std::future*-Objekt abgeholt.
- Das *std::future*-Objekt wird in *results* abgespeichert, das *std::promise*-Objekt geht an den zu erzeugenden Thread.

simpson-fp.cpp

```
std::promise<T> promise;
results[i] = promise.get_future();
auto t = std::thread([=, promise = std::move(promise)]() mutable {
   promise.set_value_at_thread_exit(simpson(f, xleft, x,
      intervals));
});
t.detach(); // no longer joinable, t object can be destructed
```

- Hier wird zunächst ein *std::promise*-Objekt lokal erzeugt und von diesem ein *std::future*-Objekt abgeleitet, das in dem *results*-Array abgespeichert wird.
- Dann wird ein Thread gestartet, bei dem als Funktions-Objekt ein Lambda-Ausdruck übergeben wird.
- Der Lambda-Ausdruck übernimmt diverse Parameter implizit durch Zuweisung und im Falle von *promise* durch *std::move*, da eine normale Zuweisung nicht zulässig ist. (Dies geht so erst ab C++14, bei C++11 wird noch *std::bind* benötigt.)
- Ohne **mutable** kann der Lambda-Ausdruck *promise* nicht verändern.

simpson-fp.cpp

```
t.detach(); // no longer joinable, t object can be destructed
```

- Auf einen Thread muss entweder mit *join* irgendwann gewartet werden oder dieser muss explizit mit *detach* „vergessen“ werden. Nach *detach* ist eine Synchronisierung mit *join* nicht mehr möglich. Das ist hier auch nicht notwendig, da die Synchronisierung über die *std::future*-Objekte erfolgt.
- Allerdings sollte dann bei *std*::*promise* die Methode *set_value_at_thread_exit* verwendet werden, damit der vollständige Abbau des Threads abgewartet wird.
- Wenn ein Thread-Objekt dekonstruiert wird, für den weder *join* noch *detach* aufgerufen wurde, dann wird die gesamte Programmausführung gewaltsam mit *std::terminate* beendet.

simpson-async.cpp

```
results[i] = std::async([=]() -> T {
   return simpson(f, xleft, x, intervals);
});
```

- Den Standard-Fall, dass ein Thread genau einen Wert berechnet, der über ein *std::future*-Objekt synchronisiert abgeholt werden kann, wird mit *std::async* vereinfacht.
- *std::async* benötigt ein Funktionsobjekt (hier ein Lambda-Ausdruck), das den gewünschten Wert mit **return** zurückliefert.
- *std::async* erzeugt ein *std::promise*-Objekt, leitet davon das *std::future*-Objekt ab, das zurückgegeben wird, ruft dann in einem neu erzeugten Thread das Funktionsobjekt auf und weist den zurückgegebenen Wert dem *std::promise*-Objekt zu.

Funktionsobjekte lassen sich mit *std::future* und *std::promise* verknüpfen:

- Die C++-Standardbibliothek bietet hierfür *std*::*packaged_task* an, das genauso wie *std*::*function* parametrisiert wird. Beispiel: *std*::*packaged_task*<*T*()>
- *std*::*packaged_task* bietet die Methode *get_future* an, mit der ein entsprechendes *std*::*future*-Objekt geliefert wird.
- *std*::*packaged_task*-Objekte sind wiederum Funktionsobjekte mit Return-Typ **void**. Der Return-Wert des ursprünglichen Funktionsobjekts wird somit implizit dem *std*::*promise*-Objekt zugewiesen.
- Achtung: *std*::*package_task*-Objekte sind nicht kopierbar, nur *std*::*move* wird unterstützt.

thread_pool.hpp

```
class thread_pool {
   public:
      /* ... */

   private:
      std::mutex mutex;
      std::condition_variable cv;
      std::condition_variable joining_finished; // joined==true?
      std::vector<std::thread> threads; // fixed number of threads
      std::deque<std::function<void()>> tasks; // submitted tasks
      unsigned int active; // # of threads that are executing a task
      bool joining; // initially false, set to true if join() is invoked
      bool joined; // initially false, set to true if join is completed
      bool terminating; // initially false, set to true by terminate()
};
```

- Diese Lösung arbeitet mit *std::future*-Objekten und unterstützt Methoden wie *join* (Synchronisierung mit dem Ende des Thread-Pools) und *terminate* (erzwungener Abbau des Thread-Pools).
- Quellen: `https://github.com/afborchert/tpool`

thread_pool.hpp

```
thread_pool(unsigned int nofthreads) :
      threads(nofthreads), active(0),
      joining(false), joined(false), terminating(false) {
   for (auto& t: threads) {
      t = std::thread([=]() mutable -> void {
         for (;;) {
            std::function<void()> task;
            /* fetch next task, if there is any */
            {
               std::unique_lock<std::mutex> lock(mutex);
               while (!terminating &&
                     tasks.empty() &&
                     (active > 0 || !joining)) {
                  cv.wait(lock);
               }
               if (tasks.empty()) {
                  break; /* all done */
               }
               task = std::move(tasks.front());
               tasks.pop_front(); ++active;
            }
            /* process task */
            task();
            {
               std::unique_lock<std::mutex> lock(mutex); --active;
            }
         }
         cv.notify_all();
      });
   }
}
```

thread_pool.hpp

```
        {
           std::unique_lock<std::mutex> lock(mutex);
           while (!terminating &&
                  tasks.empty() &&
                  (active > 0 || !joining)) {
              cv.wait(lock);
           }
           if (tasks.empty()) {
              break; /* all done */
           }
           task = std::move(tasks.front());
           tasks.pop_front(); ++active;
        }
```

- Wenn wir terminieren, hören wir sofort auf.
- Wenn die Queue leer ist, warten wir nur, wenn noch nicht *join* aufgerufen und/oder noch andere Threads aktiv sind.
- Wir müssen hier beachten, dass noch aktive Threads selbst neue Jobs mit *submit* hinzufügen können – zum Beispiel im Teile-und-Herrsche-Pattern.

thread_pool.hpp

```
template<typename F, typename... Parameters>
auto submit(F&& task_function, Parameters&&... parameters)
      -> std::future<decltype(task_function(parameters...))> {
   using T = decltype(task_function(parameters...));
   auto task = std::make_shared<std::packaged_task<T()>>(
      std::bind(std::forward<F>(task_function),
         std::forward<Parameters>(parameters)...)
   );
   std::future<T> result = task->get_future();
   std::lock_guard<std::mutex> lock(mutex);
   if (!terminating && !joined) {
      tasks.emplace_back([task]() { (*task)(); });
      cv.notify_one();
   }
   return result;
}
```

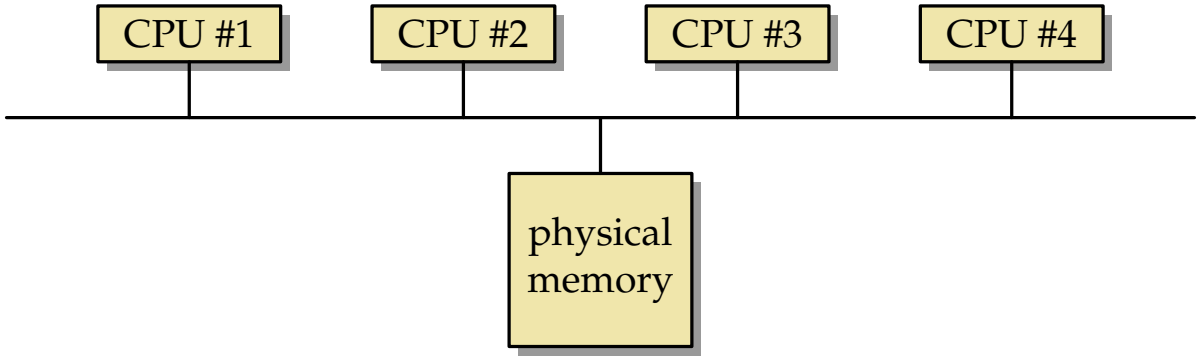
thread_pool.hpp

```
template<typename F, typename... Parameters>
auto submit(F&& task_function, Parameters&&... parameters)
      -> std::future<decltype(task_function(parameters...))> {
   using T = decltype(task_function(parameters...));
   auto task = std::make_shared<std::packaged_task<T()>>(
      std::bind(std::forward<F>(task_function),
         std::forward<Parameters>(parameters)...)
   );
   std::future<T> result = task->get_future();
   std::lock_guard<std::mutex> lock(mutex);
   if (!terminating && !joined) {
      tasks.emplace_back([task]() { (*task)(); });
      cv.notify_one();
   }
   return result;
}
```

thread_pool.hpp

```
template<typename F, typename... Parameters>
auto submit(F&& task_function, Parameters&&... parameters)
      -> std::future<decltype(task_function(parameters...))> {
   using T = decltype(task_function(parameters...));
   auto task = std::make_shared<std::packaged_task<T()>>(
      std::bind(std::forward<F>(task_function),
         std::forward<Parameters>(parameters)...)
   );
   std::future<T> result = task->get_future();
   /* ... */
}
```

- *std*::*function*-Objekte sind kopierbar und akzeptieren nur Funktionsobjekte, die wiederum kopierbar sind.
- *std*::*packaged_task*-Objekte sind nur *move-constructible* und somit nicht kopierbar, da sie ein *std*::*promise*-Objekt enthalten.
- Deswegen muss das *std*::*packaged_task*-Objekte hinter einen *std*::*shared_ptr*-Zeiger verpackt werden.

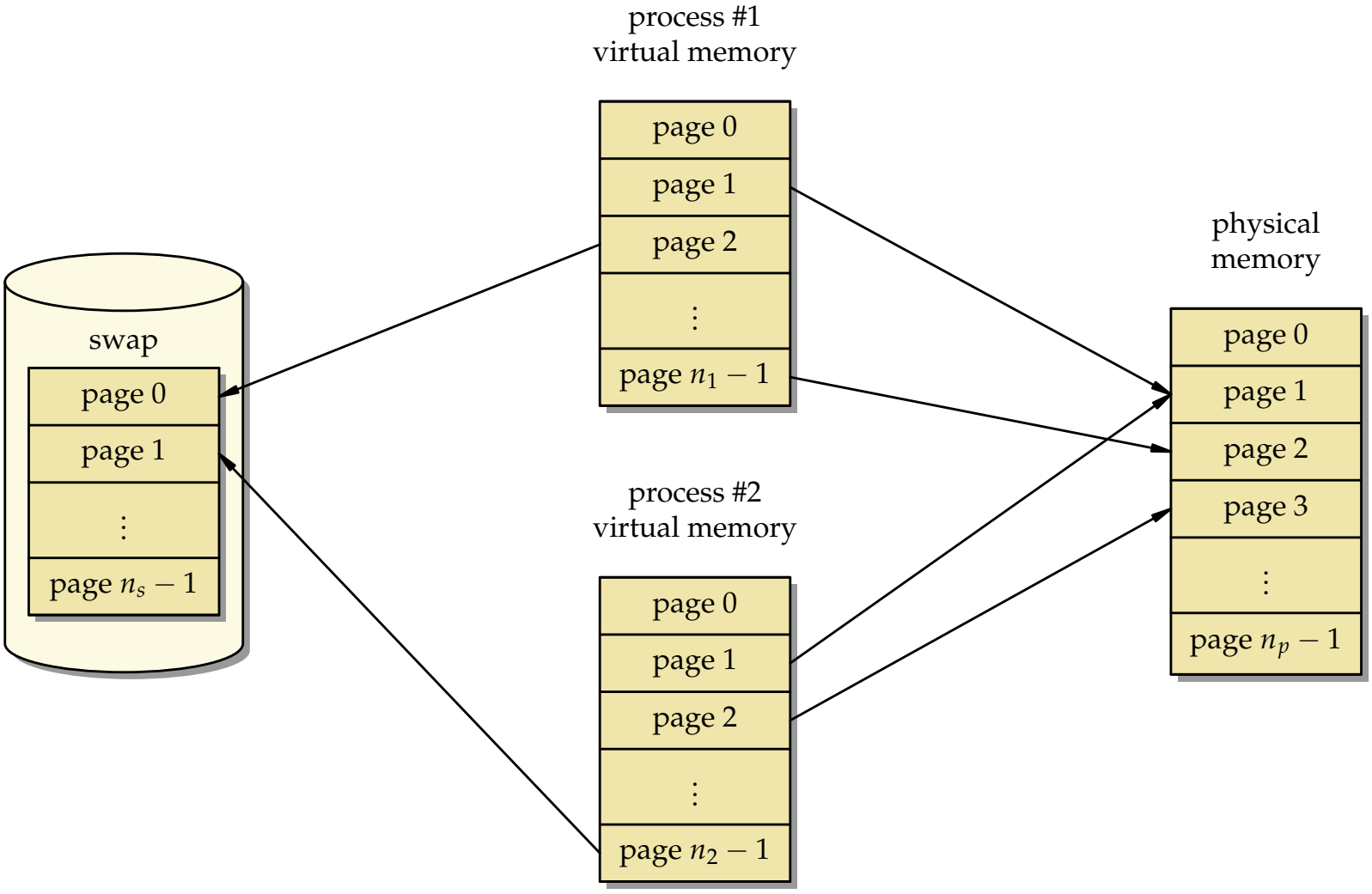


- Sofern gemeinsamer Speicher zur Verfügung steht, so bietet dies die effizienteste Kommunikationsbasis parallelisierter Programme.
- Am häufigsten anzutreffen ist die UMA-Variante (*uniform memory access*, siehe Abbildung). Diese Architektur finden wir auf unseren Rechnern vor.
- Die Synchronisation muss jedoch auf anderem Wege erfolgen. Dies kann entweder mittels der entsprechenden Operationen für Threads (etwa mit *join* und auf Basis von Bedingungsvariablen), über lokale Netzwerkkommunikation oder anderen Synchronisierungsoperationen des Betriebssystems erfolgen.

Aus der Sicht unserer Programme ist Speicher eine virtuelle Ressource, auf die über eine Cache-Hierarchie zugegriffen wird:

- Die Hardware-Caches L1, L2 und typischerweise L3: Die CPU greift auf L1 zu, L1 ggf. auf L2 usw.
- Der physisch zur Verfügung stehende Speicher wird als Cache für den virtuellen Speicher betrachtet. Es ist also möglich, dass Inhalte des virtuellen Speichers zeitweilig nur auf einer Platte (*swap space*) liegen.
- Die Zugriffsgeschwindigkeiten auf den einzelnen Ebenen sind extrem unterschiedlich.

- Der Adressraum eines Prozesses ist eine virtuelle Speicherumgebung, die von dem Betriebssystem mit Unterstützung der jeweiligen Prozessorarchitektur (MMU = *memory management unit*) umgesetzt wird.
- Die virtuellen Adressen eines Prozesses werden dabei in physische Adressen des Hauptspeichers konvertiert.
- Für diese Abbildung wird der Speicher in sogenannte Kacheln (*pages*) eingeteilt.
- Die Größe einer Kachel ist systemabhängig. Auf der Theseus sind es 8 KiB, auf Theon und Livingstone 4 KiB (abzurufen über den Systemaufruf *getpagesize*()).
- Wenn nicht genügend physischer Hauptspeicher zur Verfügung steht, können auch einzelne Kacheln auf eine Platte ausgelagert werden (*swap space*), was zu erheblichen Zeitverzögerungen bei einem nachfolgendem Zugriff führt.


process #1
virtual memory
page 0
page 1
page 2
⋮
page $n_1 - 1$
physical
memory
page 0
page 1
page 2
page 3
⋮
page $n_p - 1$
swap
page 0
page 1
⋮
page $n_s - 1$
process #2
virtual memory
page 0
page 1
page 2
⋮
page $n_2 - 1$

- Jeder Prozess hat unter UNIX einen eigenen Adressraum.
- Mehrere Prozesse können gemeinsame Speicherbereiche haben (nicht notwendigerweise an den gleichen Adressen). Die Einrichtung solcher gemeinsamer Bereiche ist möglich mit den Systemaufrufen *mmap* (*map memory*) oder *shm_open* (*open shared memory object*).
- Jeder Prozess hat zu Beginn einen Thread und kann danach (mit *pthread_create* bzw. *std::thread*) weitere Threads erzeugen.
- Alle Threads eines Prozesses haben einen gemeinsamen virtuellen Adressraum. Gelegentlich wird bei Prozessen von Rechtegemeinschaften gesprochen, da alle Threads die gleichen Zugriffsmöglichkeiten und -rechte haben.

Zwei Aspekte in Bezug auf Korrektheit und Performance sind besonders relevant:

- Wie ist das Verhalten des gemeinsamen Speichers, wenn mehrere Threads konkurrierend und unsychronisiert (d.h. insbesondere ohne Mutex-Variablen) auf die gleichen Datenbereiche zugreifen?
- Welche Auswirkungen hat das Zugriffsverhalten der Threads auf die Performance?

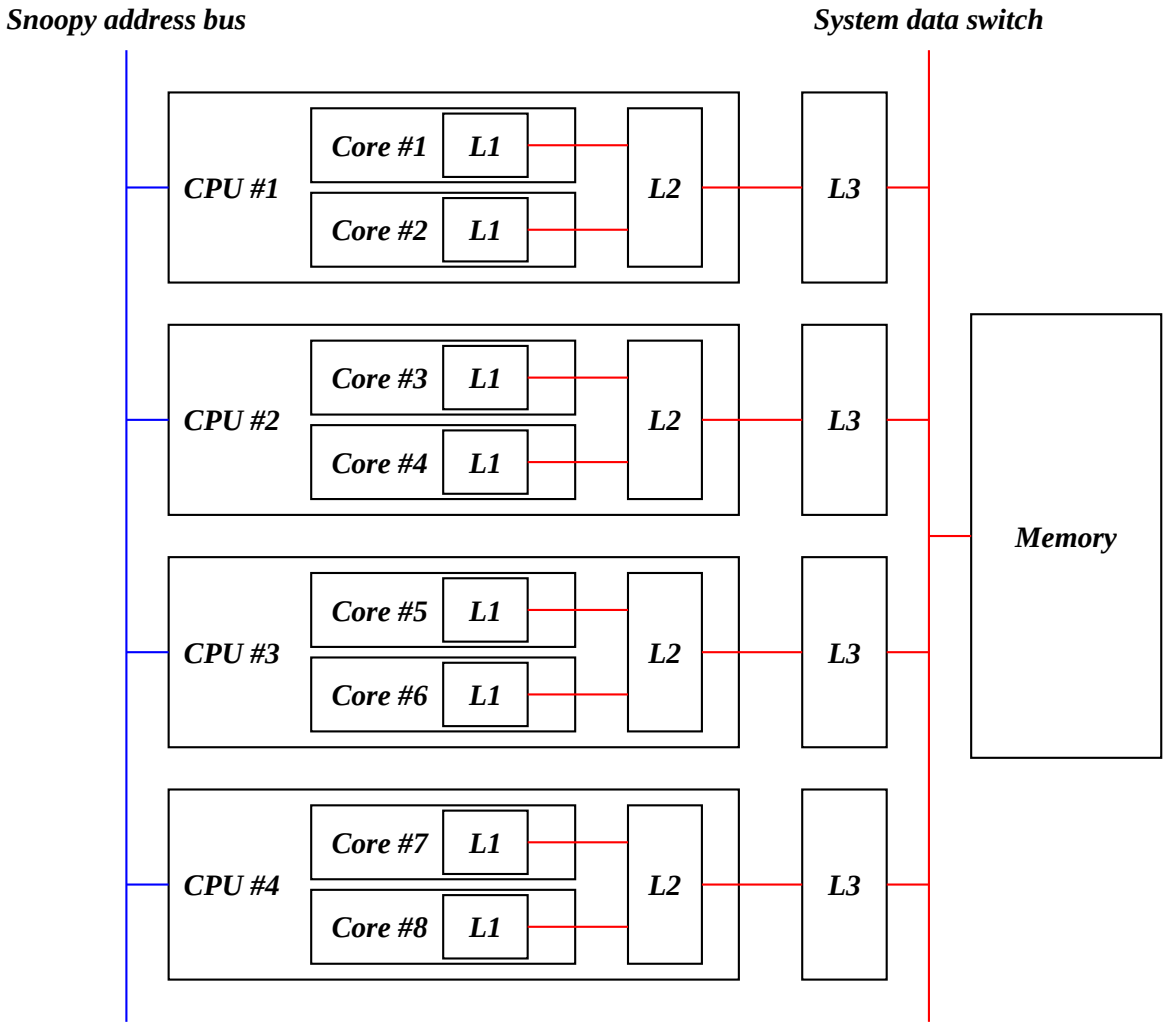
Die Relevanz ergibt sich daraus, dass

- gemeinsamer Speicher sich nicht mehr „intuitiv“ verhält bzw. ein erzwungenes „intuitives“ Verhalten so restriktiv wäre, dass ein erhebliches Optimierungspotential verloren gehen würde, und dass
- ohne sorgfältige Planung und Koordinierung von Speicherzugriffen in Abhängigkeit der gegebenen Architektur die Laufzeit ein Vielfaches im Vergleich mit einer optimierten Konfiguration betragen kann.

- Zugriffe einer CPU auf den primären Hauptspeicher sind vergleichsweise langsam. Obwohl Hauptspeicher generell schneller wurde, behielten die CPUs ihren Geschwindigkeitsvorsprung.
- Grundsätzlich ist Speicher direkt auf einer CPU deutlich schneller. Jedoch lässt sich Speicher auf einem CPU-Chip aus Komplexitäts-, Produktions- und Kostengründen nicht beliebig ausbauen.
- Deswegen arbeiten moderne Architekturen mit einer Kette hintereinander geschalteter Speicher. Zur Einschätzung der Größenordnung sind hier die Angaben für die Theseus, die mit Prozessoren des Typs UltraSPARC IV+ ausgestattet ist:

Cache	Kapazität	Taktzyklen
Register		1
L1-Cache	64 KiB	2-3
L2-Cache	2 MiB	um 10
L3-Cache	32 MiB	um 60
Hauptspeicher	32 GiB	um 250

Cache-Struktur der Theseus



Messungen arbeiten mit Speicherbereichen, die als Array von Zeigern organisiert sind, für die folgendes gilt:

- Alle Zeiger verweisen in das gleiche Array,
- mit einem beliebigen Zeiger in dem Array lassen sich alle anderen Zeiger erreichen und
- das gesamte Array wird abgedeckt.

Wenn ein solcher Speicherbereich konfiguriert ist, lassen sich die Speicherzugriffszeiten messen, indem die Zeigerkette n Mal verfolgt wird:

```
void** p = (void**) memory[0];
while (n-- > 0) {
    p = (void**) *p;
}
```

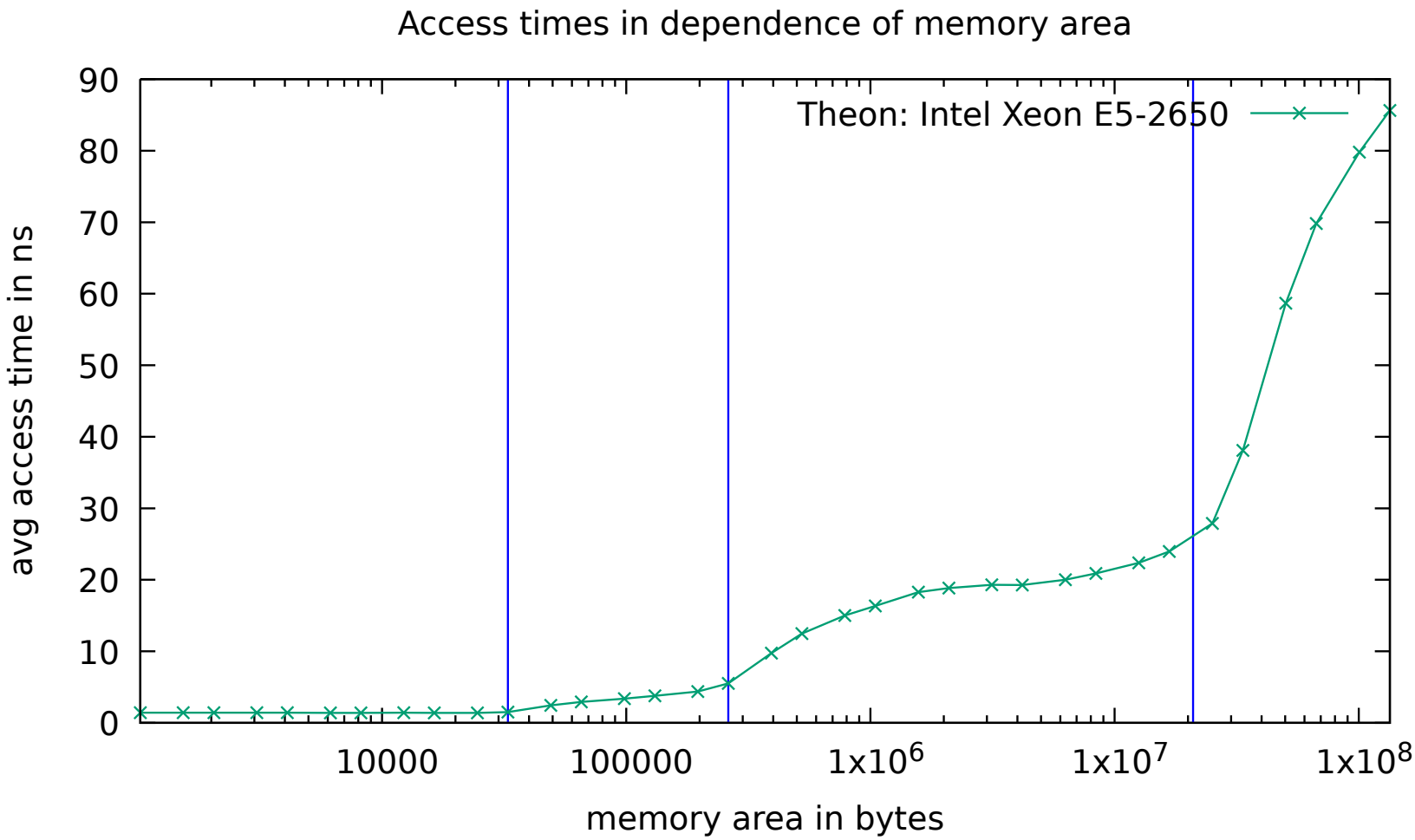
Das Konstrukt $p = (\textbf{void}{**})*p$ erzwingt die Serialisierung der Speicherzugriffe.

Möglichkeiten, die Zeigerkette in dem Speicherbereich zu organisieren:

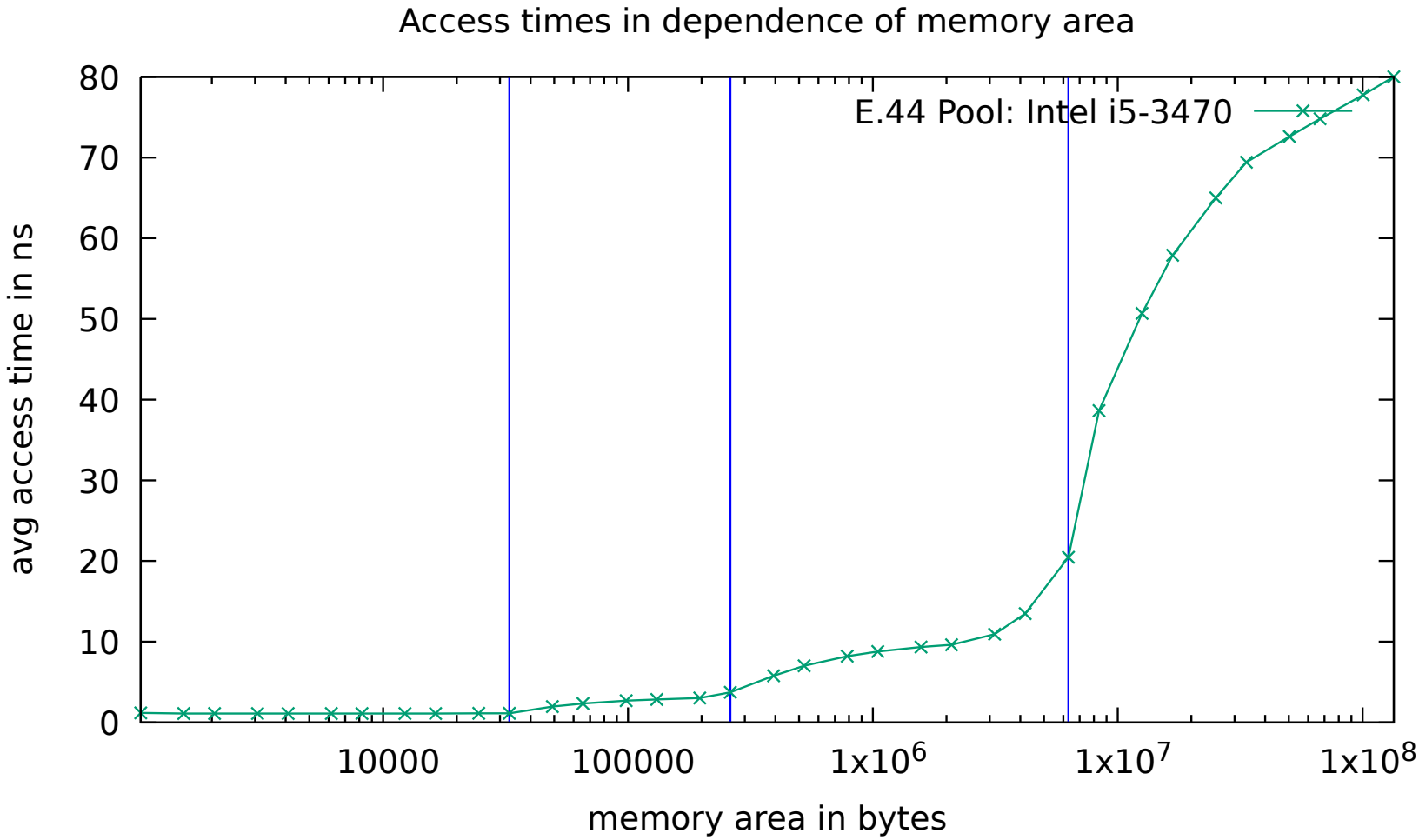
- Zufällig: Bei einer zufälligen Anordnung ist die jeweils nächste Lokation im Speicher für die Maschine nicht vorhersehbar. Entsprechend nähern wir uns damit dem *worst case*-Fall, bei dem der nächste Zeiger nicht in einem der Caches liegt und deswegen aus dem Hauptspeicher geholt werden muss.
- Linear: Die Zeigerkette verwendet weitgehend konstante Abstände. Moderne Prozessoren erkennen solche Zugriffsmuster und können dann bereits auf Verdacht aus dem Speicher in den Cache laden, bevor die entsprechende Lade-Instruktion kommt.
- Fused: Hier arbeiten wir mit n miteinander verwobenen linearen Zugriffsketten. Damit lässt sich testen, wieviele linearen Zugriffsmuster von dem Prozessor parallel verfolgt werden können.

Voraussetzung ist natürlich in jedem Fall, dass der Speicherbereich deutlich größer als die Kapazität der Caches ist.
Werkzeuge hierzu:
`https://github.com/afborchert/pointer-chasing`

Gemessene Speicherzugriffszeiten: Theon

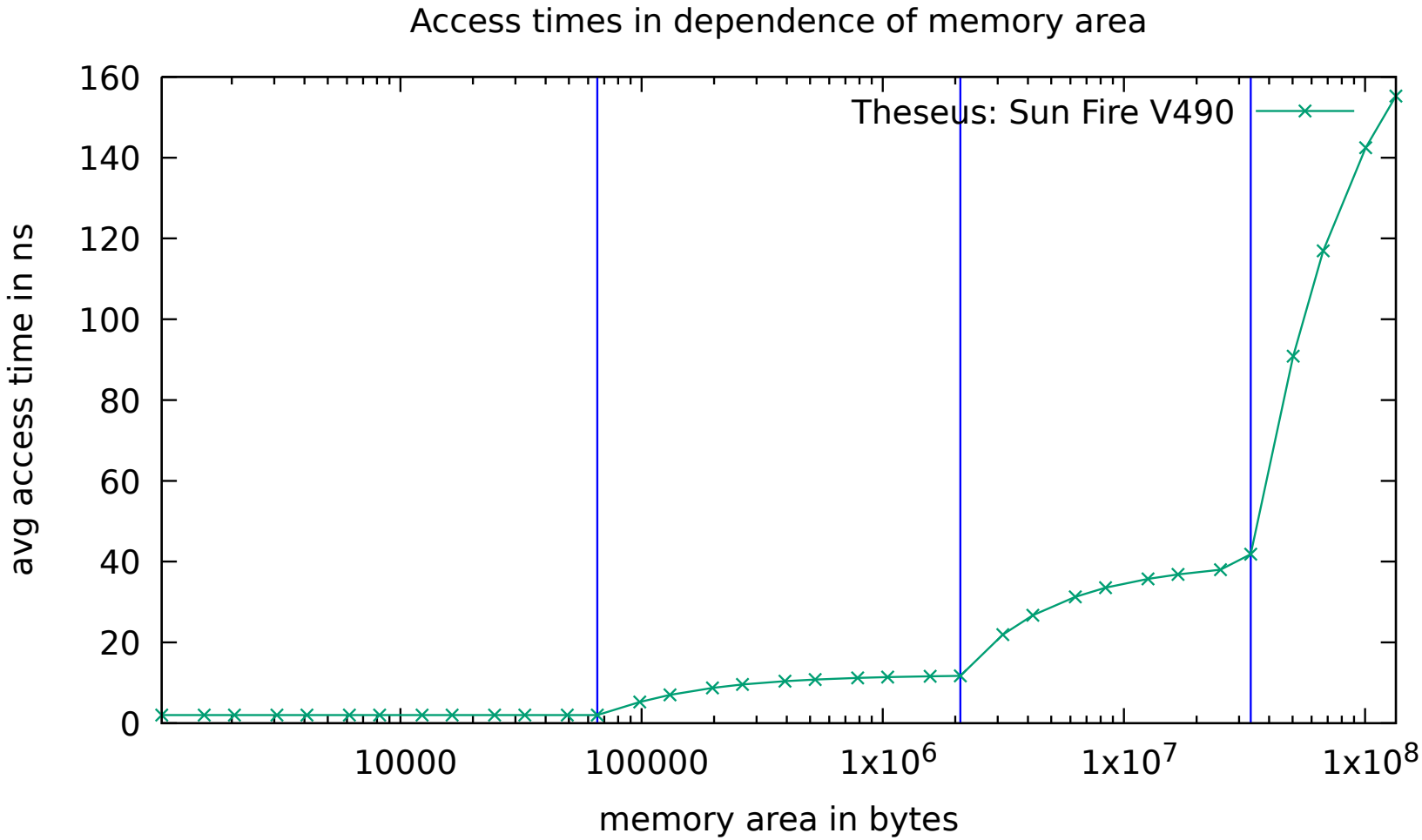


Theon: 1 Intel E5-2650-Prozessor mit 12 Kernen mit je 2 Threads
Caches: L1 (32 KiB), L2 (256 KiB), L3 (20 MiB)



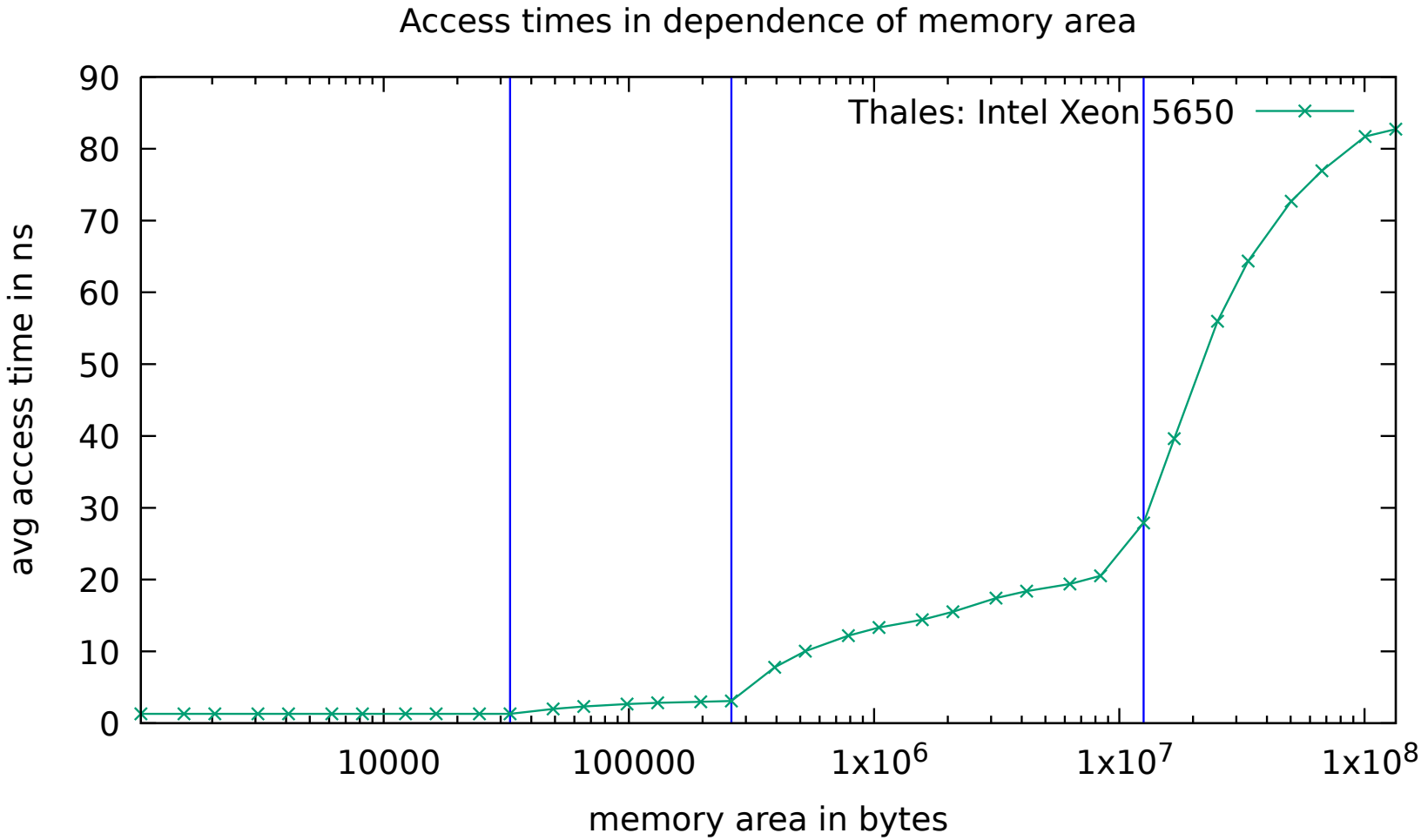
E.44: 1 Intel i5-3470-Prozessor mit 4 Kernen
Caches: L1 (32 KiB), L2 (256 KiB), L3 (6 MiB)

Gemessene Speicherzugriffszeiten: Theseus

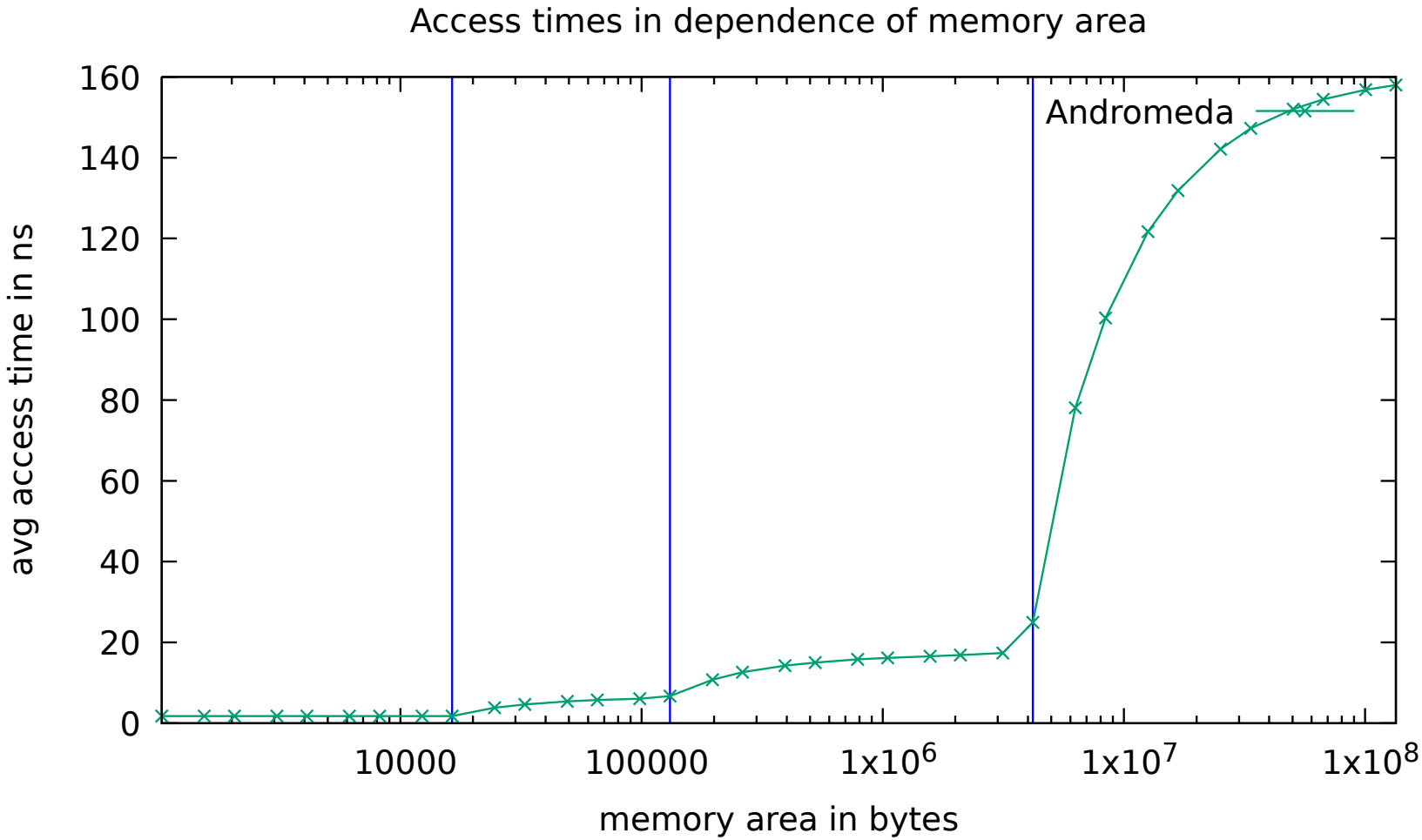


Theseus: 4 UltraSPARC IV+ Prozessoren mit je zwei Kernen
Caches: L1 (64 KiB), L2 (2 MiB), L3 (32 MiB)

Gemessene Speicherzugriffszeiten: Thales

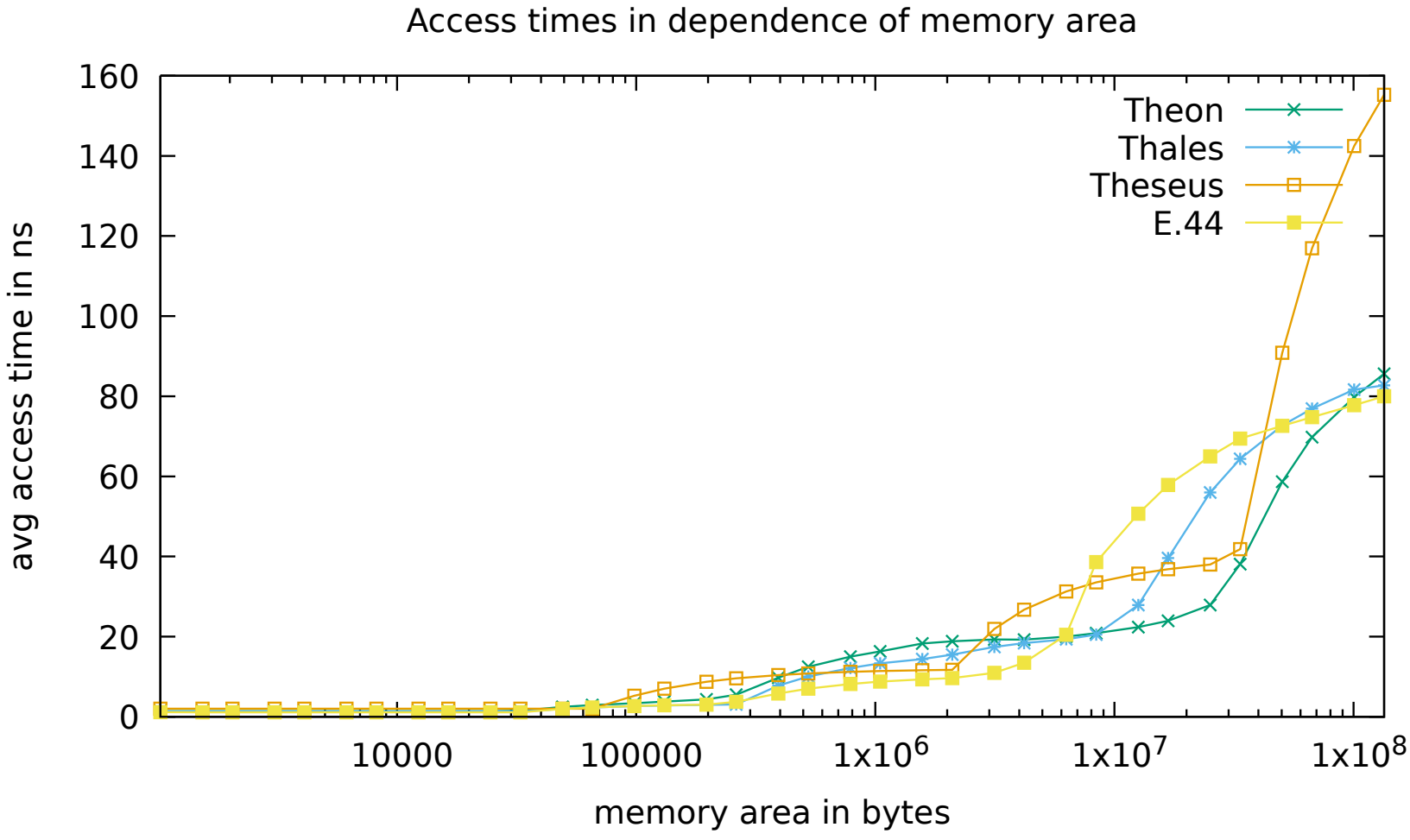


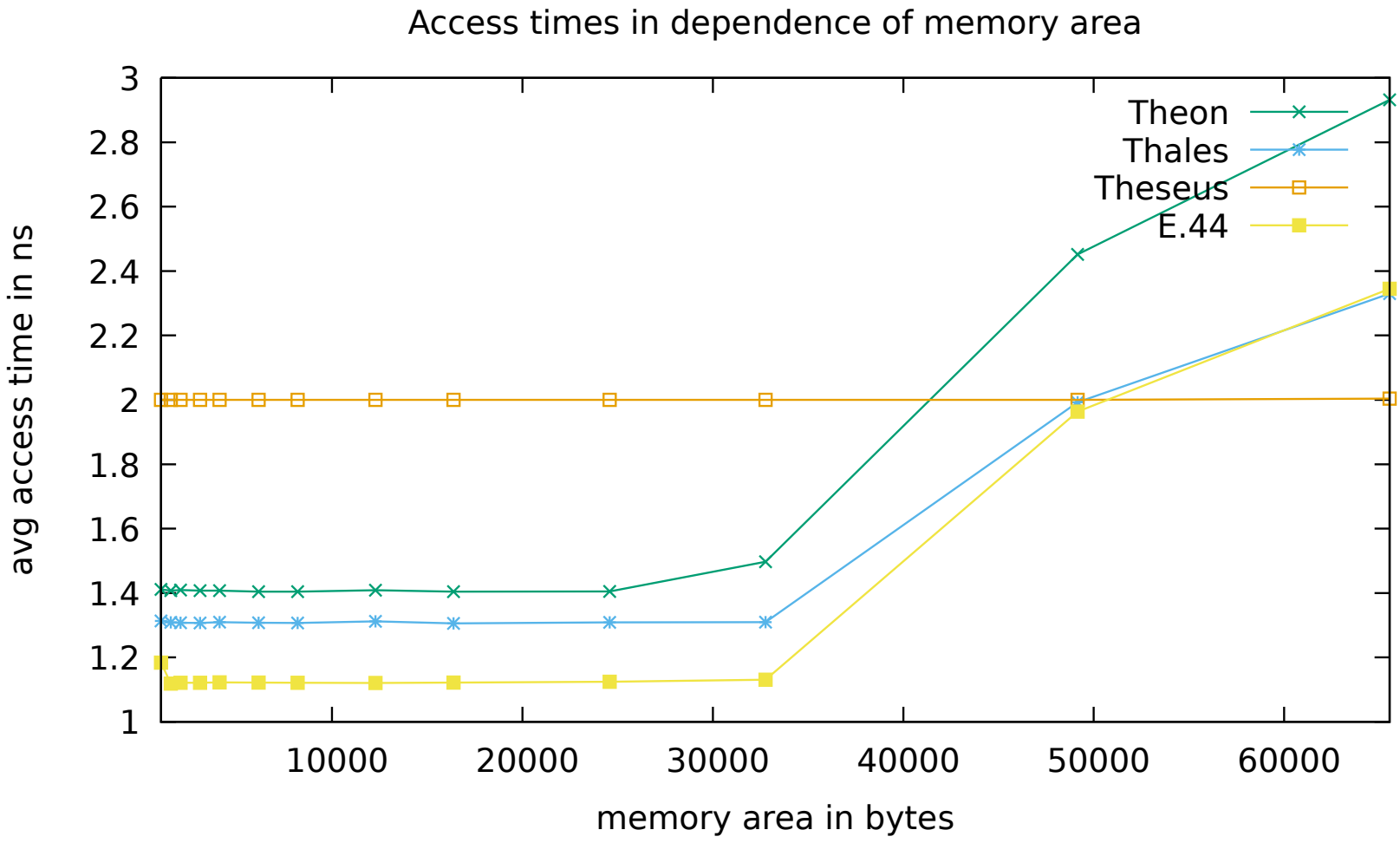
Thales: 2 Intel X5650-Prozessoren mit je 6 Kernen mit je 2 Threads
Caches: L1 (32 KiB), L2 (256 KiB), L3 (12 MiB)



Andromeda: 2 SPARC-T4-Prozessoren mit je 8 Kernen mit je 8 Threads
Caches: L1 (16 KiB), L2 (128 KiB), L3 (4 MiB)

Gemessene Speicherzugriffszeiten: Vergleich




Access times in dependence of memory area
avg access time in ns
memory area in bytes
Theon
Thales
Theseus
E.44
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
10000
20000
30000
40000
50000
60000

- Ein Cache ist in sogenannten *cache lines* organisiert, d.h. eine *cache line* ist die Einheit, die vom Hauptspeicher geladen oder zurückgeschrieben wird.
- Jede der *cache lines* umfasst – je nach Architektur – 32 - 128 Bytes. Auf unseren Maschinen sind fast durchweg 64 Bytes.
- Jede der *cache lines* kann unabhängig voneinander gefüllt werden und einem Abschnitt im Hauptspeicher entsprechen.
- Das bedeutet, dass bei einem Zugriff auf `a[i]` mit recht hoher Wahrscheinlichkeit auch `a[i+1]` zur Verfügung steht.
- Entweder sind Caches vollassoziativ (d.h. jede *cache line* kann einen beliebigen Hauptspeicherabschnitt aufnehmen) oder für jeden Hauptspeicherabschnitt gibt es nur eine *cache line*, die in Frage kommt (*fully mapped*), oder jeder Hauptspeicherabschnitt kann in einen von *n* *cache lines* untergebracht werden (*n-way set associative*).

- Es gibt keine portable Möglichkeit, die Cache-Konfiguration zu ermitteln.
- Unter Linux gibt es in der Hierarchie unter */sys/devices/system/cpu/cpu0/cache* für jeden der Caches der *cpu0* ein Verzeichnis *index0*, *index1* usw.
- In jedem Cache-Verzeichnis finden sich folgende Dateien:

level	Level des Cache, typischerweise 1, 2 oder 3
type	„Data“, „Instruction“ oder „Unified“
coherency_line_size	Größe einer *cache line*
ways_of_associativity	wieviele *cache lines* kommen in Frage, um einen Abschnitt unterzubringen? Alle *cache lines*, die auf diese Weise zusammengehören, werden einem Set zugeordnet.
number_of_sets	Zahl der Sets (s.o.)
size	Produkt aus *coherency_line_size*, *ways_of_associativity* und *number_of_sets*.

Ziel ist es, die zur Verfügung stehenden CPUs auszulasten, d.h. es sollte möglichst wenig Zeit damit verbracht werden, auf Speicherzugriffe zu warten. Dazu gibt es folgende Ansätze:

- Cache-optimierte Datenstrukturen mit möglichst wenig Indirektionen durch Zeiger. Eine Matrix sollte beispielsweise zusammenhängend im Speicher abgelegt werden und nicht mit Hilfe einer Zeigerliste (wie in Java) realisiert werden.
- Einsatz fortgeschrittener Optimierungstechniken wie *instruction schedulung*, ggf. in Verbindung mit *loop unrolling*.

- Bei einem Mehrprozessorsystem hat jede CPU ihre eigenen Caches, die voneinander unabhängig gefüllt werden.
- Problem: Was passiert, wenn mehrere CPUs die gleiche *cache line* vom Hauptspeicher holen und sie dann jeweils verändern? Kann es passieren, dass konkurrierende CPUs von unterschiedlichen Werten im Hauptspeicher ausgehen?

- Die Eigenschaft der Cache-Kohärenz stellt sicher, dass es nicht durch die Caches zu Inkonsistenzen in der Sichtweise mehrerer CPUs auf den Hauptspeicher kommt.
- Die Cache-Kohärenz wird durch ein Protokoll sichergestellt, an dem die Caches aller CPUs angeschlossen sind. Typischerweise erfolgt dies durch Broadcasts über einen sogenannten Snooping-Bus, über den jeder Cache den anderen Caches über Änderungen benachrichtigen kann.
- Das hat zur Folge, dass bei konkurrierenden Zugriffen auf die gleiche *cache line* einer der CPUs sich diese wieder vom Hauptspeicher holen muss, was zu einer erheblichen Verzögerung führen kann.
- Deswegen sollten konkurrierende Threads nach Möglichkeit Schreibzugriffe auf die gleichen *cache lines* vermeiden.

vectors.cpp

```
template<typename T>
void axpy(std::size_t n, T alpha, const T* x,
      std::ptrdiff_t incX, T* y, std::ptrdiff_t incY) {
   for (std::size_t i = 0; i < n; ++i, x += incX, y += incY) {
      *y += alpha * *x;
   }
}
```

- Die Funktion *axpy* berechnet

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} \leftarrow \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} + \alpha \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

- Sowohl $\vec{x}$ als auch $\vec{y}$ liegen dabei nicht notwendigerweise als zusammenhängende Arrays vor. Stattdessen gilt $x_i = x[i*incX]$ (für $\vec{y}$ analog), so dass auch etwa die Spalte einer Matrix übergeben werden kann, ohne dass deswegen eine zusätzliche (teure) Umkopieraktion notwendig wird.

Abschnittsweise Aufteilung

vectors.cpp

```
template<typename T>
void mt_axpy(std::size_t n, T alpha, T* x, std::ptrdiff_t incX, T* y,
             std::ptrdiff_t incY, unsigned int nofthreads) {
   assert(n > 0 && nofthreads > 0);
   std::vector<std::thread> axpy_thread(nofthreads);

   // spawn threads and pass parameters
   std::size_t chunk = n / nofthreads;
   unsigned int remainder = n % nofthreads;
   std::ptrdiff_t nextX = 0; std::ptrdiff_t nextY = 0;
   for (unsigned int i = 0; i < nofthreads; ++i) {
      std::size_t len = chunk;
      if (i < remainder) ++len;
      if (len == 0) break; // # of threads > n
      axpy_thread[i] = std::thread([=]() -> void {
         axpy(len, alpha, x + nextX * incX, incX, y + nextY * incY, incY);
      });
      nextX += len; nextY += len;
   }

   // wait for all threads to finish
   for (auto& t: axpy_thread) {
      if (t.joinable()) t.join();
   }
}
```

Alternierende Aufteilung

bad-vectors.cpp

```
template<typename T>
void mt_axpy(std::size_t n, T alpha, T* x, std::ptrdiff_t incX, T* y,
             std::ptrdiff_t incY, unsigned int nofthreads) {
   assert(n > 0 && nofthreads > 0);
   std::vector<std::thread> axpy_thread(nofthreads);

   // spawn threads and pass parameters
   std::size_t chunk = n / nofthreads;
   unsigned int remainder = n % nofthreads;
   std::ptrdiff_t nextX = 0; std::ptrdiff_t nextY = 0;
   for (unsigned int i = 0; i < nofthreads; ++i) {
      std::size_t len = chunk;
      if (i < remainder) ++len;
      if (len == 0) break; // # of threads > n
      axpy_thread[i] = std::thread([=]() -> void {
         axpy(len, alpha,
            x + i * incX, incX * nofthreads,
            y + i * incY, incY * nofthreads);
      });
      nextX += len; nextY += len;
   }

   // wait for all threads to finish
   for (auto& t: axpy_thread) {
      if (t.joinable()) t.join();
   }
}
```

Laufzeiten je nach Aufteilung


```
theon$ time vectors 10000000 1

real 0m0.280s
user 0m0.084s
sys 0m0.144s
theon$ time vectors 10000000

real 0m0.106s
user 0m0.233s
sys 0m0.634s
theon$ time bad-vectors 10000000

real 0m0.376s
user 0m0.950s
sys 0m2.635s
theon$
```

- Der erste Parameter spezifiziert die Länge der Vektoren $n = 10^7$, der zweite die Zahl der Threads.
- Die ungünstige alternierende Aufteilung führt dazu, dass Threads um die gleichen *cache lines* konkurrieren, so dass erhebliche Wartezeiten entstehen, die zu einer längeren Ausführungszeit führen als bei der Single-Thread-Variante.

```
theseus$ cputrack -c pic0=L2_rd_miss,pic1=L2L3_snoop_inv_sh \
> vectors 10000000 4
   time lwp      event      pic0      pic1
[..]
  0.142   2   lwp_exit       291      6167
  0.208   3   lwp_exit       170      7292
  0.230   4   lwp_exit       175      5811
  0.230   5   lwp_exit       144      7906
[..]
  0.364   6   lwp_exit       141      4405
  0.431   7   lwp_exit       153      6961
  0.449   8   lwp_exit       169      2313
  0.449   9   lwp_exit        50      5382
[..]
  0.696  11   lwp_exit    625166       581
  0.727  10   lwp_exit    625160    159516
  0.727  12   lwp_exit    625164       257
  0.736  13   lwp_exit    625172       404
  0.737   1       exit   2508726    207166
theseus$
```

- Moderne CPUs erlauben die Auslesung diverser Cache-Statistiken. Unter Solaris geht dies mit *cputrack*. Hier liegt die Zahl der L2- und L3-Cache-Invalidierungen wegen Parallelzugriffen bei 207.166, während bei der ungünstigen Aufteilung diese auf 10.503.359 ansteigt...

```
theseus$ cputrack -c pic0=L2_rd_miss,pic1=L2L3_snoop_inv_sh \
> bad-vectors 10000000 4
   time lwp      event      pic0      pic1
  0.020   2 lwp_create         0         0
  0.061   3 lwp_create         0         0
  0.114   4 lwp_create         0         0
  0.201   5 lwp_create         0         0
  0.427   3   lwp_exit       294    817869
  0.491   2   lwp_exit       177    784073
  0.491   4   lwp_exit       153   1242056
  0.510   5   lwp_exit       172    773822
  0.519   6 lwp_create         0         0
  0.561   7 lwp_create         0         0
  0.601   8 lwp_create         0         0
  0.641   9 lwp_create         0         0
  0.935   7   lwp_exit       156   1115834
  0.935   8   lwp_exit        48   1120283
  0.939   6   lwp_exit       174    928491
  0.939   9   lwp_exit       160       168
  0.945  10 lwp_create         0         0
  0.961  11 lwp_create         0         0
  0.991  12 lwp_create         0         0
  1.011  13 lwp_create         0         0
  1.599  11   lwp_exit   1575955   1218185
  1.599  13   lwp_exit   1200570   1195271
  1.620  10   lwp_exit   2500159   1218183
  1.620  12   lwp_exit   2500152     88961
  1.621   1       exit   7784993  10503359
theseus$
```

- Es sind hier nicht alle Threads gleichermaßen betroffen, da jeweils zwei Threads sich einen Prozessor (mit zwei Kernen) teilen, die den L2- und L3-Cache gemeinsam nutzen. (Den L1-Cache gibt es jedoch für jeden

```
theon$ cputrack -c \
> pic0=l2_rqsts.demand_data_rd_miss,pic1=l2_lines_in.i,pic2=l2_lines_out.demand_dirty \
> vectors 10000000 4
   time lwp      event      pic0      pic1      pic2
[..]
  0.071   2   lwp_exit      6947         0    173843
  0.088   3   lwp_exit     14938         0    241217
  0.088   4   lwp_exit     13801         0    227001
  0.106   5   lwp_exit      4573         0    138879
[..]
  0.150   6   lwp_exit      5416         0    126920
  0.153   9 lwp_create         0         0         0
  0.171   7   lwp_exit     16675         0    159869
  0.188   9   lwp_exit     11065         0    125569
  0.189   8   lwp_exit     12306         0    127164
[..]
  0.209  11   lwp_exit     17736         0     60528
  0.214  13 lwp_create         0         0         0
  0.221  13   lwp_exit     23740         0     72545
  0.222  10   lwp_exit     45862         0     68643
  0.222  12   lwp_exit     93323         0     64846
  0.224   1       exit    282471         0   1593124
theon$
```

```
theon$ cputrack -c \
> pic0=l2_rqsts.demand_data_rd_miss,pic1=l2_lines_in.i,pic2=l2_lines_out.demand_dirty bad-
vectors \
> 10000000 4
   time lwp      event      pic0      pic1      pic2
  0.012   2 lwp_create         0         0         0
  0.051   3 lwp_create         0         0         0
  0.061   4 lwp_create         0         0         0
  0.071   5 lwp_create         0         0         0
  0.188   2   lwp_exit     23118        32    415689
  0.188   3   lwp_exit     23534        30    393471
  0.189   4   lwp_exit     22914        61    398109
  0.189   5   lwp_exit     22495        33    398287
  0.192   6 lwp_create         0         0         0
  0.231   7 lwp_create         0         0         0
  0.272   8 lwp_create         0         0         0
  0.272   6   lwp_exit      2607         0    945222
  0.287   9 lwp_create         0         0         0
  0.287   7   lwp_exit      1261         0   1009765
  0.296   9   lwp_exit      1240         0    967632
  0.309   8   lwp_exit      1190         0    864913
  0.312  10 lwp_create         0         0         0
  0.330  11 lwp_create         0         0         0
  0.330  10   lwp_exit   1362846         0   1040161
  0.332  12 lwp_create         0         0         0
  0.350  13 lwp_create         0         0         0
  0.350  12   lwp_exit   1371179         0    964744
  0.351  11   lwp_exit   1537411         0    988887
  0.373  13   lwp_exit   1034884         0    909211
  0.374   1       exit   5419798       156   9302393
theon$
```

- Gegeben seien die Prozessoren $CPU_1, \ldots, CPU_n$, die die Prozesse $P_1, \ldots, P_n$ ausführen.
- Jede der Prozesse P_i führt sequentiell Instruktionen aus, zu denen auch Speicher-Instruktionen gehören.
- Die Speicher-Instruktionen lassen sich vereinfacht einteilen:
  - Schreib-Instruktionen, die den Inhalt einer Speicherzelle an einer gegebenen Adresse ersetzen und als abgeschlossen gelten, sobald der neue Inhalt für alle anderen Prozessoren sichtbar ist.
  - Lade-Instruktionen, die den Inhalt einer Speicherzelle an einer gegebenen Adresse auslesen und als abgeschlossen gelten, sobald der geladene Inhalt durch Schreib-Instruktionen anderer Prozessoren nicht mehr beeinflusst werden kann.
  - Atomare Lade/Schreib-Instruktionen, die beides miteinander integrieren und sicherstellen, dass die adressierte Speicherzelle nicht mittendrin durch andere Instruktionen verändert wird.
- (Diese und die folgenden Definitionen und Notationen wurden dem Kapitel D des *SPARC Architecture Manual* entnommen, siehe `https://sparc.org/technical-documents/#V9`.)

$$X_i <_p Y_i$$

- X_i und Y_i seien beliebige Speicher-Operationen des Prozesses P_i.
- Dann gilt die auf den Prozess P_i bezogene Relation $X_i <_p Y_i$, wenn die Instruktion, die zu X_i führt, vor der Instruktion ausgeführt wird, die zu Y_i führt.
- Diese Relation reflektiert die Reihenfolge, die durch das Programm gegeben ist, das durch den Prozess P_i ausgeführt wird. Dies ist nicht notwendigerweise die Reihenfolge, in der die entsprechenden Operationen tatsächlich ausgeführt werden.
- Die Relation $<_p$ ist eine Totalordnung, d.h. es gilt entweder $X_i <_p Y_i$ oder $Y_i <_p X_i$, falls $X_i \neq Y_i$.

$$X_i <_d Y_i$$

- Es gilt $X_i <_d Y_i$ genau dann, wenn $X_i <_p Y_i$ und mindestens eine der folgenden Bedingungen erfüllt ist:
  - Y_i ist oder enthält eine Schreib-Instruktion, die einer bedingten Sprunganweisung folgt, die von X_i abhängt.
  - Y_i liest ein Register, das von X_i abhängt.
  - Die Schreib-Instruktion X_i und die Lade-Instruktion Y_i greifen auf die gleiche Speicherzelle zu.
- $<_d$ ist eine partielle Ordnung, die transitiv ist.

$$X <_m Y$$

- Die totale Speicher-Ordnung reflektiert die Reihenfolge, in der die Speicher-Operationen zur Laufzeit umgesetzt werden.
- Die Reihenfolge ist im allgemeinen nicht deterministisch.
- Stattdessen gibt es in Abhängigkeit des zur Verfügung stehenden Speichermodells einige Zusicherungen.
- Ferner können auch gewisse Ordnungen erzwungen werden.
- Die Relation $<_m$ reflektiert die Restriktion, dass X vor Y erfolgen muss.
- Das Speichersystem ist in der Wahl der Ausführungsreihenfolge frei, sofern die durch $<_m$ definierte partielle Ordnung respektiert wird.

$$M(X, Y)$$

- Alle gängigen Prozessorarchitekturen bieten Instruktionen an, die eine Ordnung erzwingen. Diese werden *memory barriers* bzw. bei den Intel/AMD-Architekturen *fences* genannt.
- Eine Barrier-Instruktion wirkt sich zunächst unmittelbar auf die Instruktionen aus, zwischen denen sie steht.
- Hinzu kommt, dass sie über alle Prozessoren hinweg erzwingt, dass bestimmte Operationen abgeschlossen werden müssen, bevor bestimmte Operationen begonnen werden können.
- Eine Barrier-Instruktion spezifiziert, welche Sequenz von Speicher-Instruktionen geordnet wird. Prinzipiell gibt es vier Varianten bzw. Kombinationen davon: Lesen-Lesen, Lesen-Schreiben, Schreiben-Lesen und Schreiben-Schreiben.
- $M(X, Y)$ gilt dann, wenn wegen einer dazwischenliegende Barrier-Instruktion zuerst X und dann Y ausgeführt werden muss.

- Lesen-Lesen: Alle Lese-Operationen vor der Barrier-Instruktion müssen beendet sein, bevor folgende Lese-Operationen durchgeführt werden können.
- Lesen-Schreiben: Alle Lese-Operationen vor der Barrier-Instruktion müssen beendet sein, bevor die folgenden Schreib-Operationen für irgendeinen Prozessor sichtbar werden.
- Schreiben-Lesen: Die Schreib-Operationen vor der Barrier-Instruktion müssen abgeschlossen sein, bevor die folgenden Lese-Operationen durchgeführt werden.
- Schreiben-Schreiben: Die Schreib-Operationen vor der Barrier-Instruktion müssen alle beendet sein, bevor die folgenden Schreib-Operationen ausgeführt werden.

- Auf der x86-Architektur:

LFENCE	Lesen-Lesen
SFENCE	Schreiben-Schreiben
MFENCE	Lesen/Schreiben-Lesen/Schreiben

  All diese Operationen serialisieren global.

- Auf der SPARC-Architektur:

  MEMBAR *membar_mask* mit $membar\_mask = cmask \mid mmask$

  Mögliche kombinierbare Bits für *mmask*: #StoreStore, #LoadStore, #StoreLoad und #LoadLoad.

  Mögliche kombinierbare Bits für *cmask*: #Sync, #MemIssue und #Lookaside. (Per Voreinstellung wirkt sich **MEMBAR** nur auf die Operationen eines Prozessors aus, bei #Sync wirkt sich das global auf alle Prozesse aus.)

Relaxed Memory Order (RMO)

Eine Speicherordnung $<_m$ ist RMO genau dann, wenn

- $X <_d Y \wedge L(X) \Rightarrow X <_m Y$
- $M(X, Y) \Rightarrow X <_m Y$
- $Xa <_p Ya \wedge S(Y) \Rightarrow X <_m Y$

Xa steht für eine Speicher-Instruktion auf der Speicherzelle an der Adresse a; die Prädikate $L(X)$ und $S(Y)$ treffen zu, wenn X eine Lese-Instruktion ist oder umfasst bzw. Y eine Schreib-Instruktion ist oder umfasst.

Die drei Punkte bedeuten im Einzelnen:

- RMO sichert zu, dass die Abhängigkeitsrelation beachtet wird, wenn die vorangehende Instruktion eine Lade-Operation ist.
  Darauffolgende Speicher-Instruktionen können sich verzögern und ihre Ausführungreihenfolge ist zunächst nicht festgelegt.
- Memory-Barriers werden respektiert.
- Bei Speicher-Instruktionen an die gleiche Adresse wird die durch das Programm gegebene Ordnung respektiert.

- *Partial Store Order* (PSO) erfüllt alle Bedingungen der RMO. Hinzu kommt, dass allen ladenden Speicher-Instruktionen implizit eine Barrier-Instruktion für Lesen-Lesen und Lesen-Schreiben folgt.
- *Total Store Order* (TSO) erfüllt alle Bedingungen der PSO. Hinzu kommt, dass allen schreibenden Speicher-Instruktionen implizit eine Barrier-Instruktion für Schreiben-Schreiben folgt.
- *Sequential Consistency* (SQ) erfüllt alle Bedingungen der TSO. Hinzu kommt, dass allen schreibenden Speicher-Instruktionen eine Barrier-Instruktion für Schreiben-Lesen folgt.
- SQ ist das einfachste Modell, bei dem die Reihenfolge der Instruktionen eines einzelnen Prozesses sich direkt aus der Programmordnung ergibt. Andererseits bedeutet SQ, dass bei einem Schreibzugriff sämtliche Speicherzugriffe blockiert werden, bis die Cache-Invalidierungen abgeschlossen sind. Das ist sehr zeitaufwendig.

- Gegeben seien die beiden globalen Variablen a und b mit einem Initialwert von 0 und drei Prozesse mit den folgenden Instruktionen:

P_1	P_2	P_3
$a = 1;$	**int** $r1 = a;$	**int** $r1 = b;$
$b = 1;$	**int** $r2 = b;$	**int** $r2 = a;$

- Mögliche Szenarien (bei SQ und TSO nicht vollständig):

	P_2: $r1$	P_2: $r2$	P_3: $r1$	P_3: $r2$
Unter SQ:	0	0	0	0
	1	1	1	1
Unter TSO, !SQ:	1	0	1	1
Unter PSO, !TSO:	0	0	1	0
	0	1	1	0
	1	1	1	0
Unter RMO, !PSO:	1	0	1	0

- Jede Prozessorarchitektur bietet einige Datentypen an, die atomar geladen oder geschrieben werden.
- Atomizität bedeutet hier, dass bei einer Lade-Instruktion, die einer Speicher-Instruktion folgt, entweder einer der früheren Werte oder der neue Wert zu sehen ist.
- Wenn beispielsweise ein 32-Bit-Wort auf einer 4-Byte-Kante sich in diesem Sinne atomar verhält, kann es nicht passieren, dass beim Lesen die ersten zwei Bytes noch den alten Wert aufweisen, die letzten zwei Bytes den neuen.
- Entscheidend für die Eigenschaft der Atomizität eines Datentyps ist die Größe, das Alignment bzw. der Punkt, ob die Daten vollständig in einer Cache-Line liegen.

- Bei den x86-Architekturen von Intel wird Atomizität für Lese- und Schreibzugriffe folgender Datentypen zugesichert:
  - einzelne Bytes (also etwa **char**),
  - 16-Bit-Worte auf 2-Byte-Kanten (also etwa **short**),
  - 32-Bit-Worte auf 4-Byte-Kanten (also etwa **int**),
  - 64-Bit-Worte auf 8-Byte-Kanten ab der Pentium-Architektur (also etwa **long long int** oder **double**),
  - 16-Bit-Worte, die in ein 4-Byte-Wort auf einer 4-Byte-Kante fallen ab der Pentium-Architektur und
  - 16-, 32- und 64-Bit-Worte, die in beliebiger Weise in eine Cache-Line fallen ab der Pentium-Pro-Architektur (P6).
- Bei der SPARCv9-Architektur wird die Atomizität für alle 64-Bit-Worte auf 8-Byte-Kanten (und alles, was kleiner ist) zugesichert.

- Alle gängigen Prozessorarchitekturen bieten atomare Instruktionen auf Maschinenebene an.
- Jede dieser Instruktionen lädt und speichert (in dieser Reihenfolge) von und in den Hauptspeicher in einer Weise, die sicherstellt, dass keine andere Lade- oder Speicherinstruktion für die gleiche Speicherlokation dazwischen ausgeführt wird.
- Ebenso werden fremde Traps während der Ausführung der Instruktion unterdrückt.
- Die Sichtbarkeit des neu gespeicherten Werts kann sich aber durchaus verzögern; d.h. unter Umständen ist es notwendig, auf eine atomare Instruktion noch eine Barrier-Instruktion folgen zu lassen.

- Die einfachste atomare Lade- und Speicheroperation lädt den alten Wert eines atomaren Datentyps aus dem Speicher in ein Register und schreibt den Inhalt eines anderen Registers in die gleiche Speicherzelle hinein.
- Auf der x86-Architektur, jeweils $tmp = r$; $r = m$; $m = tmp$; wobei r ein Register und m eine entsprechende Speicherzelle ist.

  XCHG *r8,m8* 8-Bit-Operation
  XCHG *r16,m16* 16-Bit-Operation
  XCHG *r32,m32* 32-Bit-Operation
  XCHG *r64,m64* 64-Bit-Operation (nur im 64-Bit-Modus)
- Auf der SPARC-Architektur geht dies mit

  LDSTUB *m8,r8* 8-Bit-Operation
  SWAP *m32,r32* 32-Bit-Operation

Die SWAP-Instruktionen können für einfache Locks verwendet werden, die einen gegenseitigen Ausschluss ermöglichen. Gegeben sei eine Datenstruktur und eine **bool**-Variable, die mit einem Byte repräsentiert wird und nur die Werte 0 (Lock ist frei) und 1 (Lock ist belegt) unterstützt:

```
r = 1;
XCHG r,lock;
if (r == 0) {
    /* lock was free and is now set to 1 */
    /* critical region with exclusive access */
    /* memory barrier: write–read */
    XCHG r,lock;
} else {
    /* no access this time */
}
```

Wenn sichergestellt ist, dass im kritischen Bereich nur wenige Instruktionen ausgeführt werden, deren Ausführungszeit sehr beschränkt ist, können die anderen ggf. ungeduldig mit einer Schleife auf die Freigabe des Locks warten:

```
r = 1;
do {
    XCHG r,lock;
} while (r == 1);
/* lock was free and is now set to 1 */
/* critical region with exclusive access */
/* memory barrier: write–read */
XCHG r,lock;
```

Achtung: Wenn die kritische Region durch einen TRAP unterbrochen wird, dann hängen die anderen Threads u.U. sehr lange. Wenn innerhalb einer Signalbehandlungsfunktion ein Versuch unternommen wird, den Lock zu gewinnen, haben wir möglicherweise einen Deadlock.

- Neben der SWAP-Instruktion wird auf allen gängigen Architekturen auch eine bedingte Instruktion angeboten, die atomar den alten Wert lädt, ihn mit einen vorgegebenen Wert vergleicht und im Falle der Gleichheit den Inhalt eines anderen Registers in die gleiche Speicherzelle schreibt. Typischerweise nennt die Instruktion sich CAS (*compare and set*).
- Auf der x86-Architektur, jeweils implizit mit dem Vergleichswert im Register AL, EAX oder RAX, je nach Wortbreite:

CMPXCHG *m8,r8*	8-Bit Operation
CMPXCHG *m16,r16*	16-Bit Operation
CMPXCHG *m32,r32*	32-Bit Operation
CMPXCHG *m64,r64*	64-Bit Operation
CMPXCHG8B *m64*	64-Bit Operation mit EDX:EAX und ECX:EBX
CMPXCHG16B *m128*	128-Bit Operation mit RDX:RAX und RCX:RBX

- Auf der SPARC-Architektur:

CAS *m32,r32,r32*	32-Bit Operation
CASX *m64,r64,r64*	64-Bit Operation

Die 32-Bit-Variante der atomaren CMPXCHG-Operation in Pseudo-Code als **bool**-wertige Funktion:

```
atomic bool CMPXCHG(&mem, &eax, r) {
    if (eax == mem) {
        mem = r; return true;
    } else {
        eax = mem; return false;
    }
}
```

Mit Hilfe einer atomaren CAS-Instruktion ist es möglich, ein neues Element in eine einfache lineare Liste einzufügen:

```
/* initialization */
struct Element { T* info; Element* next; };
Element* head = 0;

/* adding an element in front of the list pointed to by head */
void add_element(Element* head, Element* new_element) {
    r = new_element; eax = head;
    do {
        r->next = eax;
    } while (!CMPXCHG(head, eax, r));
}
```

Vorteile dieser Lösung:

- Die Schleife muss nur wiederholt werden, wenn **CMPXCHG** scheitert.
- Dies passiert aber nur, wenn ein konkurrierender Prozess aktiv ist und das gleiche probiert.
- Ein inaktiver Prozess kann (anders als beim Spin-Lock) hier niemanden aufhalten.
- Das Verfahren ist somit frei von Deadlocks und längeren Verzögerungen.
- Es gilt nur, dass ABA-Problem zu beachten (dazu später mehr).

- Das Speichermodell in C++ berücksichtigt zunächst nicht mehrere Threads. Stattdessen wird zunächst nur sichergestellt, dass der Speicher sich aus der Sicht eines Threads intuitiv korrekt verhält, d.h. SQ (*sequential consistency*) erfüllt ist.
- Wenn mehrere Threads auf die gleiche Datenstruktur zugreifen und dies nicht durch die Verwendung von *std*::*mutex*-Variablen geregelt wird, dann liegt ein *data race* vor und der Effekt ist undefiniert.
- Die Verwendung von **volatile** hilft hier nicht (anders als in Java).
- Da (wie präsentiert) die gängigen Prozessorarchitekturen Barriers, atomare Datentypen und atomare Instruktionen anbieten, lag es auch nahe, dies in C++ zu unterstützen.
- Die große Herausforderung lag hier darin, dass die zu unterstützenden Architekturen sehr unterschiedlich sind und dass andererseits eine Vereinfachung (wie etwa in Java) Optimierungsmöglichkeiten verschenkt.

- Wenn auf gemeinsame Datenstrukturen konkurrierend zugegriffen werden soll, ohne dies durch *std*::*mutex*-Variablen zu regeln, dann ist der Einsatz atomarer Datentypen zwingend notwendig.
- Atomare Datentypen werden mit Hilfe der *std*::*atomic*-Template-Klasse deklariert:

```
#include <atomic>
// ...
std::atomic<unsigned int> counter {0};
```

- Die **bool**-wertige Methode *is_lock_free* teilt mit, ob das Lesen und Schreiben des gesamten Objekts atomar ohne Locks möglich sind. Bei elementaren Datentypen ist dies normalerweise möglich, bei zusammengesetzten Datentypen eher nicht.
- Die Klasse bietet *load*- und *store*-Methoden, die eine Spezifikation der gewünschten *memory_order* ermöglichen.

Grundsätzlich kann auch *std*::*atomic* für selbstdefinierte Typen unter Voraussetzungen verwendet werden:

- Trivialer Default-Konstruktor
- Trivialer Dekonstruktor
- Standard-Layout
- Initialisierbar als Aggregat

Zulässig sind somit traditionelle Typen aus C, d.h. Tupel aus elementaren Datentypen einschließlich Zeigern. Entsprechend können solche Datentypen sicher mit *memcpy* kopiert und mit *memcmp* verglichen werden. Virtuelle Methoden sind nicht zulässig, Vergleichs-Operatoren werden nicht honoriert.
Neuere Übersetzer (wie g++ 6.x) unterstützen atomare Aktualisierungen solcher Typen ohne Locks, wenn das die Architektur unterstützt (bis zu 16 Bytes bei neueren x86-Architekturen).

- *std*::*memory_order* ist in C++ wie folgt definiert:

```
enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_ret,
    memory_order_seq_cst
};
```

- Zu beachten ist hier, dass dies nicht nur die Prozessorarchitektur betrifft, sondern auch die Freiheit des Übersetzers, die Anordnung von Lade- und Speicherinstruktionen umzuordnen (*instruction scheduling*).
- Wenn nichts explizit angegeben wird, dann kommt *memory_order_seq_cst* zum Einsatz (*sequential consistency*), die dem SQ-Modell entspricht. Dies lässt sich ggf. noch nachvollziehen, kostet aber den Einsatz entsprechender Barriers.
- Die anderen Varianten erschweren sehr den Nachweis, kommen aber teilweise ohne Barriers aus.

- *memory_order_relaxed* steht dafür, dass es keinerlei Zusicherung über die Ausführungsreihenfolge gibt.
- Bjarne Stroustrup nennt dafür folgendes Beispiel mit den atomaren ganzzahligen Variablen x und y, jeweils mit 0 initialisiert:

```
// thread 1:
    int r1 = y.load(memory_order_relaxed);
    x.store(r1, memory_order_relaxed);
// thread 2:
    int r2 = x.load(memory_order_relaxed);
    y.store(42, memory_order_relaxed);
```

- Dann ist es hier möglich, dass der zweite Thread danach den Wert 42 in der Variable *r2* vorfindet. Das liegt daran, dass sogar die Anordnung der Anweisungen des gleichen Threads sogar dann flexibilisiert werden, wenn dem Abhängigkeiten entgegenstehen. Denkbar ist also:

```
y.store(42, memory_order_relaxed); // thread 2
int r1 = y.load(memory_order_relaxed); // thread 1
x.store(r1, memory_order_relaxed); // thread 1
int r2 = x.load(memory_order_relaxed); // thread 2
```

- *memory_order_relaxed* erfüllt nicht die Bedingungen, die an die RMO (*relaxed memory order*) im SPARC-Architecture-Manual gestellt werden.
- Bei der x86-Architektur und SPARC kann dies auf der Maschinenebene nicht vorkommen. Eine der Ausnahmen ist die DEC-Alpha-Architektur.
- Aber prinzipiell gibt *memory_order_relaxed* dem Übersetzer auch auf anderen Architekturen die Freiheit, entsprechende Anweisungen zu vertauschen.

Zähler mit *memory_order_relaxed*

counter.cpp

```
std::atomic<unsigned int> counter {0};

void count() {
   for (unsigned int i = 0; i < 1000; ++i) {
      counter.fetch_add(1, std::memory_order_relaxed);
   }
}

int main() {
   std::vector<std::thread> threads;
   for (unsigned int i = 0; i < 10; ++i) {
      threads.push_back(std::thread(count));
   }
   for (auto& t: threads) t.join();
   std::cout << "counter = " << counter << std::endl;
}
```

- *fetch_add* ist eine atomare Lade- und Schreiboperation, die den Wert inkrementiert.
- Die Reihenfolge der Additionen ist hier nicht relevant, deswegen ist *memory_order_relaxed* ausreichend. Der Haupt-Thread kann das Resultat korrekt ausgeben, da er sich mit *join* synchronisiert.

- Häufig ist eine atomare Variable mit einer Datenstruktur verbunden. Wenn die Datenstruktur und danach die atomare Variable aktualisiert wird, soll beim Feststellen des neuen Werts der atomaren Variable auch die Datenstruktur aktualisiert vorgefunden werden.
- Zwei Threads P_1 und P_2 gelten als synchronisiert, wenn
  - Prozess P_1 die atomare Variable a mit *memory_order_release* aktualisiert und
  - Prozess P_2 lesenderweise auf die atomare Variable a mit *memory_order_acquire* zugreift und dabei den von P_1 geschriebenen Wert vorfindet.
- Für die entsprechende Schreiboperation X des Prozesses P_1 und den aktualisierten Wert vorfindenden Lese-Operation Y des Prozesses P_2 gilt dann die Relation $X <_s Y$.

Die *happens-before*-Relation $<_{hb}$ in C++ für einen Thread sei dann die minimale Relation, die folgende Bedingungen erfüllt:

- $X <_p Y \Rightarrow X <_{hb} Y$
- $X <_s Y \Rightarrow X <_{hb} Y$
- $X <_{hb} Y \wedge Y <_{hb} Z \Rightarrow X <_{hb} Z$

rel-acq.cpp

```
std::atomic<bool> done {false};
unsigned int result_of_overlong_computation = 0;

void compute() {
   result_of_overlong_computation = 42;
   done.store(true, std::memory_order_release);
}

void print_result() {
   /* busy loop that waits for done to become true */
   while (!done.load(std::memory_order_acquire)) {
      std::this_thread::yield();
   }
   std::cout << result_of_overlong_computation << std::endl;
}

int main() {
   std::thread t1(compute); std::thread t2(print_result);
   t1.join(); t2.join();
}
```

rel-acq.cpp

```
void compute() {
   result_of_overlong_computation = 42;
   done.store(true, std::memory_order_release);
}

void print_result() {
   /* busy loop that waits for done to become true */
   while (!done.load(std::memory_order_acquire)) {
      std::this_thread::yield();
   }
   std::cout << result_of_overlong_computation << std::endl;
}
```

- Für die Berechnung X und die Ausgabe Y gilt hier $X <_{hb} Y$, da *done.store* und *done.load* zur Synchronisierung führen, sobald *done.load* den von *done.store* abgespeicherten Wert zu sehen bekommt.
- *std*::*this_thread*::*yield*() weist den Scheduler an, dass andere Threads jetzt eher zum Zuge kommen sollten. Das ist etwas freundlicher als eine reine *busy*-Loop.

memory_order_acq_rel
: Vereinigt *memory_order_acquire* mit *memory_order_release* und ist insbesondere sinnvoll für atomare Lese-und-Schreiboperationen analog zu den SWAP- und CAS-Instruktionen.

memory_order_consume
: Ist die billigere Ausführung von *memory_order_acquire*, bei der die Sequentialisierung nur für die Ausdrücke gewährleistet ist, die direkt oder indirekt von der zu ladenden atomaren Variable abhängen.

memory_order_seq_cst
: Ist die Voreinstellung, die global SQ erzwingt, d.h. mit *memory_order_seq_cst* versehene Lese- und Schreib-Operationen sind global über alle Threads hinweg total geordnet. Das ist vergleichsweise teuer.

Lineare Liste ohne Locks

lflist.hpp

```
template <typename T>
class LFList {
   private:
      struct Element;
   public:
      LFList() : head(nullptr) { }
      /* ... */
   private:
      struct Element {
         Element() : next(nullptr) {
         }
         Element(const T& item) : item(item), next(nullptr) {
         }
         T item;
         Element* next;
      };
      std::atomic<Element*> head;
};
```

- Einfach verkettete Liste mit nur einem Ende.

lflist.hpp

```
void push(const T& item) {
   Element* element = new Element(item);
   Element* p = head.load();
   element->next = p;
   while (!head.compare_exchange_weak(p, element)) {
      element->next = p;
   }
}
```

- *head.compare_exchange_weak*(*p*, *element*) entspricht folgender atomaren Operation:

if (*head* == *p* && /* *some good fortune* */) {
    head = *element*;
    return true;
} **else** {
    p = *head*;
    return false;
}

lflist.hpp

```
void push(const T& item) {
   Element* element = new Element(item);
   Element* p = head.load();
   element->next = p;
   while (!head.compare_exchange_weak(p, element)) {
      element->next = p;
   }
}
```

- Alternativ gibt es *compare_exchange_strong*, da fällt die Nebenbedingung weg. Auf einigen Architekturen sind die beiden unterschiedlich.
- Schleifen mit *compare_exchange_strong* sind keine sinnlosen *busy*-Loops, da im Wiederholungsfall ein anderer Prozess nachweislich vorangekommen ist. Wenn kein Fortschritt andersweitig erfolgt, ist die Schleife in der nächsten Iteration erfolgreich.

Atomares Entfernen aus der Liste

lflist.hpp

```
bool pop(T& item) {
   Element* p = head.load();
   while (p && !head.compare_exchange_weak(p, p->next))
      ;
   if (p) {
      item = p->item;
      return true;
   } else {
      return false;
   }
}
```

- Alle Zugriffsoperationen erfolgen hier implizit mit *memory_order_seq_cst*. Das ist teuer, aber leichter nachzuvollziehen.

lflist.hpp

```
   element->next = p;
   while (!head.compare_exchange_weak(p, element)) {
      element->next = p;
   }
```

Verwendung der Register im folgenden durch *g++* erzeugten Assembler-Text:

%esi *element*
%ecx *&head*
%eax *&p*

```
        movl    %eax, 8(%esi)
        mfence
.L352:
        movl    -64(%ebp), %eax
        lock cmpxchgl   %esi, (%ecx)
        je      .L343
        movl    %eax, -64(%ebp)
        movl    %eax, (%edx)
        mfence
        jmp     .L352
```

lflist.hpp

```
bool pop(T& item) {
   Element* p = head.load();
   // p can be a dangling reference here:
   while (p && !head.compare_exchange_weak(p, p->next))
      ;
   if (p) {
      item = p->item;
      delete p; // now with delete
      return true;
   } else {
      return false;
   }
}
```

- Die vorherige Fassung von *pop* verzichtete auf **delete**. Das ließe sich natürlich wie hier hinzufügen.
- Wenn aber mehrere *pop*-Operationen parallel laufen, kann es passieren, dass *p* auf ein gerade eben gelöschtes Element zeigt.
- Der Zugriff auf *p*–>*next* ist dann nicht mehr wohldefiniert.

lflist.hpp

```
std::atomic<Element*> head;
std::atomic<Element*> free;
```

- Idee: Nicht mehr benötigte Elemente werden nicht mit **delete** freigegeben, sondern in eine andere Liste eingefügt.
- Dann wird zuerst überprüft, ob es noch wiederbenutzbare Elemente in der *free*-Liste gibt, bevor **new** aufgerufen wird.
- Erst bei dem Abbau der Liste werden die verbliebenen Elemente freigegeben:

```
~LFList() {
   Element* next;
   for (Element* p = head.load(); p; p = next) {
      next = p->next;
      delete p;
   }
   for (Element* p = free.load(); p; p = next) {
      next = p->next;
      delete p;
   }
}
```

lflist.hpp

```
void push_element(std::atomic<Element*>& head, Element* element) {
   Element* p = head.load();
   element->next = p;
   while (!head.compare_exchange_weak(p, element)) {
      element->next = p;
   }
}
Element* pop_element(std::atomic<Element*>& head) {
   Element* p = head.load();
   while (p && !head.compare_exchange_weak(p, p->next))
      ;
   return p;
}
```

- Da wir nun zwei Listen haben, lohnt es sich entsprechend verallgemeinerte private Methoden zu haben.

lflist.hpp

```
void push(const T& item) {
   Element* element = pop_element(free);
   if (element) {
      element->item = item;
   } else {
      element = new Element(item);
   }
   push_element(head, element);
}
bool pop(T& item) {
   Element* p = pop_element(head);
   if (p) {
      item = p->item;
      push_element(free, p);
      return true;
   } else {
      return false;
   }
}
```

- Die öffentlichen Methoden *push* und *pop* berücksichtigen nun die *free*-Liste.

lflist.hpp

```
Element* pop_element(std::atomic<Element*>& head) {
   Element* p = head.load();
   while (p && !head.compare_exchange_weak(p, p->next))
      ;
   return p;
}
```

- Der Aufruf von *head.compare_exchange_weak* erfolgt in drei Schritten:
  1. *p* laden,
  2. *p*–>*next* laden und die
  3. Methode *compare_exchange_weak* aufrufen.
- Zwischen dem zweiten und dritten Schritt könnte eine parallel laufende *pop*-Operation das Element entfernen, wodurch es in die *free*-Liste eingefügt wird. Wenn weitere Änderungen erfolgen einschließlich einem *push*, dann könnte aus der *free*-Liste das zuvor gelöschte Element wieder eingefügt werden.
- Dann hat möglicherweise der erste Parameter den gleichen Wert von *head*, allerdings ist es dann gut möglich, dass der zweite Parameter nicht mehr *p*–>*next* entspricht.

- Bei *compare_exchange_weak* aktualisieren wir atomar einen Wert (hier *head*) unter der impliziten Annahme, dass während des Updates der zweite Parameter noch den Wert von *p–>head* hat.
- Dies muss aber nicht der Fall sein.
- Das ist das ABA-Problem, d.h. wir „sehen“ *A* und übergeben die beiden Parameter, dann gibt es zwischendurch *B*, das nicht beobachtet wird und danach kehrt *A* zurück (jedoch mit einem anderen *next*-Zeiger), worauf mit dem Update die Datenstruktur korrumpiert wird.

ABA-Problem: Szenario

thread #1	thread # 2	thread # 3	Status
			head $\longrightarrow$ *A* $\longrightarrow$ *B* $\longrightarrow$ *C* *free* $\longrightarrow$
pop *pop_element*(*head*) *p* = *head.load*() // *p points to A* *load p−>next* // *p−>next points to B*			
	pop		*head* $\longrightarrow$ *B* $\longrightarrow$ *C* *free* $\longrightarrow$ A
	pop *p* = *pop_element*(*head*)		*head* $\longrightarrow$ *C* *free* $\longrightarrow$ A
		push	*head* $\longrightarrow$ *A* $\longrightarrow$ *C* *free* $\longrightarrow$
head.compare_exchange_weak (*A*, *B*) // *succeeds*			*head* $\longrightarrow$ *B* $\longrightarrow$ *C* *free* $\longrightarrow$

- *A*, *B* und *C* repräsentieren die Adressen von Elementen und nicht deren Inhalt, der sich hier in diesem Szenario bei *A* durch die *push*-Operation bei thread # 3 verändert.

Der prinzipielle Lösungsansatz sieht vor, dass A nicht unerwartet auftaucht, nachdem es durch B ersetzt wurde und während noch ein Thread in einer entsprechenden Ersetzungsschleife sich aufhält.
Dazu gibt es folgende Lösungsansätze:

- Aus A wird ein Tupel, bestehend aus dem eigentlichen zu aktualisierenden Wert und einem zusätzlichen einmaligen Wert. Letzteres kann beispielsweise durch einen atomaren Zähler erreicht werden. Auf diese Weise wird ABA durch ABA' ersetzt, d.h. der zweite Wert des Tupels unterscheidet sich.
- Da das Problem bei Zeigern durch Recycling entsteht, könnte das Recycling zeitlich auf einen Moment verschoben werden, in dem sich kein Thread in einer Ersetzungsschleife befindet.

Beide Ansätze verursachen zusätzliche Kosten, da weitere atomare Operationen anfallen. Nur neuere Übersetzer (wie etwa g++ 6.x) sind in der Lage, Tupel atomar ohne die Verwendung von Locks zu aktualisieren. Dies geht auf moderneren Architekturen, die atomare 128-Bit-Operationen unterstützen.

- Bei einer Datenbank verändern Transaktionen den Zustand in einer Weise, die
  - atomar (Ausführung ganz oder überhaupt nicht),
  - konsistent (wenn die Datenbank zuvor konsistent war, bleibt sie es),
  - isoliert (konkurrierende Transaktionen beeinflussen sich nicht gegenseitig) und
  - dauerhaft

  ist (ACID-Prinzip).
- Dieses Konzept wird auch gerne auf den Hauptspeicher übernommen, wobei hier die Dauerhaftigkeit wegfällt.
- Die prinzipielle Idee ist, dass vom optimistischen Fall ausgegangen wird, dass kein Konflikt vorliegt. Wird dennoch ein solcher festgestellt, dann wird eine der betroffenen Transaktionen zurückgerollt und neu gestartet, sobald die andere abgeschlossen ist.
- Von der Nutzerseite ist ein besonderer Vorteil darin zu sehen, dass die Verwendung expliziter Locks wegfällt.

- Prinzipiell sind jetzt bereits einige Architekturen wie die modernen x86-Prozessoren in der Lage, im Rahmen des Pipelinings im Umgang mit bedingten Sprüngen mehrere Varianten parallel spekulativ zu verfolgen und dann mit Verzögerung sich für eine der Varianten zu entscheiden.
- Das lässt sich auch ausdehnen auf Speicherzugriffe, wenn die Cache-Hierarchie einen spekulativen Zustand unterstützt, der im Erfolgsfalle atomar in den Hauptspeicher abgesichert werden kann oder der im Falle eines Abbruchs der Transaktion wieder zurückgerollt werden kann.
- Der Cache kann dann ähnlich wie die Register zumindest partiell als zum lokalen Zustand eines Cores gehörig betrachtet werden.

Sun Microsystems plante dies für den 2005 bis 2009 entwickelten Rock-Prozessor, der die SPARC-Architektur weiterführen sollte:

- Jeder Core unterstützte 2 reguläre Threads und je nach Konfiguration ein oder zwei sogenannte Scout-Threads, die die spekulative Ausführung einer Transaktion übernahmen.
- Im Rahmen einer Transaktion markierte ein Scout-Thread im eigenen L1-Cache veränderte Cache-Lines und führte in einem Commit-Buffer eine Liste der veränderten Cache-Lines.
- Im Erfolgsfalle wurde der Commit-Buffer genutzt, um die veränderten Cache-Lines aus dem L1 atomar im Hauptspeicher zu sichern.
- Wenn ein eine veränderte Cache-Line durch einen anderen Thread invalidiert wurde, kam es zum Abbruch der Transaktion.
- Die Kapazitäten waren wegen der Beschränkung auf den L1 mit 32 KiB begrenzt.

In seiner Supercomputer-Serie Blue Gene führte IBM 2011 bei den Blue-Gene/Q-Systemen ebenfalls Transactional Memory ein:

- Statt dem L1 wird hier der L2 mit 32 MiB verwendet.
- Es werden keine Scout-Threads benötigt. Stattdessen unterstützt der L2-Cache mehrere Versionen für eine Cache-Line und entsprechende atomare Operationen.
- Seit 2015 wurde von IBM die weitere Entwicklung der Blue-Gene-Familie eingestellt.

TSX sind ein Erweiterung der x86-Architektur, die zunächst auf der Haswell- und Broadwell-Architektur eingeführt wurde, dort aber fehlerhaft implementiert war. Benutzbar ist TSX somit ab der 2015 eingeführten Skylake-Architektur.

- Zum Einsatz kommt hier der L1-Cache mit 32 KiB (auf Haswell).
- Wie bei Rock führen Cache-Konflikte zum Abbruch der Transaktion.
- Wenn eine spekulativ veränderte Cache-Line ersetzt werden muss (*cache line eviction*), dann wird die Transaktion ebenfalls abgebrochen.
- Umgesetzt wird dies über x86-Instruktionspräfixe **XACQUIRE** und **XRELEASE** (sind NOPs bei älteren Architekturen, in Kombination mit dem **LOCK**-Präfix, Ausführung zunächst ohne **LOCK**, im Erfolgsfalle Wiederholung mit **LOCK**) bzw. **XBEGIN** und **XEND**, bei denen explizit spezifiziert wird, wie im Falle einer abgebrochenen Transaktion zu reagieren ist, und die mit der **XABORT**-Instruktion einen Abbruch ermöglichen.

- Mit N4514 gibt es einen Vorschlag für eine entsprechende C++-Erweiterung.
- Momentan ist nicht abzusehen, ob und wann das in den C++-Standard aufgenommen wird. Für C++20 ist dies bislang nicht geplant.
- g++ bietet ab 6.1 eine entsprechende Unterstützung mit der Option „-fgnu-tm“. Nach meiner Erfahrung führt diese aber noch häufig zu internen Übersetzerfehlern und ist daher noch nicht ausgereift.
- Angeboten werden Synchronisationsblöcke (keine Transaktionen, aber gegenseitiger Ausschluss ohne explizite Locks) und atomare Blöcke, die auch den Abbruch einer Transaktion vorsehen. Letztere haben das Potential mit entsprechenden Hardware-Erweiterungen umgesetzt zu werden, kommen daher aber auch mit einer Reihe von Restriktionen.

- OpenMP ist ein seit 1997 bestehender Standard mit Pragma-basierten Spracherweiterungen zu Fortran, C und C++, die eine Parallelisierung auf MP-Systemen unterstützt.
- Pragmas sind Hinweise an den Compiler, die von diesem bei fehlender Unterstützung ignoriert werden können.
- Somit sind alle OpenMP-Programme grundsätzlich auch mit traditionellen Compilern übersetzbar. In diesem Falle findet dann keine Parallelisierung statt.
- OpenMP-fähige Compiler instrumentieren OpenMP-Programme mit Aufrufen zu einer zugehörigen Laufzeitbibliothek, die dann zur Laufzeit in Abhängigkeit von der aktuellen Hardware eine geeignete Parallelisierung umsetzt.
- Die Webseiten des zugehörigen Standardisierungsgremiums mit dem aktuellen Standard finden sich unter `http://www.openmp.org/`. Aktuell ist 5.0 – der GCC unterstützt bislang den Standard in der Version 4.0. Der GCC 9 unterstützt inzwischen teilweise OpenMP 5.0.

openmp-vectors.cpp

```
template<typename T>
void axpy(std::size_t n, T alpha, const T* x, std::ptrdiff_t incX,
      T* y, std::ptrdiff_t incY) {
#pragma omp parallel for
   for (std::size_t i = 0; i < n; ++i) {
      y[i*incY] += alpha * x[i*incX];
   }
}
```

- Im Unterschied zur vorherigen Fassung der *axpy*-Funktion wurde die **for**-Schleife vereinfacht (nur eine Schleifenvariable) und es wurde darauf verzichtet, die Zeiger x und y zu verändern.
- Alle für OpenMP bestimmten Pragmas beginnen mit **#pragma** omp, wonach die eigentliche OpenMP-Anweisung folgt. Hier bittet **parallel for** um die Parallelisierung der nachfolgenden **for**-Schleife.
- Die Schleifenvariable ist für jeden implizit erzeugten Thread privat und alle anderen Variablen werden in der Voreinstellung gemeinsam verwendet.

Makefile

```
Sources :=      $(wildcard *.cpp)
Objects :=      $(patsubst %.cpp,%.o,$(Sources))
Targets :=      $(patsubst %.cpp,%,$(Sources))
CXX :=          g++
CXXFLAGS :=     -std=gnu++11 -Ofast -fopenmp
CC :=           g++
LDFLAGS :=      -fopenmp
.PHONY:         all clean
all:            $(Targets)
clean:          ; rm -f $(Objects) $(Targets)
```

- Die GNU Compiler Collection (GCC) unterstützt OpenMP für Fortran, C und C++ ab der Version 4.2, wenn die Option „-fopenmp“ spezifiziert wird.
- Der C++-Compiler von Sun berücksichtigt OpenMP-Pragmas, wenn die Option „-xopenmp“ angegeben wird.
- Diese Optionen sind auch jeweils beim Binden anzugeben, damit die zugehörigen Laufzeitbibliotheken mit eingebunden werden.

```
theon$ time env OMP_NUM_THREADS=1 openmp-vectors 100000000
time in ms: 145.496

real 0m2.223s
user 0m0.707s
sys 0m1.091s
theon$ time env OMP_NUM_THREADS=4 openmp-vectors 100000000
time in ms: 58.2448

real 0m1.127s
user 0m0.947s
sys 0m1.593s
theon$ cd ../threads-vectors
theon$ time vectors 100000000 4

real 0m1.519s
user 0m0.875s
sys 0m1.641s
theon$
```

- Mit der Umgebungsvariablen *OMP_NUM_THREADS* lässt sich festlegen, wieviele Threads insgesamt durch OpenMP erzeugt werden dürfen.

- Zu parallelisierende Schleifen müssen bei OpenMP grundsätzlich einer der folgenden Formen entsprechen:

$$\textbf{for}\ (index = start;\ index \left\{ \begin{array}{c} < \\ <= \\ >= \\ > \end{array} \right\} end; \left\{ \begin{array}{l} index{+}{+} \\ {+}{+}index \\ index{-}{-} \\ {-}{-}index \\ index \mathrel{+}= inc \\ index \mathrel{-}= inc \\ index = index + inc \\ index = inc + index \\ index = index - inc \end{array} \right\})$$

- Die Schleifenvariable darf dabei auch innerhalb der **for**-Schleife deklariert werden.

openmp-vectors.cpp

```
template<typename T>
void axpy(std::size_t n, T alpha, const T* x, std::ptrdiff_t incX,
      T* y, std::ptrdiff_t incY) {
#pragma omp parallel for
   for (std::size_t i = 0; i < n; ++i) {
      y[i*incY] += alpha * x[i*incX];
   }
}
```

- Per Voreinstellung ist nur die Schleifenvariable privat für jeden Thread.
- Alle anderen Variablen werden von allen Threads gemeinsam verwendet, ohne dass dabei eine Synchronisierung implizit erfolgt. Deswegen sollten gemeinsame Variable nur lesenderweise verwendet werden (wie etwa bei *alpha*) oder die Schreibzugriffe sollten sich nicht ins Gehege kommen (wie etwa bei *y*).
- Abhängigkeiten von vorherigen Schleifendurchläufen müssen entfernt werden. Dies betrifft insbesondere weitere Schleifenvariablen oder Zeiger, die fortlaufend verschoben werden.
- Somit muss jeder Schleifendurchlauf unabhängig berechnet werden.

simpson4.cpp

```
template<typename T, typename F>
T simpson(F&& f, T a, T b, std::size_t n) {
   assert(n > 0 && a <= b);
   T value = f(a)/2 + f(b)/2;
   T xleft;
   T x = a;
   for (std::size_t i = 1; i < n; ++i) {
      xleft = x; x = a + i * (b - a) / n;
      value += f(x) + 2 * f((xleft + x)/2);
   }
   value += 2 * f((x + b)/2);
   value *= (b - a) / n / 3;
   return value;
}
```

- *xleft* und *x* sollten für jeden Thread privat sein.
- Die Variable *xleft* wird in Abhängigkeit des vorherigen Schleifendurchlaufs festgelegt.
- Die Variable *value* wird unsynchronisiert inkrementiert.

omp-simpson.cpp

```
template<typename F, typename T>
T simpson(F&& f, T a, T b, std::size_t n) {
   assert(n > T() && a <= b);
   T xleft;
   T x = a;
   T sum{};
#pragma omp parallel for \
      private(xleft) \
      lastprivate(x) \
      reduction(+:sum)
   for (std::size_t i = 1; i < n; ++i) {
      xleft = a + (i-1) * (b - a) / n;
      x = a + i * (b - a) / n;
      sum += f(x) + 2 * f((xleft + x)/2);
   }
   T value = f(a)/2 + f(b)/2 + sum + 2 * f((x + b)/2);
   value *= (b - a) / n / 3;
   return value;
}
```

- Einem OpenMP-Parallelisierungs-Pragma können diverse Klauseln folgen, die insbesondere die Behandlung der Variablen regeln.
- Mit **private**(*xleft*) wird die Variable *xleft* privat für jeden Thread gehalten. Die private Variable ist zu Beginn undefiniert. Das gilt auch dann, wenn sie zuvor initialisiert war.
- *lastprivate*(x) ist ebenfalls ähnlich zu **private**(x), aber der Haupt-Thread übernimmt nach der Parallelisierung den Wert, der beim letzten Schleifendurchlauf bestimmt wurde.
- Mit *reduction*($+$:*sum*) wird *sum* zu einer auf 0 initialisierten privaten Variable, wobei am Ende der Parallelisierung alle von den einzelnen Threads berechneten *sum*-Werte aufsummiert und in die entsprechende Variable des Haupt-Threads übernommen werden.
- Ferner gibt es noch *firstprivate*, das ähnlich ist zu **private**, abgesehen davon, dass zu Beginn der Wert des Haupt-Threads übernommen wird.

omp-simpson-explicit.cpp

```
template<typename T, typename F>
T mt_simpson(F&& f, T a, T b, std::size_t n) {
   assert(n > 0 && a <= b);
   T sum{};
#pragma omp parallel reduction(+:sum)
   {
      auto nofthreads = omp_get_num_threads();
      auto nofintervals = n / nofthreads;
      auto remainder = n % nofthreads;
      auto i = omp_get_thread_num();
      auto interval = nofintervals * i;
      auto intervals = nofintervals;
      if (i < remainder) {
         ++intervals;
         interval += i;
      } else {
         interval += remainder;
      }
      auto xleft = a + interval * (b - a) / n;
      auto x = a + (interval + intervals) * (b - a) / n;
      sum += simpson(f, xleft, x, intervals);
   }
   return sum;
}
```

- Grundsätzlich ist es auch möglich, die Parallelisierung explizit zu kontrollieren.
- In diesem Beispiel entspricht die Funktion *simpson* wieder der nicht-parallelisierten Variante.
- Mit **#pragma** omp parallel wird die folgende Anweisung entsprechend dem Fork-And-Join-Pattern parallelisiert.
- Als Anweisung wird sinnvollerweise ein eigenständiger Block verwendet. Alle darin lokal deklarierten Variablen sind damit auch automatisch lokal zu den einzelnen Threads.
- Die Funktion *omp_get_num_threads* liefert die Zahl der aktiven Threads zurück und *omp_get_thread_num* die Nummer des aktuellen Threads (wird von 0 an gezählt). Aufgrund dieser beiden Werte kann wie gehabt die Aufteilung erfolgen.

- Die bekanntesten numerischen Verfahren zum Auffinden von Nullstellen wie etwa die Bisektion und die Regula Falsi arbeiten nur lokal, d.h. für eine stetige Funktion f wird ein Intervall $[a, b]$ benötigt, für das gilt $f(a) * f(b) < 0$.
- Für das globale Auffinden aller einfachen Nullstellen in einem Intervall (a, b) erweist sich der von Plagianakos et al vorgestellte Ansatz als recht nützlich: V. P. Plagianakos et al: *Locating and computing in parallel all the simple roots of special functions using PVM*, Journal of Computational and Applied Mathematics 133 (2001) 545–554
- Dieses Verfahren lässt sich parallelisieren. Prinzipiell kann das Gesamtintervall auf die einzelnen Threads aufgeteilt werden. Da sich jedoch die Nullstellen nicht notwendigerweise gleichmäßig verteilen, lohnt sich ein dynamischer Ansatz, bei dem Aufträge erzeugt und bearbeitet werden.

- Gegeben sei die zweimal stetig differenzierbare Funktion $f : [a, b] \in \mathbb{R} \to \mathbb{R}$.
- Dann lässt sich die Zahl der einfachen Nullstellen $N_{f,a,b}$ der Funktion f auf dem Intervall (a, b) folgendermaßen bestimmen:

$$N_{f,a,b} = -\frac{1}{\pi}\left[\int_a^b \frac{f(x)f''(x) - f'^2(x)}{f^2(x) + f'^2(x)}\,dx - \arctan\left(\frac{f'(b)}{f(b)}\right) + \arctan\left(\frac{f'(a)}{f(a)}\right)\right]$$

- Da das Resultat eine ganze Zahl ist, lässt sich das Integral numerisch recht leicht berechnen, weil nur wenige Schritte notwendig sind, um die notwendige Genauigkeit zu erreichen.

- Um alle Nullstellen zu finden, wird die Zahl der Nullstellen auf dem Intervall (a, b) ermittelt.
- Wenn sie 0 ist, kann die weitere Suche abgebrochen werden.
- Wenn sie genau 1 ist, dann kann eines der traditionellen Verfahren eingesetzt werden.
- Bei größeren Werten kann das Intervall per Bisektion aufgeteilt werden. Auf jedem der Teilintervalle wird dann rekursiv die gleiche Prozedur angewandt nach dem Teile- und Herrsche-Prinzip.

rootfinder.hpp

```
void get_roots(Real a, Real b, Real eps, unsigned int numOfRoots,
          RootVector* roots) const {
   if (numOfRoots == 0) return;
   if (numOfRoots == 1) {
      roots->push_back(bisection(a, b, eps)); return;
   }
   Real midpoint = (a + b) / 2;
   unsigned int numOfLeftRoots = get_count(a, midpoint);
   unsigned int numOfRightRoots = get_count(midpoint, b);
   if (numOfLeftRoots + numOfRightRoots < numOfRoots) {
      roots->push_back(midpoint);
   }
   get_roots(a, midpoint, eps, numOfLeftRoots, roots);
   get_roots(midpoint, b, eps, numOfRightRoots, roots);
}
```

- Wenn keine Parallelisierung zur Verfügung steht, wird ein Teile- und Herrsche-Problem typischerweise rekursiv gelöst.

prootfinder.hpp

```
struct Task {
   Task(Real a, Real b, unsigned int numOfRoots) :
      a(a), b(b), numOfRoots(numOfRoots) {
   }
   Task() : a(0), b(0), numOfRoots(0) {
   }
   Real a, b;
   unsigned int numOfRoots;
};
```

- Wenn bei einer Parallelisierung zu Beginn keine sinnvolle Aufteilung durchgeführt werden kann, ist es sinnvoll, Aufträge in Datenstrukturen zu verpacken und diese in einer Warteschlange zu verwalten.
- Dann können Aufträge auch während der Abarbeitung eines Auftrags neu erzeugt und an die Wartschlange angehängt werden.

prootfinder.hpp

```
   // now we are working on task
   if (task.numOfRoots == 0) continue;
   if (task.numOfRoots == 1) {
      Real root = bisection(task.a, task.b, eps);
#pragma omp critical
      roots->push_back(root); continue;
   }
   Real midpoint = (task.a + task.b) / 2;
   unsigned int numOfLeftRoots = get_count(task.a, midpoint);
   unsigned int numOfRightRoots = get_count(midpoint, task.b);
   if (numOfLeftRoots + numOfRightRoots < task.numOfRoots) {
#pragma omp critical
      roots->push_back(midpoint);
   }
#pragma omp critical
   {
      tasks.push_back(Task(task.a, midpoint, numOfLeftRoots));
      tasks.push_back(Task(midpoint, task.b, numOfRightRoots));
   }
```

prootfinder.hpp

```
#pragma omp critical
   {
      tasks.push_back(Task(task.a, midpoint, numOfLeftRoots));
      tasks.push_back(Task(midpoint, task.b, numOfRightRoots));
   }
```

- OpenMP unterstützt kritische Regionen.
- Zu einem gegebenen Zeitpunkt kann sich nur ein Thread in einer kritischen Region befinden.
- Bei der Pragma-Instruktion **#pragma** omp critical zählt die folgende Anweisung als kritische Region.
- Optional kann bei der Pragma-Instruktion in Klammern die kritische Region benannt werden. Dann kann sich maximal nur ein Thread in einer kritischen Region dieses Namens befinden.

prootfinder.hpp

```
void get_roots(Real a, Real b, Real eps, unsigned int numOfRoots,
          RootVector* roots) const {
   std::list<Task> tasks; // shared

   if (numOfRoots == 0) return;
   tasks.push_back(Task(a, b, numOfRoots));
#pragma omp parallel
   for(;;) {
#pragma omp critical
      if (tasks.size() > 0) {
         task = tasks.front(); tasks.pop_front();
      } else {
         break;
      }
      // process task and possibly generate new tasks
   }
}
```

- Wenn als Abbruchkriterium eine leere Auftragsschlange genommen wird, besteht das Risiko, dass sich einzelne Threads verabschieden, obwohl andere Threads noch neue Aufträge erzeugen könnten.

prootfinder.hpp

```
void get_roots(Real a, Real b, Real eps, unsigned int numOfRoots,
          RootVector* roots) const {
   std::list<Task> tasks; // shared
   if (numOfRoots == 0) return;
   tasks.push_back(Task(a, b, numOfRoots));
#pragma omp parallel
   while (roots->size() < numOfRoots) {
      // fetch next task, if there is any
      // ...

      // now we are working on task
      // ...
   }
}
```

- Alternativ bietet es sich an, die einzelnen Threads erst dann zu beenden, wenn das Gesamtproblem gelöst ist.
- Aber was können die einzelnen Threads dann tun, wenn das Gesamtproblem noch ungelöst ist und es zur Zeit keinen Auftrag gibt?

Lösung mit einer *busy loop*

prootfinder.hpp

```
#pragma omp parallel
   while (roots->size() < numOfRoots) {
      // fetch next task, if there is any
      Task task;
      bool busyloop = false;
#pragma omp critical
      if (tasks.size() > 0) {
         task = tasks.front(); tasks.pop_front();
      } else {
         busyloop = true;
      }
      if (busyloop) {
         continue;
      }

      // now we are working on task
      // ...
   }
```

- Diese Lösung geht in eine rechenintensive Warteschleife, bis ein Auftrag erscheint.

Nachteil einer *busy loop*

- Eine rechenintensive Warteschleife (*busy loop*) nimmt eine Recheneinheit sinnlos ein, ohne dabei etwas Nützliches zu tun.
- Bei Maschinen mit anderen Nutzern oder mehr Threads als zur Verfügung stehenden Recheneinheiten, ist dies sehr unerfreulich.
- Eine Lösung wären Bedingungsvariablen. Aber diese werden von OpenMP nicht unterstützt.
- Eine Lösung zu diesem Problem kam erst mit der Einführung von OpenMP 4.0.

- Beginnend mit OpenMP 4.0 sind einige Erweiterungen hinzugekommen, die auch die Unterstützung des Master/Worker-Patterns vorsehen.
- Um das Master/Worker-Pattern umzusetzen, wird normalerweise **#pragma** omp parallel unmittelbar mit **#pragma** omp single kombiniert, d.h. die nachfolgende Anweisung oder der nachfolgende Block wird nur von einem Thread ausgeführt (dem Master), während die anderen Threads (die Worker) auf Aufträge warten.
- Mit Hilfe von **#pragma** omp task können dann einzelne Anweisungen oder Blöcke an einen Worker delegiert werden. Dies kann in beliebiger dynamischer und rekursiver Form erfolgen. Insbesondere dürfen auch die Worker selbst diese Direktive verwenden und damit neue Aufträge erzeugen.
- Am Ende des **#pragma** omp parallel-Blocks findet implizit eine Synchronisierung statt. Lokale Synchronisierungsblöcke, auch innerhalb eines Worker-Prozesses, sind mit **#pragma** omp parallel möglich.

prootfinder.hpp

```
template<typename OutputIterator>
void get_roots(OutputIterator outit,
      Real a, Real b, Real eps) const {
   unsigned int numOfRoots = get_count(a, b);
   if (numOfRoots > 0) {
      #pragma omp parallel
      #pragma omp single
      get_roots(outit, a, b, eps, numOfRoots);
   }
}
```

- Zu Beginn der Rekursion wird mit **#pragma** omp parallel die Parallelisierung eröffnet, wobei wegen **#pragma** omp single zunächst nur der Master-Thread beginnt und die anderen noch warten – entweder auf das Ende des Blocks oder die Vergebung von Aufträgen.

prootfinder.hpp

```
template<typename OutputIterator>
void get_roots(OutputIterator& outit,
          Real a, Real b, Real eps, unsigned int numOfRoots) const {
   unsigned int numOfRootsFound = 0; // shared
   if (numOfRoots == 0) return;
   if (numOfRoots == 1) {
      Real root = bisection(a, b, eps);
      #pragma omp critical
      { *outit++ = root; ++numOfRootsFound; }
   } else {
      // more than one root in the remaining interval
      // ...
   }
}
```

- In dieser Implementierung ist die private *get_roots*-Methode rekursiv. Sie wird zu Beginn vom Master-Thread aufgerufen, danach aber auch von den Workern.
- Eine explizite Verwaltung der Aufträge findet nicht mehr statt. Kritische Regionen werden daher nur noch für den Output-Iterator benötigt.

prootfinder.hpp

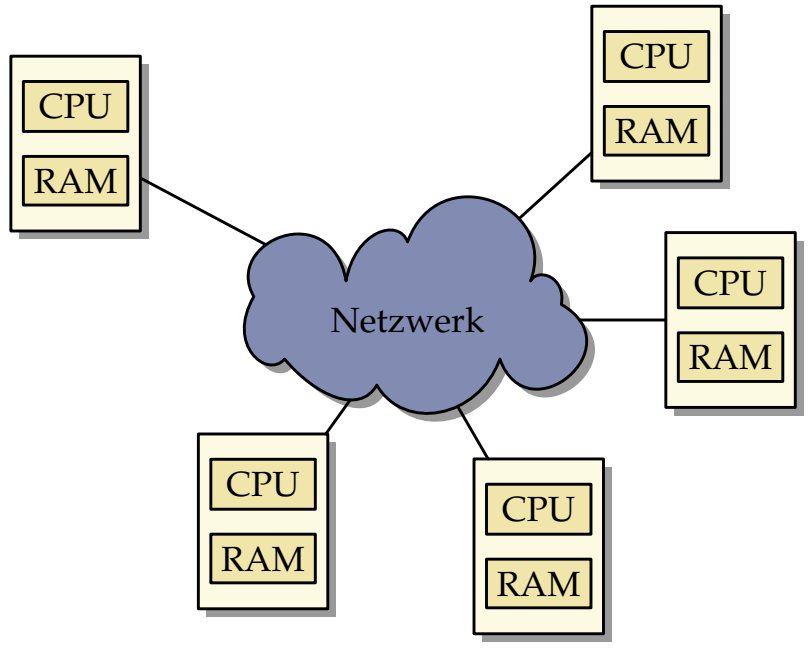
```
// more than one root in the remaining interval
Real midpoint = (a + b) / 2;
unsigned int numOfLeftRoots = 0; // shared
unsigned int numOfRightRoots = 0; // shared
#pragma omp taskgroup
{
   #pragma omp task shared(numOfLeftRoots)
   numOfLeftRoots = get_count(a, midpoint);
   #pragma omp task shared(numOfRightRoots)
   numOfRightRoots = get_count(midpoint, b);
}
// divide and conquer
// ...
```

- Zu Beginn muss die Zahl der Nullstellen im linken und im rechten Teilintervall ermittelt werden.
- Die entsprechenden Aufträge werden hier separat vergeben und durch **#pragma** omp taskgroup synchronisieren wir uns mit der Fertigstellung der beiden Threads gemäß dem Fork-and-Join-Pattern.

prootfinder.hpp

```
if (numOfLeftRoots + numOfRightRoots < numOfRoots) {
   #pragma omp critical
   {
      *outit++ = midpoint; ++numOfRootsFound;
   }
}
#pragma omp task shared(outit)
get_roots(outit, a, midpoint, eps, numOfLeftRoots);
#pragma omp task shared(outit)
get_roots(outit, midpoint, b, eps, numOfRightRoots);
```

- Gemäß dem Teile- und Herrsche-Prinzip wird hier die Aufgabe rekursiv aufgeteilt in die beiden Teilintervalle.
- Alle Variablen, auf die ein **#pragma** omp task erzeugter Auftrag zugreift, sind lokale Kopien – es sei denn, die Variablen werden mit *shared* deklariert. Bei allen Referenzen (wie hier *outit*) ist dies zwingend erforderlich.



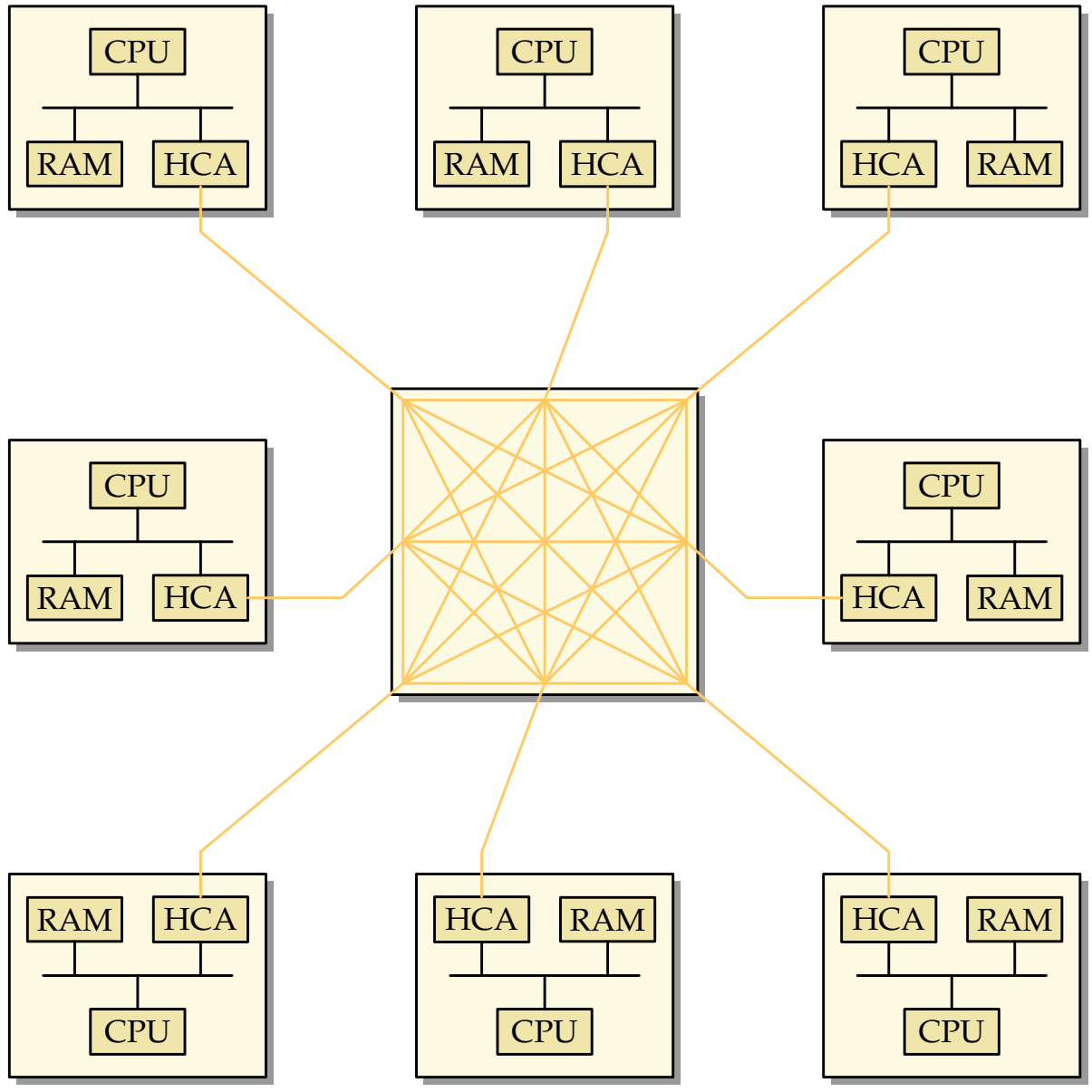
- Multicomputer bestehen aus einzelnen Rechnern mit eigenem Speicher, die über ein Netzwerk miteinander verbunden sind.
- Ein direkter Zugriff auf fremden Speicher ist nicht möglich.
- Die Kommunikation kann daher nicht über gemeinsame Speicherbereiche erfolgen. Stattdessen geschieht dies durch den Austausch von Daten über das Netzwerk.

- Eine traditionelle Vernetzung einzelner unabhängiger Maschinen über Ethernet und der Verwendung von TCP/IP-Sockets erscheint naheliegend.
- Der Vorteil ist die kostengünstige Realisierung, da die bereits vorhandene Infrastruktur genutzt wird und zahlreiche Ressourcen zeitweise ungenutzt sind (wie etwa Pools mit Desktop-Maschinen).
- Zu den Nachteilen gehört
  - die hohe Latenzzeit (ca. 150μs bei GbE auf Pacioli, ca. 500μs über das Uni-Netzwerk),
  - die vergleichsweise niedrige Bandbreite,
  - das Fehlen einer garantierten Bandbreite und
  - die Fehleranfälligkeit (wird von TCP/IP automatisch korrigiert, kostet aber Zeit).
  - Ferner fehlt die Skalierbarkeit, wenn nicht erheblich mehr in die Netzwerkinfrastruktur investiert wird.

- Mehrere Hersteller schlossen sich 1999 zusammen, um gemeinsam einen Standard zu entwickeln für Netzwerke mit höheren Bandbreiten und niedrigeren Latenzzeiten.
- Infiniband ist heute eine der populärsten Vernetzungen bei Supercomputern: 178 der TOP-500 verwenden Infiniband (Stand: Juni 2017).
- Die Latenzzeiten liegen im Bereich von 140 *ns* bis 2,6 μs.
- Brutto-Bandbreiten sind zur Zeit bis ca. 56 Gb/s möglich. (Bei Pacioli: brutto 2 Gb/s, netto mit MPI knapp 1 Gb/s.)
- Nachteile:
  - Keine hierarchischen Netzwerkstrukturen und damit eine Begrenzung der maximalen Rechnerzahl,
  - alles muss räumlich sehr eng zueinander stehen,
  - sehr hohe Kosten insbesondere dann, wenn viele Rechner auf diese Weise zu verbinden sind.

- Bei einer Vernetzung über Infiniband gibt es einen zentralen Switch, an dem alle beteiligten Rechner angeschlossen sind.
- Jede der Rechner benötigt einen speziellen HCA (*Host Channel Adapter*), der direkten Zugang zum Hauptspeicher besitzt.
- Zwischen den HCAs und dem Switch wird normalerweise Kupfer verwendet. Die maximale Länge beträgt hier 14 Meter. Mit optischen Kabeln und entsprechenden Adaptern können auch Längen bis zu ca. 100 Meter erreicht werden.
- Zwischen einem Rechner und dem Switch können auch mehrere Verbindungen bestehen zur Erhöhung der Bandbreite.
- Die zur Zeit auf dem Markt angebotenen InfiniBand-Switches bieten zwischen 8 und 864 Ports.

Topologie bei InfiniBand

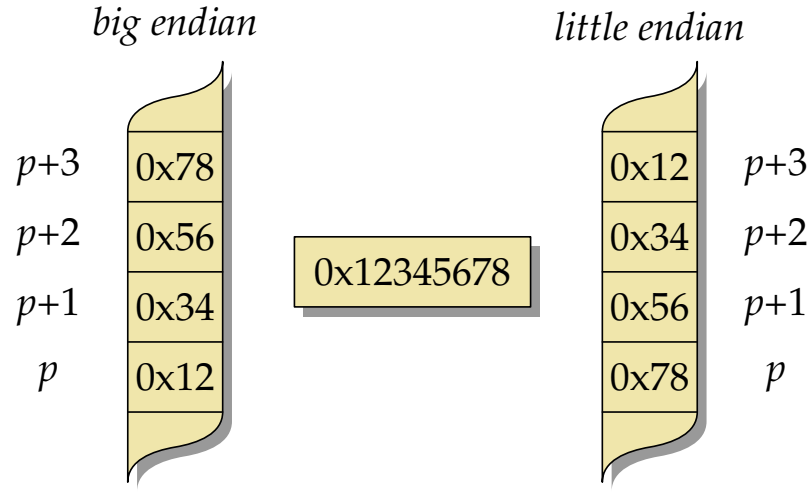


Die extrem niedrigen Latenzzeiten werden bei InfiniBand nur durch spezielle Techniken erreicht:

- Die HCAs haben direkten Zugang zum Hauptspeicher, d.h. ohne Intervention des Betriebssystems kann der Speicher ausgelesen oder beschrieben werden. Die HCAs können dabei auch selbständig virtuelle in physische Adressen umwandeln.
- Es findet kein Routing statt. Der Switch hat eine separate Verbindungsleitung für jede beliebige Anschlusskombination. Damit steht in jedem Falle die volle Bandbreite ungeteilt zur Verfügung. Die Latenzzeiten innerhalb eines Switch-Chips können bei 200 Nanosekunden liegen, von Port zu Port werden beim 648-Port-Switch von Mellanox nach Herstellerangaben Latenzzeiten von 100-300 Nanosekunden erreicht.

Auf Pacioli wurden auf Programmebene (mit MPI) Latenzzeiten von unter 5 μs erreicht.

- Da einzelne Rechner unterschiedlichen Architekturen angehören können, werden möglicherweise einige Datentypen (etwa ganze Zahlen oder Gleitkommazahlen) unterschiedlich binär repräsentiert.
- Wenn die Daten mit Typinformationen versehen werden, dann wird die Gefahr von Fehlinterpretationen vermieden.
- Die Übertragung von Daten gibt auch die Gelegenheit, die Struktur umzuorganisieren. Beispielsweise kann ein Spaltenvektor in einen Zeilenvektor konvertiert werden.
- Auch die Übertragung dynamischer Datenstrukturen ist möglich. Dann müssen Zeiger in Referenzen umgesetzt werden.
- Die Technik des Verpackens und Auspackens von Datenstrukturen in Byte-Sequenzen, die sich übertragen lassen, wird Serialisierung oder *marshalling* genannt.



- Bei *little endian* sind im ersten Byte die niedrigstwertigen Bits (hier 0*x78*).
- Die Reihenfolge ist bei *big endian* genau umgekehrt, d.h. die höchstwertigen Bits kommen zuerst (hier 0*x12*).
- Da der Zugriff immer byte-weise erfolgt, interessiert uns nur die Reihenfolge der Bytes, nicht der Bits.
- Zu den Plattformen mit *little endian* gehört die x86-Architektur von Intel, während die SPARC-Architektur normalerweise mit *big endian* operiert.

- Prinzipiell hat sich als Repräsentierung das Zweier-Komplement durchgesetzt:

$$a = \sum_{i=1}^{n-1} a_i 2^{i-1} - a_n 2^n$$

- Wertebereich: $\left[-2^{n-1}, 2^{n-1} - 1\right]$
- Dann bleibt nur noch die Entscheidung über die Größe von n und die Reihenfolge der einzelnen Bytes.
- Bei letzterem wird traditionell *big endian* gewählt (*network byte order*), siehe RFC 791, *Appendix B*, und RFC 1700, *Data Notations*.

- Durch die *Google Protocol Buffers* wurde eine alternative Repräsentierung populär, bei der ganze Zahlen mit einer variablen Anzahl von Bytes dargestellt werden.
- Von den acht Bits wird das höchstwertige nur als Hinweis verwendet, ob die Folge fortgesetzt wird oder nicht: 1 = Folge wird fortgesetzt, 0 = Folge ist mit diesem Byte beendet.
- Die anderen sieben Bits (deswegen zur Basis 128) werden als Inhalt genommen, wobei sich Google für *little endian* entschied, d.h. die niedrigstwertigen Bits kommen zuerst.
- Dieses Verfahren ist jedoch ungünstig für negative Zahlen im Zweierkomplement, da dann für die -1 die maximale Länge zur Kodierung verwendet werden muss.
- Beispiel: 300 wird kodiert als Folge der beiden Bytes 0xac und 0x02 (binär: 1010 1100 0000 0010).

- Bei vorzeichenbehafteten ganzen Zahlen verwenden die *Google Protocol Buffers* die sogenannte Zickzack-Kodierung, die jeder ganzen Zahl eine nicht-negative Zahl zuordnet:

ganze Zahl	Zickzack-Kodierung
0	0
-1	1
1	2
-2	3
2147483647	4294967294
-2147483648	4294967295

- Das bedeutet, dass das höchst- und das niedrigstwertige Bit jeweils vertauscht worden sind. Bei 32-Bit-Zahlen sieht das dann so aus:

  $(n << 1) \hat{} (n >> 31)$

- IEEE-754 (auch IEC 60559 genannt) hat sich als Standard für die Repräsentierung von Gleitkommazahlen durchgesetzt.
- Eine Gleitkommazahl nach IEEE-754 besteht aus drei Komponenten:
  - dem Vorzeichen s (ein Bit),
  - dem aus q Bits bestehenden Exponenten $\{e_i\}_{i=1}^{q}$,
  - und der aus p Bits bestehenden Mantisse $\{m_i\}_{i=1}^{p}$.
- Für **float** und **double** ist die Konfiguration durch den Standard festgelegt, bei **long double** ist keine Portabilität gegeben:

Datentyp	Bits	q	p
float	32	8	23
double	64	11	52
long double		≥ 15	≥ 63

- Festzulegen ist hier nur, ob die Binärrepräsentierung in *little* oder *big endian* übertragen wird.

- Kommunikation ist entweder bilateral (der Normalfall) oder richtet sich an viele Teilnehmer gleichzeitig (*multicast*, *broadcast*).
- Bei einer bilateralen Kommunikation ergibt sich aus der Asymmetrie der Verbindungsaufnahme eine Rollenverteilung, typischerweise die eines Klienten und die eines Diensteanbieters.
- Diese Rollenverteilung bezieht sich immer auf eine konkrete Verbindung, d.h. es können zwischen zwei Kommunikationspartnern mehrere Verbindungen mit unterschiedlichen Rollenverteilungen bestehen.
- Über ein Protokoll wird geregelt, wie die einzelnen Mitteilungen aussehen und in welcher Abfolge diese gesendet werden dürfen.

- Klassischerweise existieren Netzwerkdienste, die angerufen werden können:
  - Diese sind sinnvoll, wenn die gleichen Aufgaben wiederkehrend zu lösen sind.
  - Es bleibt aber das Problem, wie diese Dienste gefunden werden und wie eine sinnvolle Lastverteilung zwischen konkurrierenden Anwendungen erfolgt.
- Bei variierenden Aufgabenstellungen muss ggf. eine Anwendung erst auf genügend Rechnerresourcen verteilt werden:
  - Wie erfolgt die Verteilung?
  - Wird die Umfang der Ressourcen zu Beginn oder erst im Laufe der Anwendung festgelegt?
  - Wie erfolgt die Verbindungsaufnahme untereinander?

- MPI (*Message Passing Interface*) ist ein Standard für eine Bibliotheksschnittstelle für parallele Programme.
- 1994 entstand die erste Fassung (1.0), 1995 die Version 1.2 und seit 1997 gibt es 2.0. Im September 2012 erschien die Version 3.0 und im Juni 2015 kam 3.1, die bis heute aktuell ist, 4.0 ist zur Zeit in Vorbereitung. Diese wird von OpenMPI auf Theon unterstützt. Unsere Debian-Installationen unterstützen MPI 3.0. Die Standards sind öffentlich unter `http://www.mpi-forum.org/`.
- Der Standard umfasst die sprachspezifischen Schnittstellen für Fortran und C. (Es wird die C-Schnittstelle in C++ verwendet. Alternativ bietet sich die Boost-Library an: `https://www.boost.org/doc/libs/1_70_0/doc/html/mpi.html`).
- Es stehen mehrere Open-Source-Implementierungen zur Verfügung:
  - OpenMPI: `http://www.open-mpi.org/` (bei uns überall installiert)
  - MPICH: `http://www.mpich.org/`
  - MVAPICH: `http://mvapich.cse.ohio-state.edu/` (spezialisiert auf Infiniband)

- Zu Beginn wird mit n die Zahl der Prozesse festgelegt.
- Jeder Prozess läuft in seinem eigenen Adressraum und hat innerhalb von MPI eine eigene Nummer (*rank*) im Bereich von 0 bis $n-1$.
- Die Kommunikation mit den anderen Prozessen erfolgt über Nachrichten, die entweder an alle gerichtet werden (*broadcast*), an Prozessgruppen (*multicast*) oder individuell versandt werden.
- Die Kommunikation kann sowohl synchron als auch asynchron erfolgen.
- Die Prozesse können in einzelne Gruppen aufgesplittet werden. Ein Prozess kann mehreren Gruppen angehören. Alle Prozesse gehören der globalen Gruppe an.

Simpson-Verfahren mit MPI

mpi-simpson.cpp

```
int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);

   int nofprocesses; MPI_Comm_size(MPI_COMM_WORLD, &nofprocesses);
   int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   // process command line arguments
   int n; // number of intervals
   if (rank == 0) {
      cmdname = argv[0];
      if (argc > 2) usage();
      if (argc == 1) {
         n = 500;
      } else {
         std::istringstream arg(argv[1]);
         if (!(arg >> n) || n <= 0) usage();
      }
   }
   // ...

   MPI_Finalize();

   if (rank == 0) {
      std::cout << std::setprecision(14) << sum << std::endl;
   }
}
```

mpi-simpson.cpp

```
MPI_Init(&argc, &argv);

int nofprocesses; MPI_Comm_size(MPI_COMM_WORLD, &nofprocesses);
int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

- Im Normalfall starten alle Prozesse das gleiche Programm und beginnen alle mit *main*(). (Es ist auch möglich, verschiedene Programme über MPI zu koordinieren.)
- Erst nach dem Aufruf von *MPI_Init*() sind weitere MPI-Operationen zulässig.
- *MPI_COMM_WORLD* ist die globale Gesamtgruppe aller Prozesse eines MPI-Laufs.
- Die Funktionen *MPI_Comm_size* und *MPI_Comm_rank* liefern die Zahl der Prozesse bzw. die eigene Nummer innerhalb der Gruppe (immer ab 0 und konsekutiv weiterzählend).

mpi-simpson.cpp

```
// process command line arguments
int n; // number of intervals
if (rank == 0) {
   cmdname = argv[0];
   if (argc > 2) usage();
   if (argc == 1) {
      n = 500;
   } else {
      std::istringstream arg(argv[1]);
      if (!(arg >> n) || n <= 0) usage();
   }
}
```

- Der Hauptprozess hat den *rank* 0. Nur dieser sollte verwendet werden, um Kommandozeilenargumente auszuwerten und/oder Ein- und Ausgabe zu betreiben.

mpi-simpson.cpp

```
// broadcast number of intervals
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

- Mit der Funktion *MPI_Bcast* kann eine Nachricht an alle Mitglieder einer Gruppe versandt werden.
- Die Funktion bezieht sich auf eine Gruppe, wobei *MPI_COMM_WORLD* die globale Gesamtgruppe repräsentiert.
- Der erste Parameter ist ein Zeiger auf das erste zu übermittelnde Objekt. Der zweite Parameter nennt die Zahl der zu übermittelnden Objekte (hier nur 1).
- Der dritte Parameter spezifiziert den Datentyp eines zu übermittelnden Elements. Hier wird *MPI_INT* verwendet, das dem Datentyp **int** entspricht.
- Der vorletzte Parameter legt fest, welcher Prozess den Broadcast verschickt. Alle anderen Prozesse, die den Aufruf ausführen, empfangen das Paket.

mpi-simpson.cpp

```
// broadcast number of intervals
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

double value = 0; // summed up value of our intervals;
if (rank < n) {
   int nofintervals = n / nofprocesses;
   int remainder = n % nofprocesses;
   int first_interval = rank * nofintervals;
   if (rank < remainder) {
      ++nofintervals;
      if (rank > 0) first_interval += rank;
   } else {
      first_interval += remainder;
   }
   int next_interval = first_interval + nofintervals;

   double xleft = a + first_interval * (b - a) / n;
   double x = a + next_interval * (b - a) / n;
   value = simpson([](double x) -> double {
      return 4 / (1 + x*x);
   }, xleft, x, nofintervals);
}
double sum;
MPI_Reduce(&value, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

mpi-simpson.cpp

```
  double sum;
  MPI_Reduce(&value, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

- Mit der Funktion *MPI_Reduce* werden die einzelnen Ergebnisse aller Prozesse (einschließlich dem auswertenden Prozess) eingesammelt und dann mit einer auszuwählenden Funktion aggregiert.
- Der erste Parameter ist ein Zeiger auf ein Einzelresultat. Der zweite Parameter verweist auf die Variable, wo der aggregierte Wert abzulegen ist.
- Der dritte Parameter liegt wieder die Zahl der Elemente fest (hier 1) und der vierte den Datentyp (hier *MPI_DOUBLE* für **double**).
- Der fünfte Parameter spezifiziert die aggregierende Funktion (hier *MPI_SUM* zum Aufsummieren) und der sechste Parameter gibt an, welcher Prozess den aggregierten Wert erhält.

MPI unterstützt folgende Datentypen von C++:

MPI_CHAR	**char**
MPI_SIGNED_CHAR	**signed char**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_SHORT	**signed short**
MPI_INT	**signed int**
MPI_LONG	**signed long**
MPI_LONG_LONG	**signed long long int**
MPI_UNSIGNED_SHORT	**unsigned short**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned long**
MPI_UNSIGNED_LONG_LONG	**unsigned long long int**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**
MPI_LONG_DOUBLE	**long double**
MPI_WCHAR	**wchar_t**
MPI_CXX_BOOL	**bool**
MPI_CXX_FLOAT_COMPLEX	*std*::*complex*<**float**>
MPI_CXX_DOUBLE_COMPLEX	*std*::*complex*<**double**>
MPI_CXX_LONG_DOUBLE_COMPLEX	*std*::*complex*<**long double**>

Übersetzen von MPI-Anwendungen auf unseren Maschinen

Makefile

```
CC :=           mpicc
CXX :=          mpic++
CXXFLAGS :=     -g -Ofast -std=c++11
CPPFLAGS :=     -std=c++11
LDFLAGS :=      -std=c++11
```

- Wir verwenden OpenMPI auf unseren Rechnern.
- Statt den Übersetzern *g++* (und ggf. *gcc*) sind *mpic++* und *mpicc* zu verwenden.
- Dann sind alle MPI-spezifischen Header-Dateien und Bibliotheken automatisch zugänglich.
- Die Option -*Ofast* schaltet alle Optimierungen ein.

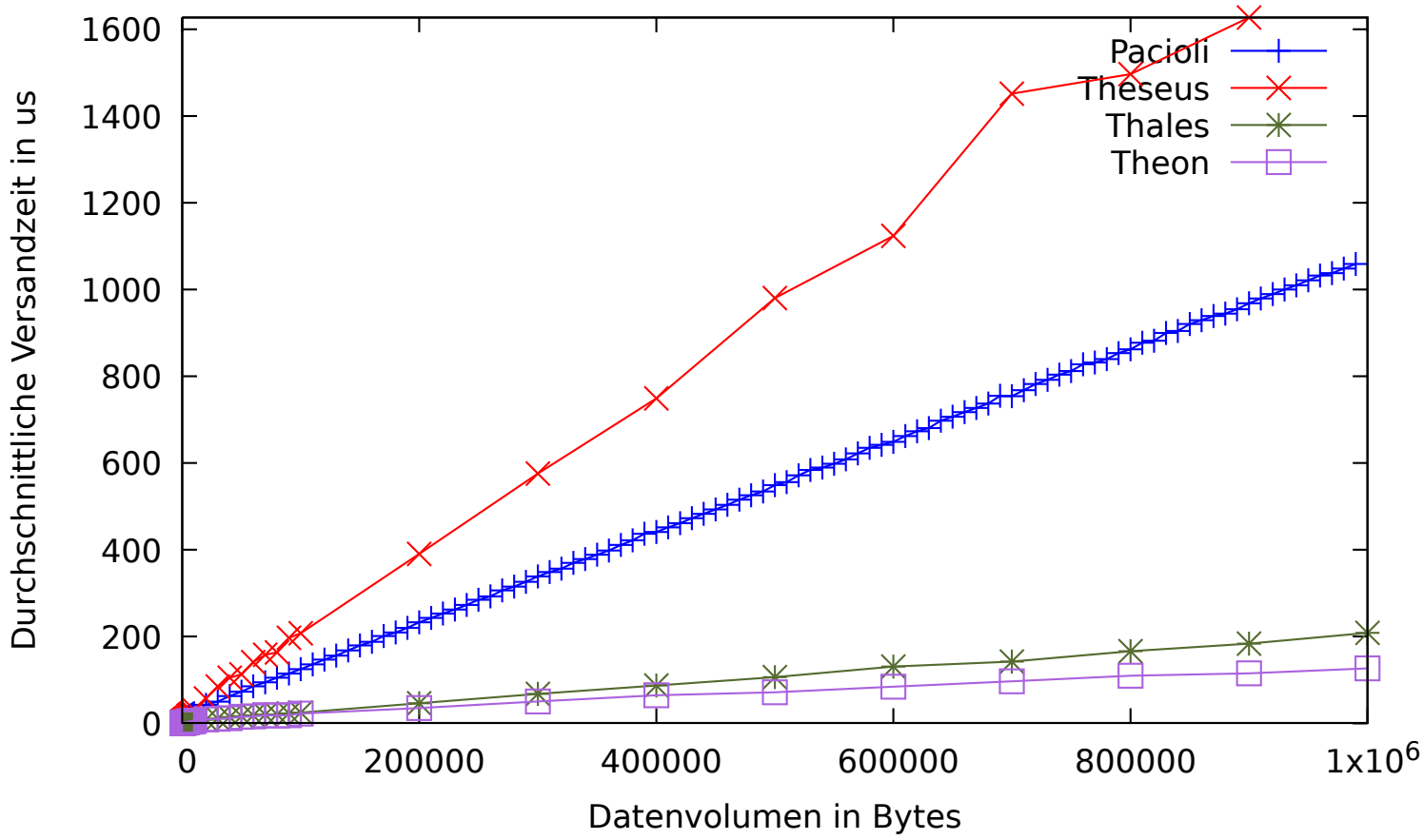
```
theon$ make
mpic++ -g -Ofast -std=c++11 -Wno-literal-suffix -std=c++11  -c -o mpi-
simpson.o mpi-simpson.cpp
mpic++ -o mpi-simpson -std=c++11 mpi-simpson.o
theon$ time mpirun -np 1 mpi-simpson 1000000000
3.1415926535896

real    0m3.468s
user    0m5.327s
sys     0m1.484s
theon$ time mpirun -np 8 mpi-simpson 1000000000
3.1415926535897

real    0m1.073s
user    0m5.926s
sys     0m0.949s
theon$
```

- Mit *mpirun* können MPI-Anwendungen gestartet werden.
- Wenn das Programm ohne *mpirun* läuft, dann gibt es nur einen einzigen Prozess.
- Die Option *-np* spezifiziert die Zahl der zu startenden Prozesse. Per Voreinstellung starten die alle auf der gleichen Maschine.

```
heim$ cat my-machines
multscher
geyer
syrlin
heim
heim$ time mpirun -hostfile my-machines -np 4 \
> mpi-simpson 10000000 2>/dev/null
3.1415926535899

real    0m0.560s
user    0m0.032s
sys     0m0.016s
heim$
```

- Die Option *-hostfile* ermöglicht auf den Suns die Spezifikation einer Datei mit Rechnernamen. Diese Datei sollte soviel Einträge enthalten, wie Prozesse gestartet werden.
- Bei OpenMPI werden die Prozesse auf den anderen Rechnern mit Hilfe der *ssh* gestartet. Letzteres sollte ohne Passwort möglich sein. Entsprechend sollte mit *ssh-keygen* ein Schlüsselpaar erzeugt werden und der eigene öffentliche Schlüssel in *~/.ssh/authorized_keys* integriert werden.
- Das reguläre Ethernet mit TCP/IP ist jedoch langsam!

Gemessene Versandzeiten bei MPI



- Pacioli: 8 Prozesse, Infiniband. Gemeinsamer Speicher: Theseus: 6 Prozesse; Thales: 8 Prozesse (2 Intel X5650-Prozessoren, 2,6 GHz); Theon: 8 Prozesse (1 Intel Xeon E5-2650v4-Prozessor, 2,2 GHz)

Warum schneidet die Pacioli mit dem Infiniband besser als die Theseus ab?

- OpenMPI nutzt zwar gemeinsame Speicherbereiche zur Kommunikation, aber dennoch müssen die Daten beim Transfer zweifach kopiert werden.
- Das Kopieren erfolgt zu Lasten der normalen CPUs.
- Hier wäre OpenMP grundsätzlich wesentlich schneller, da dort der doppelte Kopieraufwand entfällt.
- Sobald kein nennenswerter Kopieraufwand notwendig ist, dann sieht die Theseus mit ihren niedrigeren Latenzzeiten besser aus: 2,2 μs vs. 4,8 μs bei Pacioli. (Thales: 0,62 μs; Theon: 0,47 μs).

- Bei inhomogenen Rechnerleistungen oder bei einer inhomogenen Stückelung in Einzelaufgaben kann es sinnvoll sein, die Last dynamisch zu verteilen.
- In diesem Falle übernimmt ein Prozess die Koordination, indem er Einzelaufträge vergibt, die Ergebnisse aufsammelt und – sofern noch mehr zu tun ist – weitere Aufträge verschickt.
- Die anderen Prozesse arbeiten alle als Worker, die Aufträge entgegennehmen, verarbeiten und das Ergebnis zurücksenden.
- Dies wird aus Gründen der Einfachheit an einem Beispiel der Matrix-Vektor-Multiplikation demonstriert, wobei diese Technik in diesem konkreten Beispiel wegen des Kopieraufwands nichts bringt.

Angenommen, die Matrix habe m Zeilen und uns stehen n Worker zur Verfügung. Der Einfachheit halber wird $m > n$ angenommen. Dann sehen die Rollen wie folgt aus:

- Master:
  - Verteile den Wert n und den Vektor an alle n Worker.
  - Versende jedem der n Worker eine Zeile der Matrix.
  - Insgesamt $m - n$ Mal: Empfange von irgendeinem Worker einen Wert des Resultatsvektors und schicke in Antwort eine weitere Zeile der Matrix.
  - Insgesamt n Mal: Empfange von irgendeinem Worker einen Wert des Resultatsvektors und signalisiere in der Antwort das Ende.
- Worker:
  - Empfange den Wert n und den Vektor.
  - Für jede erhaltene Matrixzeile wird das entsprechende Skalarprodukt berechnet und verschickt.
  - Der Prozess endet, wenn das Ende signalisiert wird.

Seien $n = 2$ und $m = 4$. Dann kann das so in CSP übertragen werden:

$$
\begin{aligned}
P &= \mathit{Master} \parallel (\mathit{Worker} \mid\mid\mid \mathit{Worker}) \\
\mathit{Master} &= \mathit{broadcast\_parameters} \rightarrow \mathit{broadcast\_parameters} \rightarrow \\
&\quad \mathit{exchange\_row} \rightarrow \mathit{exchange\_row} \rightarrow \\
&\quad \mathit{exchange\_value} \rightarrow \mathit{exchange\_row} \rightarrow \\
&\quad \mathit{exchange\_value} \rightarrow \mathit{exchange\_row} \rightarrow \\
&\quad \mathit{exchange\_value} \rightarrow \mathit{finish} \rightarrow \\
&\quad \mathit{exchange\_value} \rightarrow \mathit{finish} \rightarrow \\
&\quad \mathit{SKIP}_{\alpha \mathit{Master}} \\
\mathit{Worker} &= \mathit{broadcast\_parameters} \rightarrow \mathit{ActiveWorker} \\
\mathit{ActiveWorker} &= \mathit{exchange\_row} \rightarrow \mathit{exchange\_value} \rightarrow \mathit{ActiveWorker} \mid \\
&\quad \mathit{finish} \rightarrow \mathit{SKIP}_{\alpha \mathit{Worker}}
\end{aligned}
$$

MPI und CSP kommen sich in der Ausdrucksform hier sehr nahe:

- Die Datenübertragung erfolgt im einfachsten Falle synchron. Die *exchange_row*- und *exchange_value*-Ereignisse entsprechen jeweils einer Paarung von *MPI_Send* und *MPI_Recv*, die ebenfalls synchron erfolgen sollten.
- Bei *MPI_Send* und *MPI_Recv* wird zusätzlich noch eine Markierung in Form eines ganzzahligen Werts mit übertragen, der die Art der Nachricht charakterisiert (*tag value*). Dieser Wert kann verwendet werden, um auf der Seite des Workers das Empfangen einer weiteren Zeile von dem Empfangen des Endesignals unterscheiden zu können. Im Beispiel werden hier die Werte *NEXT_ROW* und *FINISH* verwendet.

mpi-gemv.cpp

```
int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);
   int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   int nofworkers; MPI_Comm_size(MPI_COMM_WORLD, &nofworkers);
   --nofworkers; assert(nofworkers > 0);

   if (rank == 0) {
      int n; double* A; double* x;
      if (!read_parameters(n, A, x)) {
         std::cerr << "Invalid input!" << std::endl;
         MPI_Abort(MPI_COMM_WORLD, 1);
      }
      double* y = new double[n];
      gemv_master(n, A, x, y, nofworkers);
      for (int i = 0; i < n; ++i) {
         std::cout << " " << y[i] << std::endl;
      }
      delete[] A; delete[] x; delete[] y;
   } else {
      gemv_worker();
   }

   MPI_Finalize();
}
```

mpi-gemv.cpp

```
static void gemv_worker() {
   int n;
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   std::unique_ptr<double[]> x(new double[n]);
   MPI_Bcast(x.get(), n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   std::unique_ptr<double[]> row(new double[n]);
   // receive tasks and process them
   for(;;) {
      // receive next task
      MPI_Status status;
      MPI_Recv(row.get(), n, MPI_DOUBLE, 0,
         MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      if (status.MPI_TAG == FINISH) break;
      // process it
      double result = 0;
      for (int i = 0; i < n; ++i) {
         result += row[i] * x[i];
      }
      // send result back to master
      MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
   }
}
```

mpi-gemv.cpp

```
int n;
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
std::unique_ptr<double[]> x(new double[n]);
MPI_Bcast(x.get(), n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- Zu Beginn werden die Größe des Vektors und der Vektor selbst übermittelt.
- Da alle Worker den gleichen Vektor (mit unterschiedlichen Zeilen der Matrix) multiplizieren, kann der Vektor ebenfalls gleich zu Beginn mit *Bcast* an alle verteilt werden.

mpi-gemv.cpp

```
      MPI_Status status;
      MPI_Recv(row.get(), n, MPI_DOUBLE, 0,
         MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      if (status.MPI_TAG == FINISH) break;
```

- Mit *MPI_Recv* wird hier aus der globalen Gruppe eine Nachricht empfangen.
- Die Parameter: Zeiger auf den Datenpuffer, die Zahl der Elemente, der Element-Datentyp, der sendende Prozess, die gewünschte Art der Nachricht (*MPI_ANY_TAG* akzeptiert alles), die Gruppe und der Status, über den Nachrichtenart ermittelt werden kann.
- Nachrichtenarten gibt es hier zwei: *NEXT_ROW* für den nächsten Auftrag und *FINISH*, wenn es keine weiteren Aufträge mehr gibt.

mpi-gemv.cpp

```
MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

- *MPI_Send* versendet eine individuelle Nachricht synchron, d.h. diese Methode kehrt erst dann zurück, wenn der Empfänger die Nachricht erhalten hat.
- Die Parameter: Zeiger auf den Datenpuffer, die Zahl der Elemente (hier 1), der Element-Datentyp, der Empfänger-Prozess (hier 0) und die Art der Nachricht (0, spielt hier keine Rolle).

mpi-gemv.cpp

```
static void
gemv_master(int n, double* A, double *x, double* y,
            int nofworkers) {
   // broadcast parameters that are required by all workers
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   // send out initial tasks for all workers
   /* we need to keep record which row was handed out to which worker */
   std::unique_ptr<int[]> tasks(new int[nofworkers]);
   // ...

   // collect results and send out remaining tasks
   // ...
}
```

- Zu Beginn werden die beiden Parameter n und x, die für alle Worker gleich sind, mit *Bcast* verteilt.
- Danach erhält jeder der Worker einen ersten Auftrag.
- Anschließend werden Ergebnisse eingesammelt und – sofern noch etwas zu tun übrig bleibt – die Anschlußaufträge verteilt.

mpi-gemv.cpp

```
// send out initial tasks for all workers
// we need to keep record which row was handed out to which worker
std::unique_ptr<int[]> tasks(new int[nofworkers]);
int next_task = 0;
for (int worker = 1; worker <= nofworkers; ++worker) {
   if (next_task < n) {
      int row = next_task++; // pick next remaining task
      MPI_Send(&A[row*n], n, MPI_DOUBLE, worker, NEXT_ROW, MPI_COMM_WORLD);
      tasks[worker-1] = row; // remember which task was sent out to whom
   } else {
      // there is no work left for this worker
      MPI_Send(0, 0, MPI_DOUBLE, worker, FINISH, MPI_COMM_WORLD);
   }
}
```

- Die Worker erhalten zu Beginn jeweils eine Zeile der Matrix A, die sie dann mit x multiplizieren können.

mpi-gemv.cpp

```
// collect results and send out remaining tasks
int done = 0;
while (done < n) {
   // receive result of a completed task
   double value = 0; // initialize it to get rid of warning
   MPI_Status status;
   MPI_Recv(&value, 1, MPI_DOUBLE,
      MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
   int worker = status.MPI_SOURCE;
   int row = tasks[worker-1];
   y[row] = value; ++done;
   // send out next task, if there is one left
   if (next_task < n) {
      row = next_task++;
      MPI_Send(&A[row*n], n, MPI_DOUBLE, worker, NEXT_ROW,
         MPI_COMM_WORLD);
      tasks[worker-1] = row;
   } else {
      // send notification that there is no more work to be done
      MPI_Send(0, 0, MPI_DOUBLE, worker, FINISH, MPI_COMM_WORLD);
   }
}
```

Beachtenswert ist hier, dass bei der Übertragung eines Arrays die Länge dynamisch gewählt weden kann:

- Beim Versenden der nächsten Matrixzeile werden nebem dem *tag value* noch *n* Werte übermittelt:

```
MPI_Send(A[row], n, MPI_DOUBLE, slave, NEXT_ROW, MPI_COMM_WORLD);
```

- Beim Übermitteln des Endesignals wird als Array-Länge die 0 angegeben, d.h. es wird nur *FINISH* übertragen:

```
MPI_Send(0, 0, MPI_DOUBLE, slave, FINISH, MPI_COMM_WORLD);
```

Die Kombination von ganzzahligen Paketarten (hier *NEXT_ROW* oder *FINISH*) mit dynamischen Arrays vermeidet die Aufsplittung solcher Pakete in getrennte Header- und Datenpakete, die die Latenzzeiten erhöhen würden.

mpi-gemv.cpp

```
      MPI_Status status;
      MPI_Recv(&value, 1, MPI_DOUBLE,
         MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      int worker = status.MPI_SOURCE;
```

- Mit *MPI_ANY_SOURCE* wird angegeben, dass ein beliebiger Sender akzeptiert wird.
- Hier ist die Identifikation des Worker wichtig, damit das Ergebnis korrekt in y eingetragen werden kann. Dies erfolgt hier durch das Auslesen von *status.MPI_SOURCE*.

```
int MPI_Send(void* buf, int count,
             MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
```

- MPI-Nachrichten bestehen aus einem Header und der zu versendenden Datenstruktur (*buf*, *count* und *datatype*).
- Der (sichtbare) Header ist ein Tupel bestehend aus der
  - Kommunikationsdomäne (normalerweise *MPI_COMM_WORLD*), dem
  - Absender (*rank* innerhalb der Kommunikationsdomäne) und einer
  - Markierung (*tag*).

```
int MPI_Recv(void* buf, int count,
             MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status* status);
```

Eine mit *MPI_Send* versendete MPI-Nachricht passt zu einem *MPI_Recv* beim Empfänger, falls gilt:

- die Kommunikationsdomänen stimmen überein,
- der Absender stimmt mit *source* überein oder es wurde *MPI_ANY_SOURCE* angegeben,
- die Markierung stimmt mit *tag* überein oder es wurde *MPI_ANY_TAG* angegeben,
- die Datentypen sind identisch und
- die Zahl der Elemente ist kleiner oder gleich der angegebenen Buffergröße.

Was passiert bei *MPI_Send*?

- Wenn die Gegenseite bei einem passenden *MPI_Recv* auf ein Paket wartet, werden die Daten direkt übertragen.
- Wenn die Gegenseite noch nicht in einem passenden *MPI_Recv* wartet, **kann** die Nachricht gepuffert werden. In diesem Falle wird „im Hintergrund“ darauf gewartet, dass die Gegenseite eine passende *MPI_Recv*-Operation ausführt.
- Alternativ kann *MPI_Send* solange blockieren, bis die Gegenseite einen passenden *MPI_Recv*-Aufruf absetzt.
- Wird die Nachricht übertragen oder kommt es zu einer Pufferung, so kehrt *MPI_Send* zurück. D.h. nach dem Aufruf von *MPI_Send* kann in jedem Falle der übergebene Puffer andersweitig verwendet werden.
- Die Pufferung ist durch den Kopieraufwand teuer, ermöglicht aber die frühere Fortsetzung des sendenden Prozesses.
- Ob eine Pufferung zur Verfügung steht oder nicht und welche Kapazität sie ggf. besitzt, ist systemabhängig.

Die Gefahr von Deadlocks bei *MPI_Send* 299

mpi-deadlock.cpp

```
int main(int argc, char** argv) {
   alarm(1); // terminate ourselves if we need more than 1s
   MPI_Init(&argc, &argv);
   int nofprocesses; MPI_Comm_size(MPI_COMM_WORLD, &nofprocesses);
   int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   assert(nofprocesses == 2); const int other = 1 - rank;
   constexpr unsigned int maxsize = 8192;
   double* bigbuf = new double[maxsize];
   for (int len = 1; len <= maxsize; len *= 2) {
      MPI_Send(bigbuf, len, MPI_DOUBLE, other, 0, MPI_COMM_WORLD);
      MPI_Status status;
      MPI_Recv(bigbuf, len, MPI_DOUBLE, other, 0, MPI_COMM_WORLD, &status);
      if (rank == 0) std::cout << "len = " << len << " survived" << std::endl;
   }
   MPI_Finalize();
}
```

- Hier versuchen die beiden Prozesse 0 und 1 sich erst jeweils etwas zuzusenden, bevor sie *MPI_Recv* aufrufen. Das kann nur mit Pufferung gelingen.

```
theon$ mpirun -np 2 mpi-deadlock
len = 1 survived
len = 2 survived
len = 4 survived
len = 8 survived
len = 16 survived
len = 32 survived
len = 64 survived
len = 128 survived
len = 256 survived
--------------------------------------------------------------------------
mpirun noticed that process rank 1 with PID 0 on node theon exited on signal 14 (Alarm Clock).
--------------------------------------------------------------------------
theon$
```

- Hier war die Pufferung nicht in der Lage, eine Nachricht mit 512 Werten des Typs **double** aufzunehmen.
- MPI-Anwendungen, die sich auf eine vorhandene Pufferung verlassen, sind unzulässig bzw. deadlock-gefährdet in Abhängigkeit der lokalen Rahmenbedingungen.

Die Prozesse P_0 und P_1 wollen jeweils zuerst senden und erst danach empfangen:

$$
\begin{aligned}
P &= P_0 \parallel P_1 \parallel \mathit{Network} \\
P_0 &= (p_0\mathit{SendsMsg} \rightarrow p_0\mathit{ReceivesMsg} \rightarrow P_0) \\
P_1 &= (p_1\mathit{SendsMsg} \rightarrow p_1\mathit{ReceivesMsg} \rightarrow P_1) \\
\mathit{Network} &= (p_0\mathit{SendsMsg} \rightarrow p_1\mathit{ReceivesMsg} \rightarrow \mathit{Network} \mid \\
&\qquad p_1\mathit{SendsMsg} \rightarrow p_0\mathit{ReceivesMsg} \rightarrow \mathit{Network})
\end{aligned}
$$

Das gleiche Szenario, bei dem P_0 und P_1 jeweils zuerst senden und dann empfangen, diesmal aber mit den Puffern $P_0 Buffer$ und $P_1 Buffer$:

$$
\begin{aligned}
P &= P_0 \parallel P_1 \parallel P_0 Buffer \parallel P_1 Buffer \parallel Network \\
P_0 &= (p_0 SendsMsg \rightarrow p_0 ReceivesMsgFromBuffer \rightarrow P_0) \\
P_0 Buffer &= (p_0 ReceivesMsg \rightarrow p_0 ReceivesMsgFromBuffer \rightarrow \\
&\quad P_0 Buffer) \\
P_1 &= (p_1 SendsMsg \rightarrow p_1 ReceivesMsgFromBuffer \rightarrow P_1) \\
P_1 Buffer &= (p_1 ReceivesMsg \rightarrow p_1 ReceivesMsgFromBuffer \rightarrow \\
&\quad P_1 Buffer) \\
Network &= (p_0 SendsMsg \rightarrow p_1 ReceivesMsg \rightarrow Network \mid \\
&\quad p_1 SendsMsg \rightarrow p_0 ReceivesMsg \rightarrow Network)
\end{aligned}
$$

Zu den häufigen Kommunikationsszenarien analog zum Fork-und-Join-Pattern gehören

- die Aufteilung eines Vektors oder einer Matrix an alle beteiligten Prozesse (*scatter*) und
- nach der Durchführung der verteilten Berechnungen das Einsammeln aller Ergebnisse, die wieder zu einem Vektor oder einer Matrix zusammengefasst werden sollen (*gather*).

Dies ließe sich mit *MPI_Send* und *MPI_Recv* erledigen, wobei ein ausgewählter Prozess (typischerweise mit *rank* 0) dann wiederholt *MPI_Send* bzw. *MPI_Recv* aufrufen müsste.
Die wegen der begrenzten Pufferung sich aufaddierenden Latenzzeiten lassen sich reduzieren, wenn das Parallelisierungspotential durch optimierte Operationen ausgenutzt wird.

mpi-gemv-sg.cpp

```
int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);
   int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   int n; double* A = nullptr; double* x = nullptr; double* y = nullptr;
   if (rank == 0) {
      // process arguments where we expect an input file (in) ...
      if (!read_parameters(in, n, A, x)) {
         std::cerr << "Invalid input!" << std::endl;
         MPI_Abort(MPI_COMM_WORLD, 1);
      }
      y = new double[n];
   }
   mpi_gemv(n, A, x, y); MPI_Finalize();
   if (rank == 0) {
      for (int i = 0; i < n; ++i) {
         std::cout << " " << y[i] << std::endl;
      }
   }
}
```

- Als Beispiel wird hier wieder die Matrix-Vektor-Multiplikation verwendet, weil das Verfahren gut demonstriert, nicht weil es ansonsten sinnvoll wäre.

mpi-gemv-sg.cpp

```
static void mpi_gemv(int n, double* A, double* x, double* y) {
   /* ... preparation ... */

   /* ... scatter rows of A among all participating processes ... */

   /* ... compute assigned part of the resulting vector ... */

   /* ... gather results ... */

   /* ... clean up ... */
}
```

- Die Funktion *mpi_gemv* wird von allen beteiligten Prozessen gemeinsam aufgerufen.
- Zu beachten ist, dass die Parameter nur beim Prozess 0 sinnvoll gefüllt sind.

mpi-gemv-sg.cpp

```
int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (!x) {
   assert(rank > 0);
   x = new double[n];
}
MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

int nofprocesses; MPI_Comm_size(MPI_COMM_WORLD, &nofprocesses);

int nofrows = n / nofprocesses;
int remainder = n % nofprocesses;
if (rank < remainder) {
   ++nofrows;
}
```

- Zunächst müssen n und der Vektor x an alle Prozesse verbreitet werden.
- Da nicht sichergestellt ist, dass n durch die Zahl der Prozesse teilbar ist, können die Zahl der zu bearbeitenden Matrixzeilen für die einzelnen Prozesse unterschiedlich sein.

mpi-gemv-sg.cpp

```
/* scatter rows of A among all participating processes */
int* counts = nullptr; int* displs = nullptr;
if (rank == 0) {
   counts = new int[nofprocesses]; displs = new int[nofprocesses];
   int offset = 0;
   for (int i = 0; i < nofprocesses; ++i) {
      displs[i] = offset; counts[i] = n / nofprocesses;
      if (i < remainder) {
         ++counts[i];
      }
      counts[i] *= n; offset += counts[i];
   }
}
double* myrows = new double[nofrows * n];
MPI_Scatterv(A, counts, displs, MPI_DOUBLE,
   myrows, nofrows * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- Wenn einheitlich aufgeteilt wird, können die einfacheren Funktionen *MPI_Scatter* und *MPI_Gather* verwendet werden.
- Hier werden die Arrays *counts* und *displs* verwendet, die festlegen, wieviel Elemente (hier vom Typ *MPI_DOUBLE*) jeder Prozess erhält und ab welchem Offset in A diese zu finden sind.

mpi-gemv-sg.cpp

```
/* compute assigned part of the resulting vector */
double* result = new double[nofrows];
for (unsigned int i = 0; i < nofrows; ++i) {
   double val = 0;
   for (unsigned int j = 0; j < n; ++j) {
      val += myrows[i*n + j] * x[j];
   }
   result[i] = val;
}
```

- Alle Prozesse einschließlich dem verteilenden Prozess 0 arbeiten jetzt mit *myrows*, dem Matrix-Ausschnitt, der mit *MPI_Scatter* verteilt wurde.

mpi-gemv-sg.cpp

```
/* gather results */
if (rank == 0) {
   int offset = 0;
   for (int i = 0; i < nofprocesses; ++i) {
      displs[i] = offset;
      counts[i] = n / nofprocesses;
      if (i < remainder) {
         ++counts[i];
      }
      offset += counts[i];
   }
}
MPI_Gatherv(result, nofrows, MPI_DOUBLE,
   y, counts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- *counts* und *displs* müssen neu berechnet werden, da jetzt nur noch ein Vektor zusammengesetzt wird.
- Die Zuordnung der jeweiligen Ausschnitte beim Verteilen und Aufsammeln erfolgt strikt nach der Rangordnung der Prozesse.

Im folgenden wird unsere Vorlesungsbibliothek aus HPC I von Michael Lehn und mir verwendet, die insbesondere Matrizen und Vektoren unterstützt:

- Sie steht auf unseren Maschinen unter */home/numerik/pub/pp/ss19/lib* zur Verfügung.
- Sie lässt sich herunterladen:
  `http://www.mathematik.uni-ulm.de/numerik/pp/ss19/hpc.tar.gz`
- Sie besteht nur aus Headern unterhalb eines Verzeichnisses mit dem Namen *hpc*.
- Beim Übersetzen ist dann nur „-I“ anzugeben mit dem Verzeichnis, worunter das *hpc*-Verzeichnis liegt.

Zusätzlich wird *fmt*::*printf* verwendet, das auf unseren Maschinen installiert ist und unter `https://github.com/afborchert/fmt` zur Verfügung steht.

In der HPC-Bibliothek belegen Vektoren und Matrizen nicht mehr notwendigerweise eine zusammenhängende Speicherfläche. Stattdessen wird mit relativen Abständen gearbeitet, um zum nächsten Element in einer Dimension zu gelangen.
Beispiel: Sei

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

und folgende Deklaration gegeben:

```
using namespace hpc::matvec;
using T = double;
using Matrix = GeMatrix<T>;
Matrix A(2, 3, StorageOrder::RowMajor);
for (auto [i, j, Aij]: A) {
   Aij = 1 + i * A.numCols() + j;
});
```

Dann beträgt der Abstand zwischen $A_{i,j}$ und $A_{i,j+1}$ 1 (*incCol*) und der Abstand zwischen $A_{i,j}$ und $A_{i+1,j}$ hat den Wert 3 (*incRow*).

Wenn die mittlere Spalte ausgewählt wird mit

```
auto column = A.col(1);
```

dann beträgt der Abstand zwischen aufeinanderfolgenden Elementen 3.

Oder wenn eine Teilmatrix ausgewählt wird (mit der zweiten und dritten Spalte) mittels

```
auto A_ = A.block(0, 1).dim(2, 2);
```

(Parameter: Indizes des ersten Elements bei *block*, gefolgt bei *dim* von der Zahl der Zeilen und Spalten), dann bleiben *incCol* bei 1 bzw. *incRow* bei 3. Anders als zuvor liegt die Teilmatrix nicht mehr zusammenhängend im Speicher, da sie nur Teil einer größeren Matrix ist.

Das bedeutet, dass einfache *MPI_Send* bzw. *MPI_Recv*-Operationen mit einem der Elementartypen (wie etwa *MPI_DOUBLE*) einen Vektor oder eine Matrix im allgemeinen Fall nicht übertragen können. Entsprechend müssen neue MPI-Datentypen definiert werden.

- Es gibt die Menge der Basistypen BT in MPI, der beispielsweise *MPI_DOUBLE* oder *MPI_INT* angehören.
- Ein Datentyp T mit der Kardinalität n ist in der MPI-Bibliothek eine Sequenz von Tupeln $\{(bt_1, o_1), (bt_2, o_2), \ldots, (bt_n, o_n)\}$, mit $bt_i \in BT$ und den zugehörigen Offsets $o_i \in \mathbb{Z}$ für $i = 1, \ldots, n$.
- Die Offsets geben die relative Position der jeweiligen Basiskomponenten zur Anfangsadresse an.
- Bezüglich der Kompatibilität bei *MPI_Send* und *MPI_Recv* sind zwei Datentypen T_1 und T_2 genau dann kompatibel, falls die beiden Kardinalitäten n_1 und n_2 gleich sind und $bt_{1_i} = bt_{2_i}$ für alle $i = 1, \ldots, n_1$ gilt.
- Bei *MPI_Send* sind Überlappungen zulässig, bei *MPI_Recv* haben sie einen undefinierten Effekt.
- Alle Datentypobjekte haben in der MPI-Bibliothek den Typ *MPI_Datatype*.

- Ein Zeilenvektor des Basistyps *MPI_DOUBLE* (8 Bytes) der Länge 4 hat den Datenyp
  $\{(DOUBLE, 0), (DOUBLE, 8), (DOUBLE, 16), (DOUBLE, 24)\}$.
- Ein Spaltenvektor der Länge 3 aus einer 5×5-Matrix hat den Datentyp
  $\{(DOUBLE, 0), (DOUBLE, 40), (DOUBLE, 80)\}$.
- Die Spur einer 3×3-Matrix hat den Datentyp
  $\{(DOUBLE, 0), (DOUBLE, 32), (DOUBLE, 64)\}$.
- Die obere Dreiecks-Matrix einer 3×3-Matrix:
  $\{(DOUBLE, 0), (DOUBLE, 8), (DOUBLE, 16),$
  $(DOUBLE, 32), (DOUBLE, 40), (DOUBLE, 64)\}$

Alle Konstruktoren sind Funktionen, die als letzte Parameter den zu verwenden Elementtyp und einen Zeiger auf den zurückzuliefernden Typ erhalten:

MPI_Type_contiguous(*count*, *elemtype*, *newtype*)
: zusammenhängender Vektor aus *count* Elementen

MPI_Type_vector(*count*, *blocklength*, *stride*, *elemtype*, *newtype*)
: *count* Blöcke mit jeweils *blocklength* Elementen, deren Anfänge jeweils *stride* Elemente voneinander entfernt sind

MPI_Type_indexed(*count*, *blocklengths*, *offsets*, *elemtype*, *newtype*)
: *count* Blöcke mit jeweils individuellen Längen und Offsets

MPI_Type_create_struct(*count*, *blocklengths*, *offsets*, *elemtypes*, *newtype*)
: analog zu *MPI_Type_indexed*, aber jeweils mit individuellen Typen

Bei der Übertragung muss bezüglich der Reihenfolge einheitlich festgelegt werden, ob wir mit *row major* oder *col major* operieren. Im folgenden gehen wir von *row major* aus.

Für den Sonderfall, dass *incCol* den Wert 1 hat, d.h. dass innerhalb einer Zeile die Elemente unmittelbar hintereinander im Speicher liegen, lässt sich der Datentyp mit einem einzigen Aufruf von *MPI_Type_vector* erzeugen:

```
MPI_Datatype datatype;
MPI_Type_vector(
   /* count = */ A.numRows(),
   /* blocklength = */ A.numCols(),
   /* stride = */ A.incRow(),
   /* element type = */ get_type(A(0, 0)),
   /* newly created type = */ &datatype);
```

- Ein Matrix-Typ kann nicht einfach als Array von Vektoren definiert werden, wenn unklar ist, wie das funktioniert.
- Wenn bei einer Matrix *incRow* den Wert 1 hat (*col major*) und die Matrix in *row major* übertragen wird, dann überlappen sich die Speicherbereiche der Zeilen.

Sei T ein Datentyp mit $T = \{(et_1, o_1), (et_2, o_2), \ldots, (et_n, o_n)\}$:

- Für jeden Datentyp T sei der Umfang extent(T) definiert. Bei elementaren Datentypen wird das Resultat von **sizeof** verwendet. Beispiel: extent(MPI_DOUBLE) = 8.
- Die untere Schranke lb sei wie folgt definiert:

$$\mathrm{lb}(T) = \min_{1 \le i \le n} \{o_i\}$$

- Die obere Schranke berücksichtigt den Umfang:

$$\mathrm{ub}(T) = \max_{1 \le i \le n} \{o_i + \mathrm{extent}(et_i)\} + \epsilon$$

  Hierbei ist ϵ als notwendige Größe zum Aufrunden auf die nächste Alignment-Kante anzusehen.

- Dann lässt sich der Umfang allgemein definieren:

$$\mathrm{extent}(T) = \mathrm{ub}(T) - \mathrm{lb}(T)$$

Wenn beispielsweise durch *MPI_Type_vector* mehrere Elemente des Typs *elemtype* hintereinander gelegt werden, dann legt extent(*elemtype*) den relativen Abstand der nachfolgenden Elemente fest.

Problem: Angenommen wir haben

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

und diese Matrix ist spaltenweise im Speicher, d.h. in der Sequenz 1, 4, 2, 5, 3, 6 mit $incRow = 1$ und $incCol = 2$. Dann lässt sich der Typ für eine Zeile mit *MPI_Type_vector*(1, 1, 2, *MPI_DOUBLE*, &*row_type*) erzeugen. Dann gilt für $T = row\_type = \{(0, \text{MPI\_DOUBLE}), (16, \text{MPI\_DOUBLE}), (32, \text{MPI\_DOUBLE})\}$: $\text{lb}(T) = 0$, $\text{ub}(T) = 40$ und entsprechend $\text{extent}(T) = \text{ub}(T) - \text{lb}(T) = 40$.

Aus solchen Zeilentypen können wir nicht den korrekten Typ für die Matrix konstruieren, da der korrekte relative Offset bei aufeinanderfolgenden Zeilen 8 beträgt und nicht 40.

Datentyp für die Zeile einer Matrix

hpc/mpi/matrix.hpp

```
template<typename T, template<typename> class Matrix,
   Require<Ge<Matrix<T>>> = true>
MPI_Datatype get_row_type(const Matrix<T>& A) {
   MPI_Datatype rowtype;
   MPI_Type_vector(
      /* count = */ A.numCols(),
      /* blocklength = */ 1,
      /* stride = */ A.incCol(),
      /* element type = */ get_type(A(0, 0)),
      /* newly created type = */ &rowtype);

   /* in case of row-major we are finished */
   if (A.incRow() == A.numCols()) {
      MPI_Type_commit(&rowtype);
      return rowtype;
   }

   /* the extent of the MPI data type does not match
      the offset of subsequent rows -- this is a problem
      whenever we want to handle more than one row;
      to fix this we need to use the resize function
      which allows us to adapt the extent to A.incRow() */
   MPI_Datatype resized_rowtype;
   MPI_Type_create_resized(rowtype, 0, /* lb remains unchanged */
      A.incRow() * sizeof(T), &resized_rowtype);
   MPI_Type_commit(&resized_rowtype);
   MPI_Type_free(&rowtype);
   return resized_rowtype;
}
```

Datentyp für eine Matrix

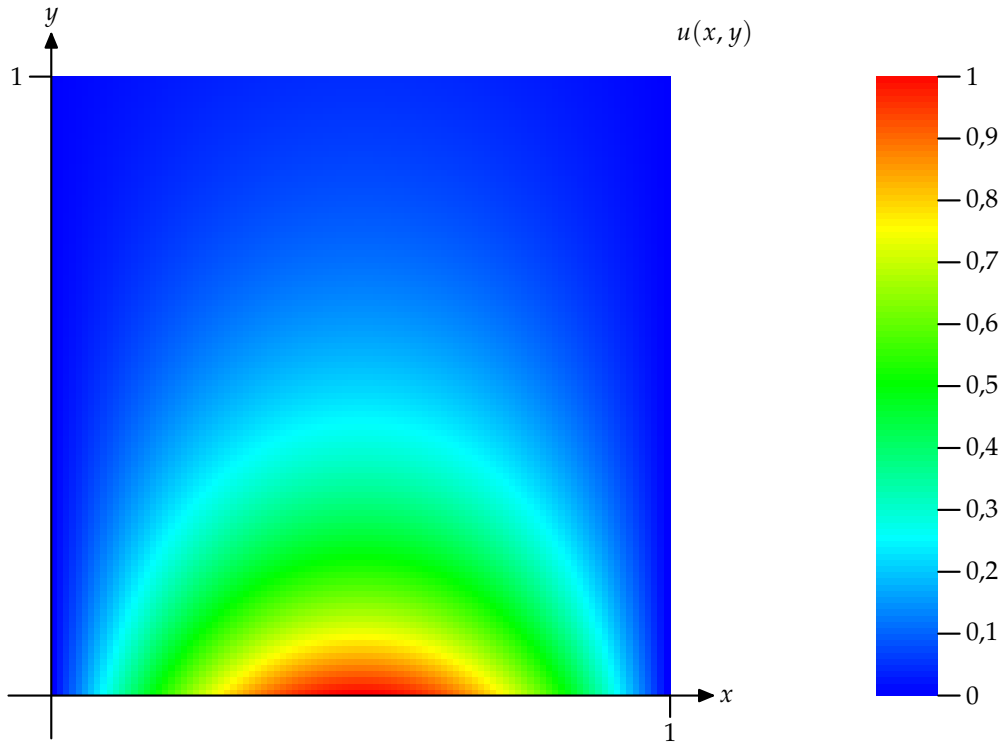
hpc/mpi/matrix.hpp

```
template<typename T, template<typename> class Matrix,
   Require<Ge<Matrix<T>>> = true>
MPI_Datatype get_type(const Matrix<T>& A) {
   MPI_Datatype datatype;
   if (A.incCol() == 1) {
      MPI_Type_vector(
         /* count = */ A.numRows(),
         /* blocklength = */ A.numCols(),
         /* stride = */ A.incRow(),
         /* element type = */ get_type(A(0, 0)),
         /* newly created type = */ &datatype);
   } else {
      /* vector of row vectors */
      MPI_Datatype rowtype = get_row_type(A);
      MPI_Type_contiguous(A.numRows(), rowtype, &datatype);
      MPI_Type_free(&rowtype);
   }
   MPI_Type_commit(&datatype);
   return datatype;
}
```

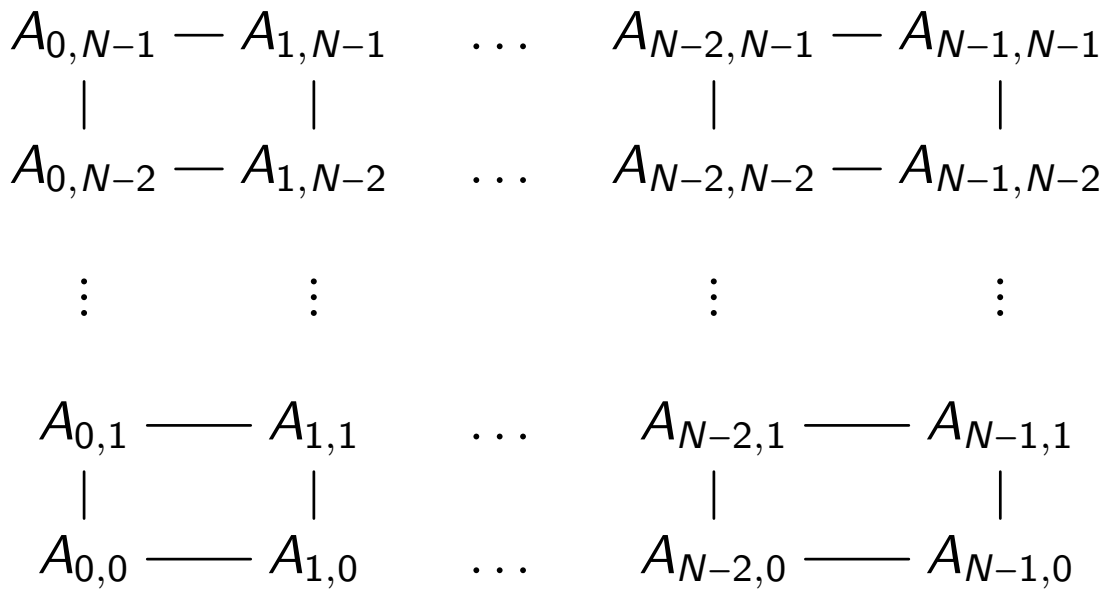
Wie können im Einzelfalle zu lösende Probleme in geeigneter Weise auf n Prozesse aufgeteilt werden?
Problempunkte:

- Mit wievielen Partnern muss ein einzelner Prozess kommunizieren? Lässt sich das unabhängig von der Problemgröße und der Zahl der beteiligten Prozesse begrenzen?
- Wieviele Daten sind mit den jeweiligen Partnern auszutauschen?
- Wie wird die Kommunikation so organisiert, dass Deadlocks sicher vermieden werden?
- Wieweit lässt sich die Kommunikation parallelisieren?

Fallstudie: Poisson-Gleichung mit Dirichlet-Randbedingung



- Gesucht sei eine numerische Näherung einer Funktion $u(x, y)$ für $(x, y) \in \Omega = [0, 1] \times [0, 1]$, für die gilt: $u_{xx} + u_{yy} = 0$ mit der Randbedingung $u(x, y) = g(x, y)$ für $x, y \in \delta\Omega$.
- Das obige Beispiel zeigt eine numerische Lösung für die Randbedingungen $u(x, 0) = sin(\pi x)$, $u(x, 1) = sin(\pi x)e^{-\pi}$ und $u(0, y) = u(1, y) = 0$.



- Numerisch lässt sich das Problem lösen, wenn das Gebiet Ω in ein $N \times N$ Gitter gleichmäßig zerlegt wird.
- Dann lässt sich $u(x, y)$ auf den Gitterpunkten durch eine Matrix A darstellen, wobei

$$A_{i,j} = u(\frac{i}{N}, \frac{j}{N})$$

für $i, j = 0 \ldots N$.

- Hierbei lässt sich A schrittweise approximieren durch die Berechnung von A_0, $A_1 \ldots$, wobei A_0 am Rand die Werte von $g(x, y)$ übernimmt und ansonsten mit Nullen gefüllt wird.
- Es gibt mehrere iterative numerische Verfahren, wovon das einfachste das Jacobi-Verfahren ist mit dem sogenannten 5-Punkt-Differenzenstern:

$$A_{k+1_{i,j}} = \frac{1}{4}\left(A_{k_{i-1,j}} + A_{k_{i,j-1}} + A_{k_{i,j+1}} + A_{k_{i+1,j}}\right)$$

  für $i, j \in 1 \ldots N-1, k = 1, 2, \ldots$
  (Zur Herleitung siehe Alefeld et al, *Parallele numerische Verfahren*, S. 18 ff.)
- Die Iteration wird solange wiederholt, bis

$$\max_{i,j=1\ldots N-1} \left|A_{k+1_{i,j}} - A_{k_{i,j}}\right| \leq \epsilon$$

  für eine vorgegebene Fehlergrenze ϵ gilt.

mpi-jacobi.cpp

```
template<typename T, template<typename> typename Matrix,
   Require<Ge<Matrix<T>>> = true>
T jacobi_iteration(const Matrix<T>& A, Matrix<T>& B) {
   assert(A.numRows() > 2 && A.numCols() > 2);
   T maxdiff = 0;
   for (std::size_t i = 1; i + 1 < B.numRows(); ++i) {
      for (std::size_t j = 1; j + 1 < B.numCols(); ++j) {
         B(i, j) = 0.25 *
            (A(i - 1, j) + A(i + 1, j) + A(i, j - 1) + A(i, j + 1));
         T diff = std::fabs(A(i, j) - B(i, j));
         if (diff > maxdiff) maxdiff = diff;
      }
   }
   return maxdiff;
}
```

- *A.numRows* bzw. *B.numRows* entsprechen hier N. A repräsentiert A_k und B die Näherungslösung des folgenden Iterationsschritts A_{k+1}.

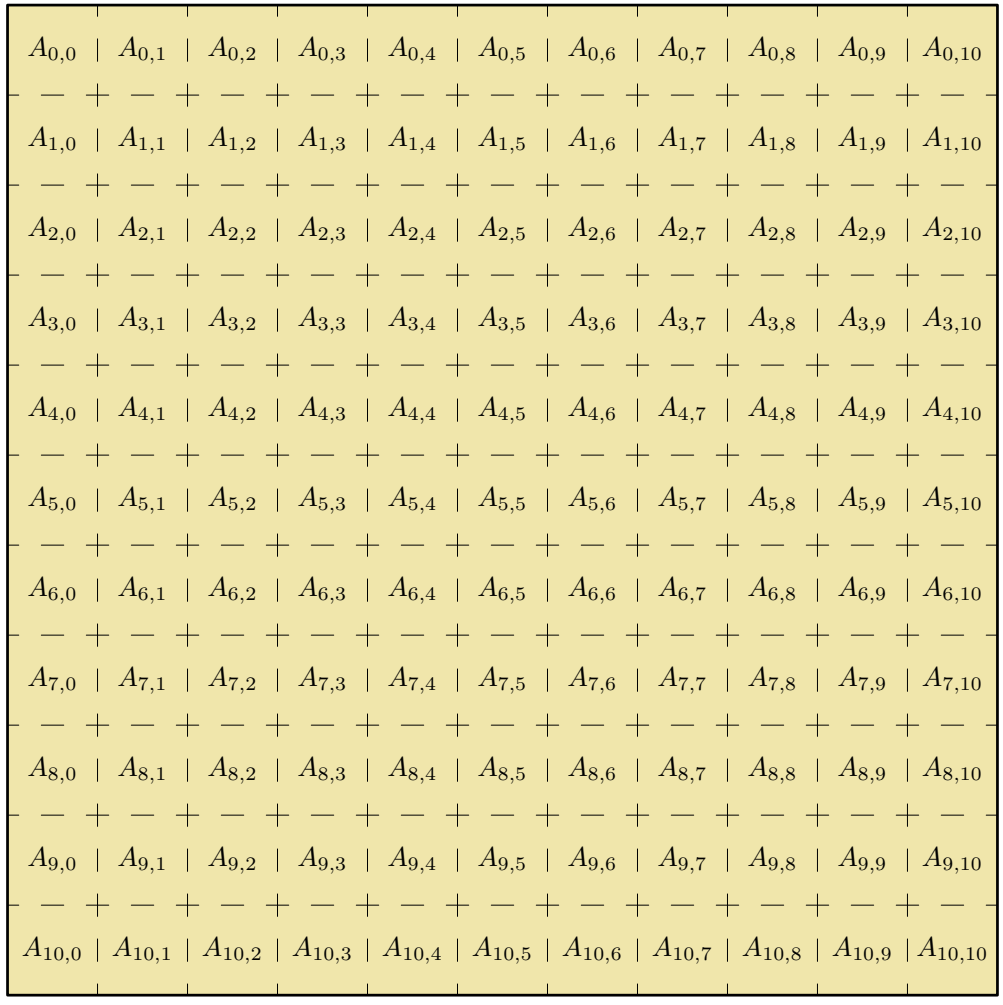
Initialisierung von A_0

mpi-jacobi.cpp

```
Matrix A(100, 100, Order::RowMajor);
if (rank == 0) {
   for (auto [i, j, Aij]: A) {
      if (j == 0) {
         Aij = std::sin(PI<T> * (T(i)/(A.numRows()-1)));
      } else if (j == A.numCols() - 1) {
         Aij = std::sin(PI<T> * (T(i)/(A.numRows()-1))) *
            E_POWER_MINUS_PI<T>;
      } else {
         Aij = 0;
      }
   }
}
```

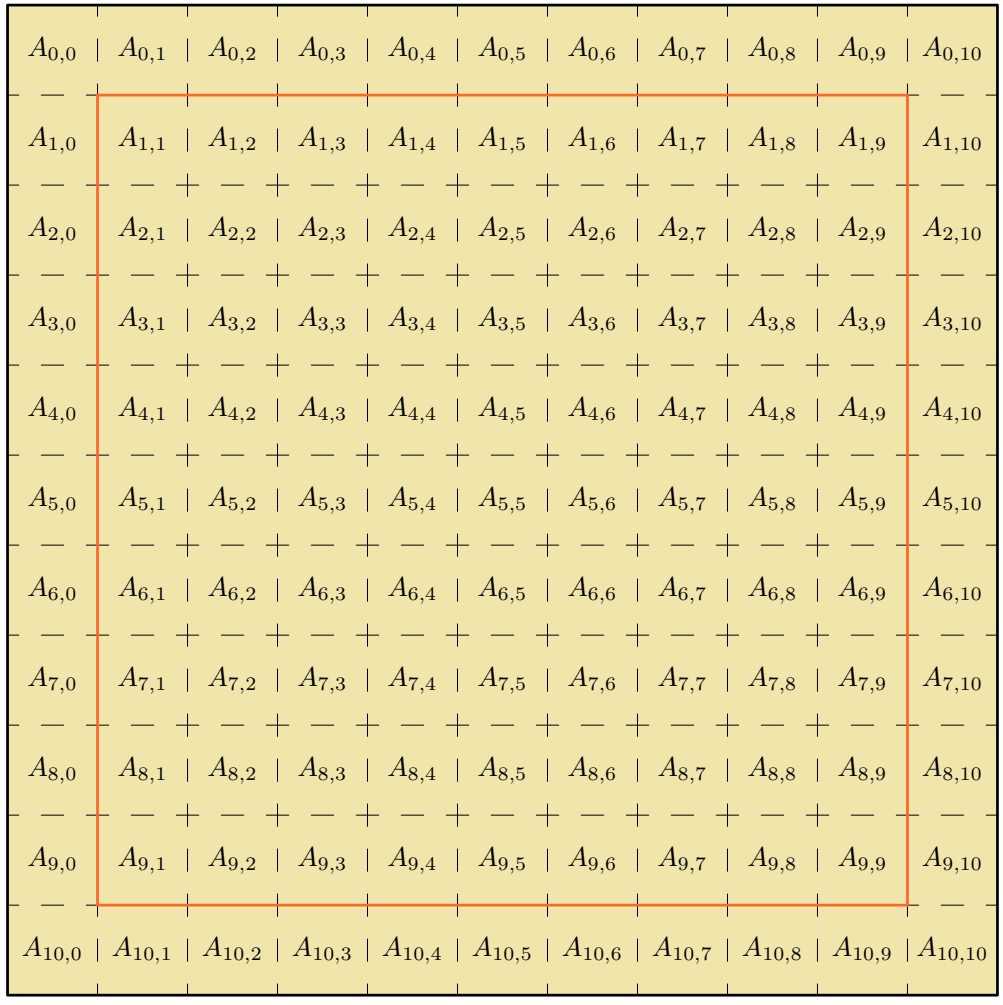
- Der gesamte Innenbereich wird mit 0 initialisiert.
- Für den Rand gelten die Randbedingungen $u(x,0) = sin(\pi x)$, $u(x,1) = sin(\pi x)e^{-\pi}$ und $u(0,y) = u(1,y) = 0$.

Eindimensionale Partitionierung



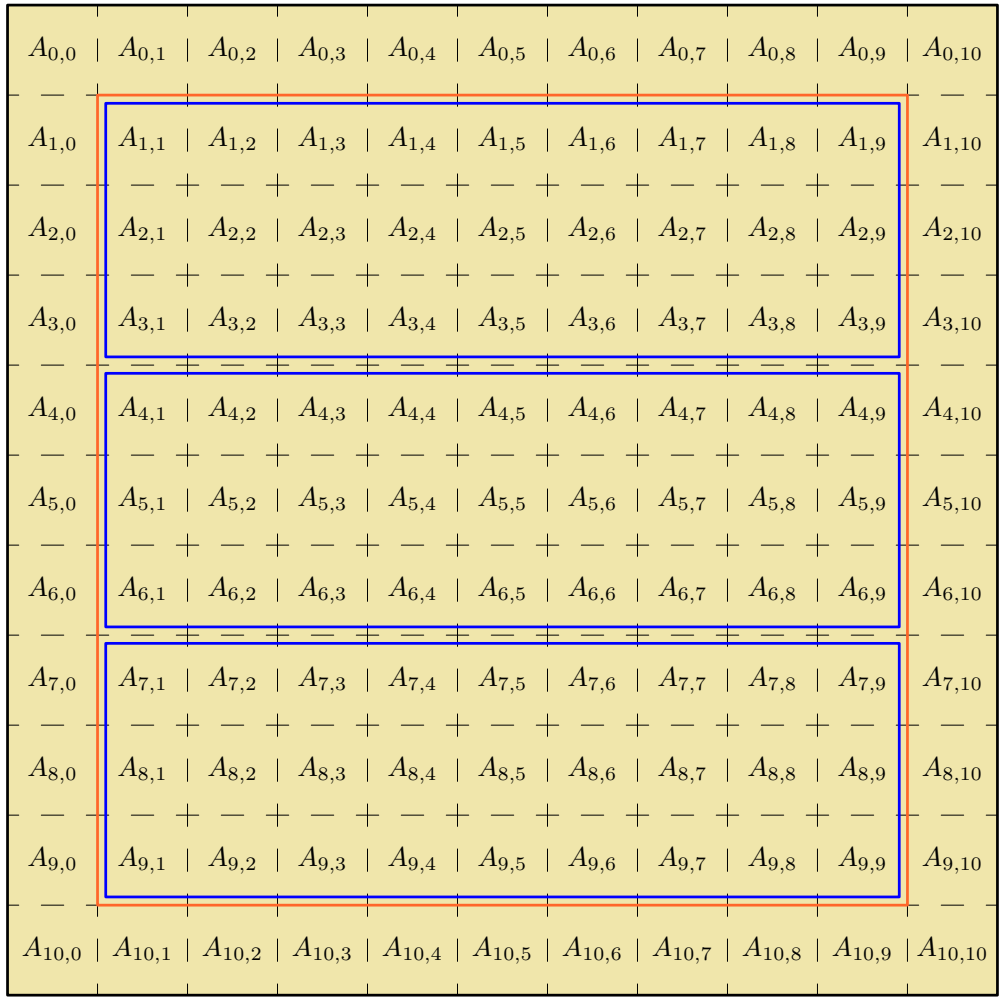
- Gegeben sei eine initialisierte Matrix A.

Eindimensionale Partitionierung



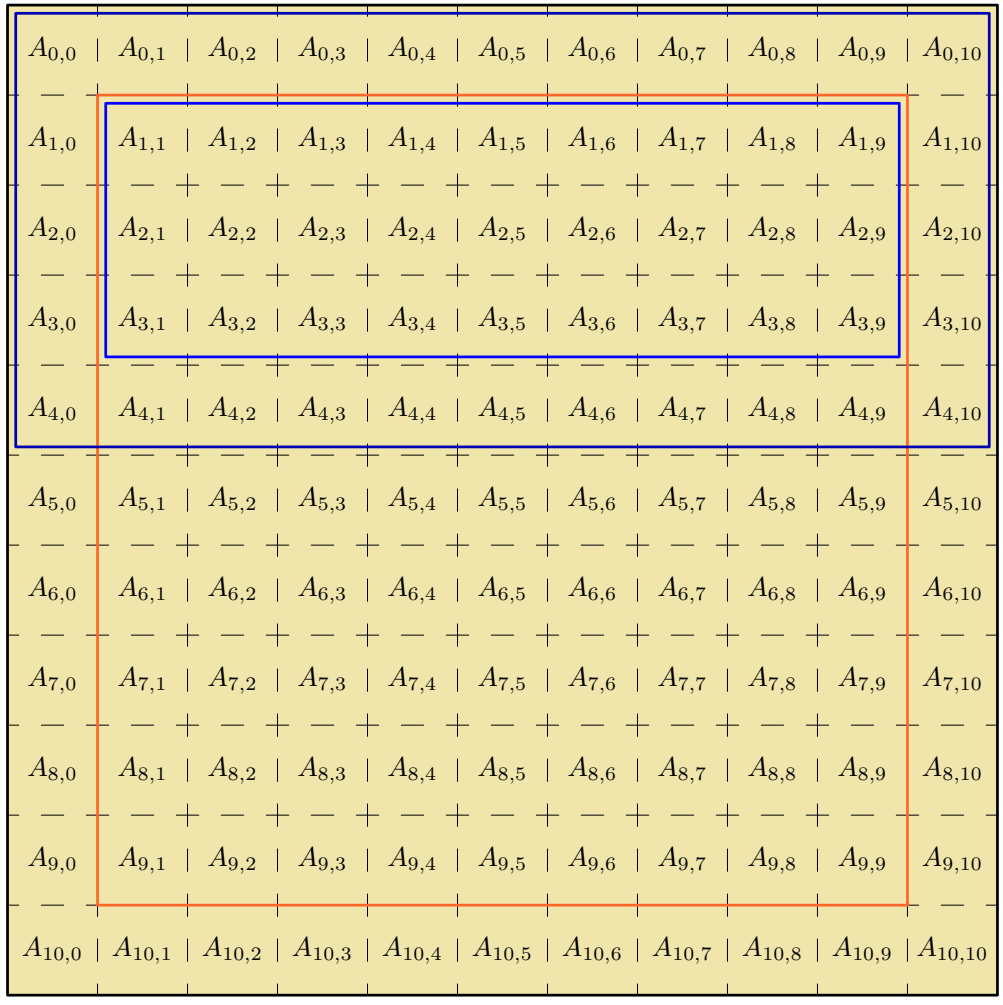
- Der Rand von A ist fest vorgegeben, der innere Teil ist näherungsweise zu bestimmen.

Eindimensionale Partitionierung



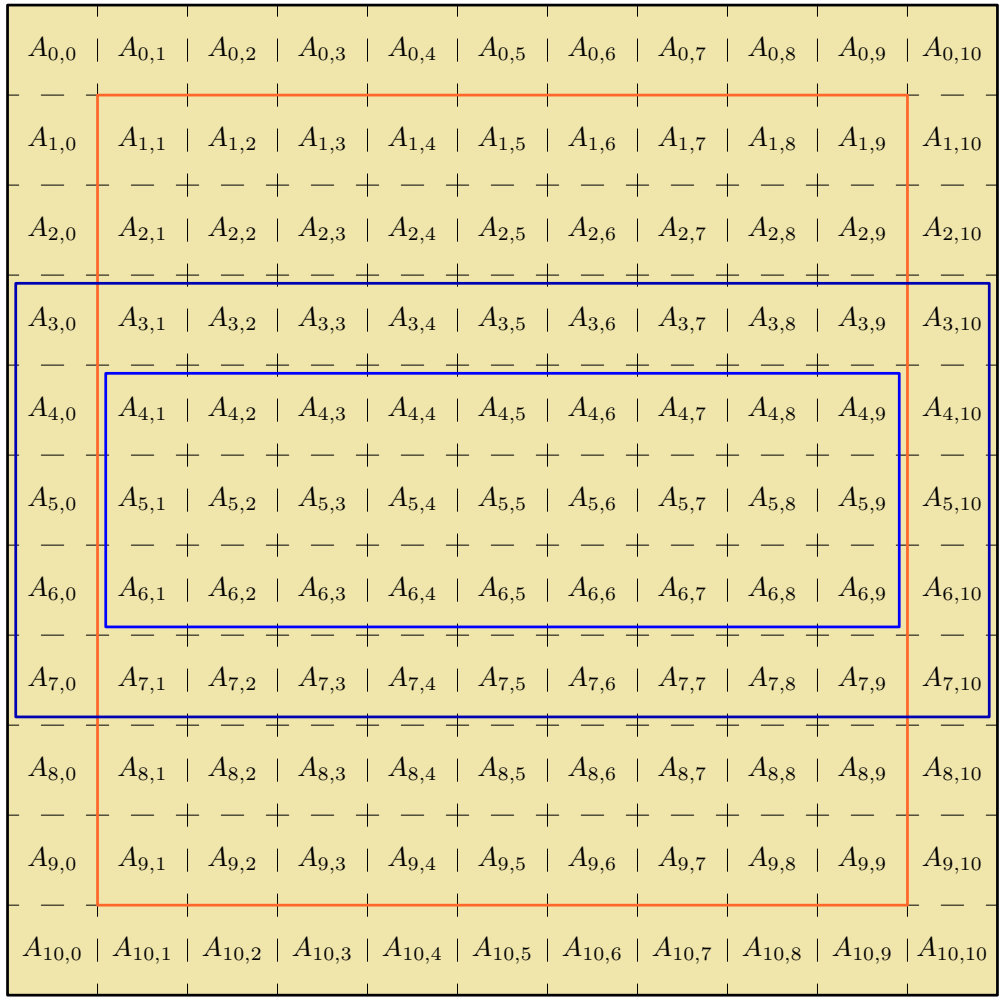
- Der innere Teil von A ist auf m Prozesse (hier $m = 3$) gleichmäßig aufzuteilen.

Eindimensionale Partitionierung

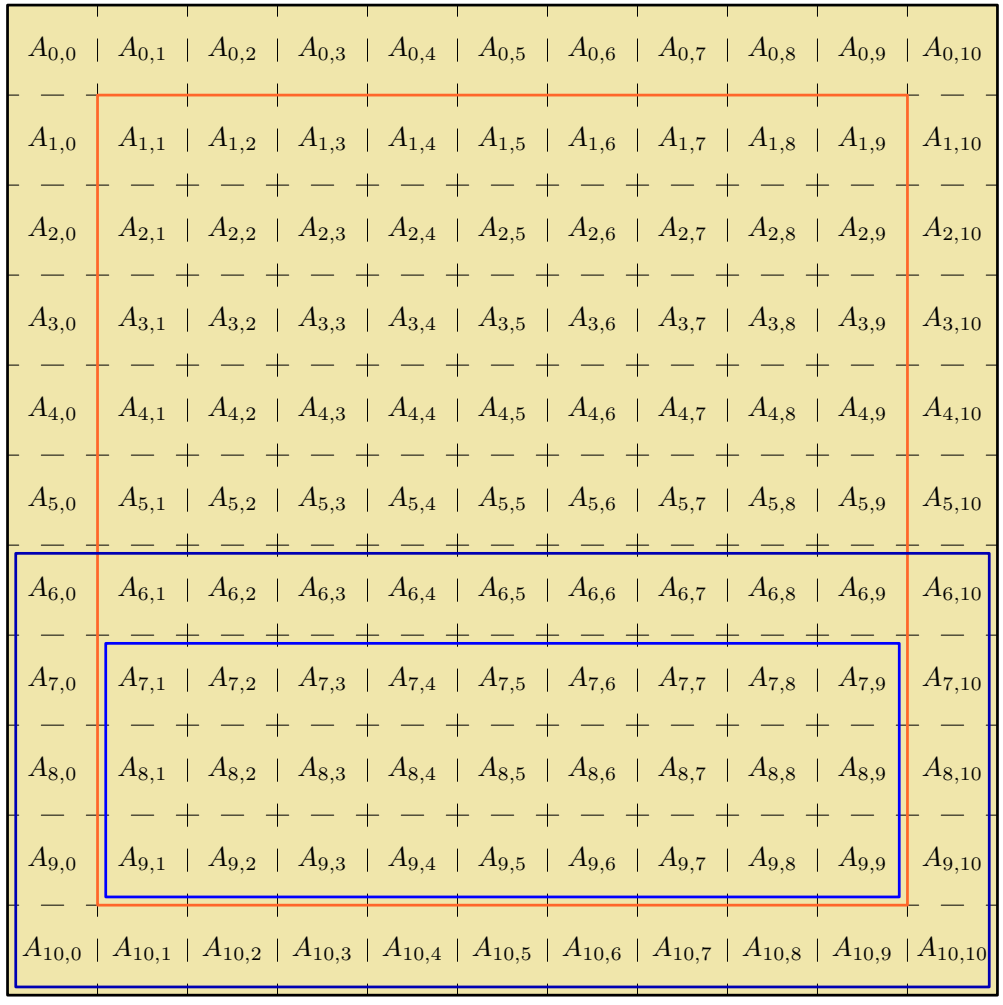


- Jeder der Prozesse benötigt lesenderweise den Rand. Soweit es sich nicht um den äußeren Rand handelt, muss dieser vom jeweiligen Nachbarn nach jedem Iterationsschritt erneut organisiert werden.

Eindimensionale Partitionierung



- Jeder der Prozesse benötigt lesenderweise den Rand. Soweit es sich nicht um den äußeren Rand handelt, muss dieser vom jeweiligen Nachbarn nach jedem Iterationsschritt erneut organisiert werden.



- Jeder der Prozesse benötigt lesenderweise den Rand. Soweit es sich nicht um den äußeren Rand handelt, muss dieser vom jeweiligen Nachbarn nach jedem Iterationsschritt erneut organisiert werden.

- Im vorgestellten Beispiel mit $N = 11$ und $m = 3$ sind folgende Übertragungen nach einem Iterationsschritt notwendig:
  - $P_1 \longrightarrow P_2 : A_{3,1} \ldots A_{3,9}$
  - $P_2 \longrightarrow P_3 : A_{6,1} \ldots A_{6,9}$
  - $P_3 \longrightarrow P_2 : A_{7,1} \ldots A_{7,9}$
  - $P_2 \longrightarrow P_1 : A_{4,1} \ldots A_{4,9}$
- Jede innere Partition erhält und sendet zwei innere Zeilen von A. Die Randpartitionen empfangen und senden jeweils nur eine Zeile.
- Generell müssen $2m - 2$ Datenblöcke mit jeweils $N - 2$ Werten verschickt werden. Dies lässt sich prinzipiell parallelisieren.

- Ein Ansatz wäre eine Paarbildung, d.h. zuerst kommunizieren die Prozesspaare $(0,1)$, $(2,3)$ usw. untereinander. Danach werden die Paare $(1,2)$, $(3,4)$ usw. gebildet. Bei jedem Paar würde zuerst der Prozess mit der niedrigeren Nummer senden und der mit der höheren Nummer empfangen und danach würden die Rollen jeweils vertauscht werden.
- Alternativ bietet sich auch die Verwendung der MPI-Operation *MPI_Sendrecv* an, die parallel eine *MPI_Send*- und eine *MPI_Recv*-Operation gleichzeitig verfolgt.
- Dann könnte der Austausch in zwei Wellen erfolgen, zuerst aufwärts von 0 nach 1, 1 nach 2 usw. und danach abwärts von m nach $m-1$, $m-1$ nach $m-2$ usw.

Vermeidung des Deadlocks mit *MPI_Sendrecv* 331

mpi-sendrecv.cpp

```
int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);
   int nofprocesses; MPI_Comm_size(MPI_COMM_WORLD, &nofprocesses);
   int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   assert(nofprocesses == 2); const int other = 1 - rank;
   const unsigned int maxsize = 8192;
   double* bigbuf[2] = {new double[maxsize], new double[maxsize]};
   for (int len = 1; len <= maxsize; len *= 2) {
      MPI_Status status;
      MPI_Sendrecv(
         bigbuf[rank], len, MPI::DOUBLE, other, 0,
         bigbuf[other], len, MPI::DOUBLE, other, 0,
         MPI_COMM_WORLD, &status);
      if (rank == 0) cout << "len = " << len << " survived" << endl;
   }
   MPI_Finalize();
}
```

- Bei *MPI_Sendrecv* werden zuerst die Parameter für *MPI_Send* angegeben, dann die für *MPI_Recv*.

mpi-jacobi.cpp

```
template<typename Matrix>
void exchange_with_neighbors(Matrix& A,
      int previous, int next, MPI_Datatype rowtype) {
   MPI_Status status;
   // upward
   MPI_Sendrecv(&A(1, 0), 1, rowtype, previous, 0,
      &A(A.numRows()-1, 0), 1, rowtype, next, 0,
      MPI_COMM_WORLD, &status);
   // downward
   MPI_Sendrecv(&A(A.numRows()-2, 0), 1, rowtype, next, 0,
      &A(0, 0), 1, rowtype, previous, 0,
      MPI_COMM_WORLD, &status);
}
```

- Der Austausch wird in zwei Wellen organisiert: Zuerst werden Zeilen von höheren Ranks zu niedrigeren übermittelt, dann umgekehrt.

mpi-jacobi.cpp

```
T eps = 1e-6; unsigned int iterations;
for (iterations = 0; ; ++iterations) {
   T maxdiff = jacobi_iteration(B1, B2);
   exchange_with_neighbors(B2, previous, next, rowtype);
   maxdiff = jacobi_iteration(B2, B1);
   if (iterations % 10 == 0) {
      T global_max;
      MPI_Reduce(&maxdiff, &global_max, 1, get_type(maxdiff),
         MPI_MAX, 0, MPI_COMM_WORLD);
      MPI_Bcast(&global_max, 1, get_type(maxdiff), 0, MPI_COMM_WORLD);
      if (global_max < eps) break;
   }
   exchange_with_neighbors(B1, previous, next, rowtype);
}
if (rank == 0) fmt::printf("%d iterations\n", iterations);
```

- Die zentrale Schleife läuft, bis das Abbruchkriterium $\max_{i,j=1\ldots N-1} \left|A_{k+1_{i,j}} - A_{k_{i,j}}\right| \leq \epsilon$ erfüllt ist.
- Die Matrizen B_1 und B_2 umfassen jeweils nur die lokal zu betrachtenden Zeilen und den Rand. Es werden immer zwei Schritte durchgeführt: $B_1 \to B_2$ gefolgt von $B_2 \to B_1$.

Initialisierung von B_1 und B_2

mpi-jacobi.cpp

```
UniformSlices<std::size_t> slices(nof_processes, A.numRows() - 2);
Matrix B1(slices.size(rank) + 2, A.numCols(), Order::RowMajor);
for (auto [i, j, Bij]: B1) {
   (void) i; (void) j; // suppress g++ warning
   Bij = 0;
}
auto B = B1.block(1, 0).dim(B1.numRows() - 2, B1.numCols());
MPI_Datatype rowtype = get_row_type(B);
```

- Mit *UniformSlices* wird festgelegt, wie die Zeilen der Gesamtmatrix A auf die einzelnen Prozesse aufgeteilt wird.
- B ist eine Teilmatrix ohne den linken und rechten Rand, der sich (nach der anfänglichen Initialisierung) nicht mehr verändert.
- Nur der innere Teil der Zeilen ist jeweils zu übermitteln.

mpi-jacobi.cpp

```
/* distribute main body of A */
auto A_ = A.block(1, 0).dim(A.numRows() - 2, A.numCols());
scatter_by_row(A_, B, 0, MPI_COMM_WORLD);
```

- $A\_$ ist der Innenteil der Matrix A ohne den Rand.
- Dann wird mit *scatter_by_row* der Innenteil der Gesamtmatrix A auf die Innenteile der individuellen Teilmatrizen B verteilt.

Übermittlung des Rands

mpi-jacobi.cpp

```
/* distribute first and last row of A */
if (rank == 0) {
   copy(A.block(0, 0).dim(1, A.numCols()),
      B1.block(0, 0).dim(1, B1.numCols()));
}
if (nof_processes == 1) {
   copy(A.block(A.numRows()-1, 0).dim(1, A.numCols()),
      B1.block(B.numRows()-1, 0).dim(1, B1.numCols()));
} else if (rank == 0) {
   MPI_Send(&A(A.numRows()-1, 0), 1, rowtype, nof_processes-1, 0,
      MPI_COMM_WORLD);
} else if (rank == nof_processes - 1) {
   MPI_Status status;
   MPI_Recv(&B1(B.numRows()-1, 0), 1, rowtype,
      0, 0, MPI_COMM_WORLD, &status);
}

Matrix B2(B1.numRows(), B1.numCols(), Order::RowMajor);
copy(B1, B2); /* actually just the border needs to be copied */
```

mpi-jacobi.cpp

```
int previous = rank == 0? MPI_PROC_NULL: rank-1;
int next = rank == nof_processes-1? MPI_PROC_NULL: rank+1;
```

- Damit die Sonderfälle am Rand nicht unnötig kompliziert werden, gibt es in MPI den sogenannten Nullprozess: *MPI_PROC_NULL*.
- Wenn bei einer der Kommunikationsoperationen der Nullprozess angegeben wird, dann findet eben keine Kommunikation statt.

mpi-jacobi.cpp

```
gather_by_row(B, A_, 0, MPI_COMM_WORLD);

MPI_Finalize();

if (rank == 0) {
   auto pixbuf = create_pixbuf(A, [](T val) -> HSVColor<float> {
      return HSVColor<float>((1-val) * 240, 1, 1);
   }, 8);
   gdk_pixbuf_save(pixbuf, "jacobi.jpg", "jpeg", nullptr,
      "quality", "100", nullptr);
}
```

- Mit *gather_by_row* wird beim Prozess mit dem Rank 0 die gesamte Matrix zusammengestellt.
- Sobald das fertig ist, kann das MPI-Kommunikationsnetzwerk herunterfahren werden.
- Mit *create_pixbuf* wird die Matrix in einen Pixelbuffer der GDK-Pixbuf-Bibliothek konvertiert und dieser danach in eine JPEG-Datei abgesichert.

- Mit *MPI_Isend* und *MPI_Irecv* bietet die MPI-Schnittstelle eine asynchrone Kommunikationsschnittstelle.
- Die Aufrufe blockieren nicht und entsprechend wird die notwendige Kommunikation parallel zum übrigen Geschehen abgewickelt.
- Wenn es geschickt aufgesetzt wird, können einige Berechnungen parallel zur Kommunikation ablaufen.
- Das ist sinnvoll, weil sonst die CPU-Ressourcen während der Latenzzeiten ungenutzt bleiben.
- Die Benutzung folgt dem Fork-And-Join-Pattern, d.h. mit *MPI_Isend* und *MPI_Irecv* läuft die Kommunikation parallel zum Programmtext nach den Aufrufen ab und mit *MPI_Wait* ist eine Synchronisierung wieder möglich.
- Zu beachten ist, dass die angegebenen Puffer während der laufenden Kommunikation nicht benutzt werden dürfen.

mpi-jacobi-nb.cpp

```
template<typename T, template<typename> class Matrix,
   Require<Ge<Matrix<T>>> = true>
void exchange_with_neighbors(Matrix<T>& A,
      /* ranks of the neighbors */
      int previous, int next,
      /* data type for an inner row, i.e. without the border */
      MPI_Datatype rowtype) {
   MPI_Request requests[4]; int request_index = 0;
   MPI_Irecv(&A(0, 1), 1, rowtype, previous, 0,
      MPI_COMM_WORLD, &requests[request_index++]);
   MPI_Irecv(&A(A.numRows()-1, 1), 1, rowtype, next, 0,
      MPI_COMM_WORLD, &requests[request_index++]);
   MPI_Isend(&A(1, 1), 1, rowtype, previous, 0,
      MPI_COMM_WORLD, &requests[request_index++]);
   MPI_Isend(&A(A.numRows()-2, 1), 1, rowtype, next, 0,
      MPI_COMM_WORLD, &requests[request_index++]);

   for (auto& request: requests) {
      MPI_Status status;
      MPI_Wait(&request, &status);
   }
}
```

mpi-jacobi-nb.cpp

```
MPI_Request requests[4]; int request_index = 0;
MPI_Irecv(&A(0, 1), 1, rowtype, previous, 0,
   MPI_COMM_WORLD, &requests[request_index++]);
MPI_Irecv(&A(A.numRows()-1, 1), 1, rowtype, next, 0,
   MPI_COMM_WORLD, &requests[request_index++]);
MPI_Isend(&A(1, 1), 1, rowtype, previous, 0,
   MPI_COMM_WORLD, &requests[request_index++]);
MPI_Isend(&A(A.numRows()-2, 1), 1, rowtype, next, 0,
   MPI_COMM_WORLD, &requests[request_index++]);
```

- Die Methoden *MPI_Isend* und *MPI_Irecv* werden analog zu *MPI_Send* und *MPI_Recv* aufgerufen, liefern aber ein *MPI_Request*-Objekt zurück.
- Die nicht-blockierenden Operationen initiieren jeweils nur die entsprechende Kommunikation. Der übergebene Datenbereich darf andersweitig nicht verwendet werden, bis die jeweilige Operation abgeschlossen ist.
- Dies kann durch die dadurch gewonnene Parallelisierung etwas bringen. Allerdings wird das durch zusätzlichen Overhead (mehr lokale Threads) bezahlt.

Abschluss einer nichtblockierenden Kommunikation 342

mpi-jacobi-nb.cpp

```
for (auto& request: requests) {
   MPI_Status status;
   MPI_Wait(&request, &status);
}
```

- Für Objekte des Typs *MPI_Request* stehen die Funktionen *MPI_Wait* und *MPI_Test* zur Verfügung.
- Mit *MPI_Wait* kann auf den Abschluss gewartet werden; mit *MPI_Test* ist die nicht-blockierende Überprüfung möglich, ob die Operation bereits abgeschlossen ist.

- Die Partitionierung eines Problems auf einzelne Prozesse und deren Kommunikationsbeziehungen kann als Graph repräsentiert werden, wobei die Prozesse die Knoten und die Kommunikationsbeziehungen die Kanten repräsentieren.
- Der Graph ist normalerweise ungerichtet, weil zumindest die zugrundeliegenden Kommunikationsarchitekturen und das Protokoll bidirektional sind.

Abbildung des Graphen auf eine physische Netzwerkstruktur

- Da die Bandbreiten und Latenzzeiten zwischen einzelnen rechnenden Knoten nicht überall gleich sind, ist es sinnvoll, die Aufteilung der Prozesse auf Knoten so zu organisieren, dass die Kanten möglichst weitgehend auf günstigere Kommunikationsverbindungen gelegt werden.
- Bei Infiniband spielt die Organisation kaum eine Rolle, es sei denn, es liegt eine Zwei-Ebenen-Architektur vor wie beispielsweise bei *SuperMUC* in München.
- Bei MP-Systemen mit gemeinsamen Speicher ist es günstiger, wenn miteinander kommunizierende Prozesse auf Kernen des gleichen Prozessors laufen, da diese typischerweise einen Cache gemeinsam nutzen können und somit der Umweg über den langsamen Hauptspeicher vermieden wird.
- Bei Titan und anderen Installationen, die in einem dreidimensionalen Torus organisiert sind, spielt Nachbarschaft eine wichtige Rolle.

- MPI bietet die Möglichkeit, beliebige Kommunikationsgraphen zu deklarieren.
- Zusätzlich unterstützt bzw. vereinfacht MPI die Deklarationen n-dimensionaler Gitterstrukturen, die in jeder Dimension mit oder ohne Ringstrukturen konfiguriert werden können. Entsprechend sind im eindimensionalen einfache Ketten oder Ringe möglich und im zweidimensionalen Fall Matrizen, Zylinder oder Tori.
- Dies eröffnet MPI die Möglichkeit, eine geeignete Zuordnung von Prozessen auf Prozessoren vorzunehmen.
- Ferner lassen sich über entsprechende MPI-Funktionen die Kommunikationsnachbarn eines Prozesses ermitteln.
- Grundsätzlich ist eine Kommunikation abseits des vorgegebenen Kommunikationsgraphen möglich. Nur bietet diese möglicherweise höhere Latenzzeiten und/oder niedrigere Bandbreiten.

mpi-jacobi-2d.cpp

```
/* create two-dimensional Cartesian grid for our prcesses */
int dims[2] = {0, 0}; int periods[2] = {false, false};
MPI_Dims_create(nof_processes, 2, dims);
MPI_Comm grid;
MPI_Cart_create(MPI_COMM_WORLD,
   2,         // number of dimensions
   dims,      // actual dimensions
   periods,   // both dimensions are non-periodical
   true,      // reorder is permitted
   &grid      // newly created communication domain
);
MPI_Comm_rank(grid, &rank); // update rank (could have changed)
```

- Mit *MPI_Dims_create* lässt sich die Anzahl der Prozesse auf ein Gitter aufteilen.
- Die Funktion *MPI_Cart_create* erzeugt dann das Gitter und teilt ggf. die Prozesse erneut auf, d.h. *rank* könnte sich dadurch ändern.

mpi-jacobi-2d.cpp

```
int dims[2] = {0, 0}; int periods[2] = {false, false};
MPI_Dims_create(nof_processes, 2, dims);
```

- Die Prozesse sind so auf ein zweidimensionales Gitter aufzuteilen, dass $dims[0]*dims[1] == nofprocesses$ gilt.
- *MPI_Dims_create* erwartet die Zahl der Prozesse, die Zahl der Dimensionen und ein entsprechend dimensioniertes Dimensions-Array.
- Die Funktion ermittelt die Teiler von *nofprocesses* und sucht nach einer in allen Dimensionen möglichst gleichmäßigen Aufteilung. Wenn *nofprocesses* prim ist, entsteht dabei die ungünstige Aufteilung $1 \times nofprocesses$.
- Das Dimensions-Array *dims* muss zuvor initialisiert werden. Bei Nullen darf *Compute_dims* einen Teiler von *nofprocesses* frei wählen; andere Werte müssen Teiler sein und sind dann zwingende Vorgaben.

mpi-jacobi-2d.cpp

```
int dims[2] = {0, 0}; int periods[2] = {false, false};
MPI_Dims_create(nof_processes, 2, dims);
MPI_Comm grid;
MPI_Cart_create(MPI_COMM_WORLD,
   2,         // number of dimensions
   dims,      // actual dimensions
   periods,   // both dimensions are non-periodical
   true,      // reorder is permitted
   &grid      // newly created communication domain
);
```

- *MPI_Cart_create* erwartet im zweiten und dritten Parameter die Zahl der Dimensionen und das entsprechende Dimensionsfeld.
- Der vierte Parameter legt über ein **int**-Array fest, welche Dimensionen ring- bzw. torusförmig angelegt sind. Hier liegt eine einfache Matrixstruktur vor und entsprechend sind beide Werte **false**.
- Der vierte und letzte Parameter erklärt, ob eine Neuzuordnung zulässig ist, um die einzelnen Prozesse und deren Kommunikationsstruktur möglichst gut auf die vorhandene Netzwerkstruktur abzubilden. Hier sollte normalerweise **true** gegeben werden. Allerdings ändert sich dann möglicherweise der *rank*, so dass dieser erneut abzufragen ist.

mpi-jacobi-2d.cpp

```
/* get our position within the grid */
int coords[2];
MPI_Cart_coords(grid, rank, 2, coords);
```

- Mit *MPI_Cart_coords* lässt sich der Rank einer Kommunikationsdomäne in Koordinaten konvertieren.
- Das nutzen wir hier, um die eigenen Koordinaten zu ermitteln.

Anlage der Blockmatrizen B_1 und B_2

mpi-jacobi-2d.cpp

```
/* create matrices B1, B2 for our subarea */
int overlap = 1;
UniformSlices<int> rows(dims[0], A.numRows() - 2*overlap);
UniformSlices<int> columns(dims[1], A.numCols() - 2*overlap);

Matrix B1(rows.size(coords[0]) + 2*overlap,
          columns.size(coords[1]) + 2*overlap,
          Order::RowMajor);
Matrix B2(rows.size(coords[0]) + 2*overlap,
          columns.size(coords[1]) + 2*overlap,
          Order::RowMajor);
```

- Mit Hilfe von *UniformSlices* wird der Innenteil von A zweidimensional möglichst gleichmäßig aufgeteilt.
- Entsprechend werden B_1 und B_2 angelegt.

mpi-jacobi-2d.cpp

```
/* distribute main body of A including left and right border */
scatter_by_block(A, B1, 0, grid, overlap);

copy(B1, B2); /* actually just the border needs to be copied */
```

- Mit *scatter_by_block* wird die gesamte Matrix blockweise aufgeteilt, wobei die sich überlappenden Teile (mit *overlap* = 1) mit berücksichtigt werden.
- Somit werden auch die Ränder mitkopiert, wobei wir darauf achten müssen, dass auch B_2 eine Kopie des Rands erhält.

mpi-jacobi-2d.cpp

```
/* get the process numbers of our neighbors */
int left, right, upper, lower;
MPI_Cart_shift(grid, 0, 1, &upper, &lower);
MPI_Cart_shift(grid, 1, 1, &left, &right);
```

- Die Funktion *MPI_Cart_shift* liefert die Nachbarn in einer der Dimensionen.
- Der zweite Parameter ist die Dimension, der dritte Parameter der Abstand (hier 1 für den unmittelbaren Nachbarn).
- Die Prozessnummern der so definierten Nachbarn werden in den beiden folgenden Variablen abgelegt.
- Wenn in einer Richtung kein Nachbar existiert (z.B. am Rande einer Matrix), wird *MPI_PROC_NULL* zurückgeliefert.

mpi-jacobi-2d.cpp

```
/* used to specify an I/O transfer with one
   of our neighbors;
   each process has four input and four output
   operations per iteration
*/
struct IOTransfer {
   int rank; /* rank of neighbor */
   void* addr; /* start address */
   MPI_Datatype datatype; /* either a column or row */
};
```

- Jeder Prozess kommuniziert in jedem Schritt mit vier anderen Prozessen bidirektional.
- Es ist sinnvoll, das mit Hilfe einer Datenstruktur zu organisieren.

mpi-jacobi-2d.cpp

```
/* compute type for inner rows and cols without the border */
auto B_inner_row = B1.block(0, 1).
   dim(overlap, B1.numCols() - 2*overlap);
MPI_Datatype rowtype = get_type(B_inner_row);
auto B_inner_col = B1.block(0, 1).
   dim(B1.numRows() - 2*overlap, overlap);
MPI_Datatype coltype = get_type(B_inner_col);

IOTransfer B1_in_vectors[] = {
   {left, &B1(1, 0), coltype}, {upper, &B1(0, 1), rowtype},
   {right, &B1(1, B1.numCols()-overlap), coltype},
   {lower, &B1(B1.numRows()-overlap, 1), rowtype},
};
IOTransfer B1_out_vectors[] = {
   {left, &B1(1, 1), coltype}, {upper, &B1(1, 1), rowtype},
   {right, &B1(1, B1.numCols()-2*overlap), coltype},
   {lower, &B1(B1.numRows()-2*overlap, 1), rowtype},
};
```

- Die vier Ein- und vier Ausgabevektoren werden hier zusammen mit den passenden Datentypen in Arrays zusammengestellt. Analog auch für B_2.

mpi-jacobi-2d.cpp

```
template<typename T, template<typename> typename Matrix,
   Require<Ge<Matrix<T>>> = true>
T jacobi_iteration(const Matrix<T>& A, Matrix<T>& B,
       const IOTransfer (&in_vectors)[4],
       const IOTransfer (&out_vectors)[4], MPI_Comm grid) {
   T maxdiff = 0;
   // compute border zones
   // ...
   // exchange borders with our neighbors
   // ...
   // compute inner block in parallel to the ongoing communication
   // ...
   // wait until the communication is finished
   // ...
   return maxdiff;
}
```

- Der generelle Aufbau der Iterationsschleife ist im Vergleich zur eindimensionalen Partitionierung gleich geblieben, abgesehen davon, dass
  - alle vier Ränder zu Beginn zu berechnen sind und
  - insgesamt acht einzelne Ein- und Ausgabe-Operationen parallel abzuwickeln sind.

mpi-jacobi-2d.cpp

```
auto jacobi = [&](std::size_t i, std::size_t j) {
   B(i, j) = 0.25 *
      (A(i - 1, j) + A(i + 1, j) + A(i, j - 1) + A(i, j + 1));
   T diff = std::fabs(A(i, j) - B(i, j));
   if (diff > maxdiff) maxdiff = diff;
};

/* compute the border first which is sent in advance
   to our neighbors */
for (std::size_t i = 1; i + 1 < B.numRows(); ++i) {
   jacobi(i, 1);
   jacobi(i, B.numCols()-2);
}
for (std::size_t j = 2; j + 2 < B.numCols(); ++j) {
   jacobi(1, j);
   jacobi(B.numRows()-2, j);
}
```

mpi-jacobi-2d.cpp

```
/* send border to our neighbors and
   initiate the receive operations */
MPI_Request requests[8]; int ri = 0;
for (auto& in_vector: in_vectors) {
   MPI_Irecv(in_vector.addr, 1, in_vector.datatype,
      in_vector.rank, 0, grid, &requests[ri++]);
}
for (auto& out_vector: out_vectors) {
   MPI_Isend(out_vector.addr, 1, out_vector.datatype,
      out_vector.rank, 0, grid, &requests[ri++]);
}

/* compute the inner block in parallel
   to the initiated communication */
for (std::size_t i = 2; i + 2 < B.numRows(); ++i) {
   for (std::size_t j = 2; j + 2 < B.numCols(); ++j) {
      jacobi(i, j);
   }
}
```

mpi-jacobi-2d.cpp

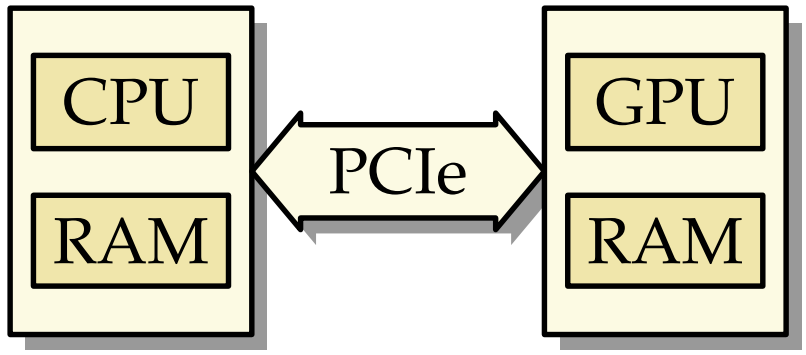
```
/* main loop for the Jacobi iterations */
T eps = 1e-6; unsigned int iterations;
for (iterations = 0; ; ++iterations) {
   T maxdiff = jacobi_iteration(B1, B2, B2_in_vectors, B2_out_vectors,
      grid);
   maxdiff = jacobi_iteration(B2, B1, B1_in_vectors, B1_out_vectors,
      grid);
   if (iterations % 10 == 0) {
      T global_max;
      MPI_Reduce(&maxdiff, &global_max, 1, get_type(maxdiff),
         MPI_MAX, 0, grid);
      MPI_Bcast(&global_max, 1, get_type(maxdiff), 0, grid);
      if (global_max < eps) break;
   }
}
if (rank == 0) fmt::printf("%d iterations\n", iterations);
```

mpi-jacobi-2d.cpp

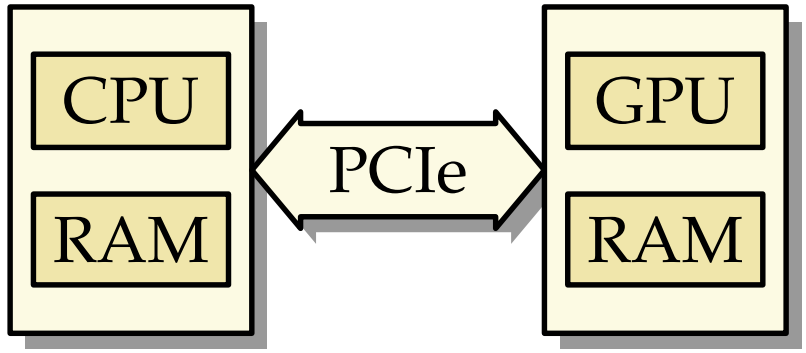
```
/* collect results */
gather_by_block(B1, A, 0, grid, overlap);
```

- Am Ende werden die Teilmatrizen mit *gather_by_block* zu einer Gesamtmatrix zusammengefügt, die dann wiederum in einen GDK-Pixbuf konvertiert und ausgegeben wird.

- Schon sehr früh gab es diverse Grafik-Beschleuniger, die der normalen CPU Arbeit abnahmen.
- Die im März 2001 von Nvidia eingeführte GeForce 3 Series führte programmierbares Shading ein.
- Im August 2002 folgte die Radeon R300 von ATI, die die Fähigkeiten der GeForce 3 deutlich erweiterte um mathematische Funktionen und Schleifen.
- Zunehmend werden die GPUs zu GPGPUs (*general purpose GPUs*).
- Zur generellen Nutzung wurden mehrere Sprachen und Schnittstellen entwickelt: OpenCL (Open Computing Language), DirectCompute (von Microsoft), CUDA (Compute Unified Device Architecture, von Nvidia) und OpenACC (*open accelerators*, von Cray, CAPS, Nvidia und PGI). Wir beschäftigen uns hier zunächst mit CUDA, da es recht populär ist und bei uns auch zur Verfügung steht.



- Überwiegend stehen GPUs als PCI-Express-Steckkarten zur Verfügung.
- Sie leben und arbeiten getrennt von der CPU und ihrem Speicher. Eine Kommunikation muss immer über die PCI-Express-Verbindung erfolgen.
- Meistens haben sie völlig eigenständige und spezialisierte Prozessorarchitekturen. Eine prominente Ausnahme ist die Larrabee-Mikroarchitektur von Intel, die sich an der x86-Architektur ausrichtete und der später die Xeon-Phi-Architektur folgte, die keine GPU mehr ist und nur noch dem High-Performance-Computing dient.



- Eine entsprechende Anwendung besteht normalerweise aus zwei Programmen (eines für die CPU und eines für die GPU).
- Das Programm für die GPU muss in die GPU heruntergeladen werden; anschließend sind die Daten über die PCI-Express-Verbindung auszutauschen.
- Die Kommunikation kann (mit Hilfe der zur Verfügung stehenden Bibliothek) sowohl synchron als auch asynchron erfolgen.

- Für die GPUs stehen höhere Programmiersprachen zur Verfügung, typischerweise Varianten, die auf C bzw. C++ basieren.
- Hierbei kommen Spracherweiterungen für GPUs hinzu, und andererseits wird C bzw. C++ nicht umfassend unterstützt. Insbesondere stehen außer wenigen mathematischen Funktionen keine Standard-Bibliotheken zur Verfügung.
- Nach dem Modell von CUDA sind die Programmtexte für die CPU und die GPU in der gleichen Quelle vermischt. Diese werden auseinandersortiert und dann getrennt voneinander übersetzt. Der entstehende GPU-Code wird dann als Byte-Array in den CPU-Code eingefügt. Zur Laufzeit wird dann nur noch der fertig übersetzte GPU-Code zur GPU geladen.
- Beim Modell von OpenCL sind die Programmtexte getrennt, wobei der für die GPU bestimmte Programmtext erst zur Laufzeit übersetzt und mit Hilfe von OpenCL-Bibliotheksfunktionen in die GPU geladen wird.
- Zunächst betrachten wir CUDA...

CUDA ist ein von Nvidia für Linux, MacOS und Windows kostenfrei zur Verfügung gestelltes Paket (jedoch nicht *open source*), das folgende Komponenten umfasst:

- einen Gerätetreiber,
- eine Spracherweiterung von C++ (CUDA C++), die es ermöglicht, in einem Programmtext die Teile für die CPU und die GPU zu vereinen,
- einen Übersetzer *nvcc* (zu finden im Verzeichnis */usr/local/cuda-10.1/bin* auf dem Rechner Livingstone, der CUDA C++ unterstützt,
- eine zugehörige Laufzeitbibliothek (*libcudart.so* in */usr/local/cuda-10.1/lib64* auf Livingstone und
- darauf aufbauende Bibliotheken (einschließlich BLAS und FFT).

URL: `https://developer.nvidia.com/cuda-downloads`

properties.cu

```
#include <cstdlib>
#include <iostream>

inline void check_cuda_error(const char* cudaop, const char* source,
       unsigned int line, cudaError_t error) {
   if (error != cudaSuccess) {
      std::cerr << cudaop << " at " << source << ":" << line
         << " failed: " << cudaGetErrorString(error) << std::endl;
      exit(1);
   }
}

#define CHECK_CUDA(opname, ...) \
   check_cuda_error(#opname, __FILE__, __LINE__, opname(__VA_ARGS__))

int main() {
   int device; CHECK_CUDA(cudaGetDevice, &device);
   // ...
}
```

- Bei CUDA-Programmen steht die CUDA-Laufzeit-Bibliothek zur Verfügung. Die Fehler-Codes aller CUDA-Bibliotheksaufrufe sollten jeweils überprüft werden.

properties.cu

```
int device_count; CHECK_CUDA(cudaGetDeviceCount, &device_count);
struct cudaDeviceProp device_prop;
CHECK_CUDA(cudaGetDeviceProperties, &device_prop, device);
if (device_count > 1) {
   std::cout << "device " << device << " selected out of "
      << device_count << " devices:" << std::endl;
} else {
   std::cout << "one device present:" << std::endl;
}
std::cout << "name: " << device_prop.name << std::endl;
std::cout << "compute capability: " << device_prop.major
   << "." << device_prop.minor << std::endl;
// ...
```

- *cudaGetDeviceProperties* füllt eine umfangreiche Datenstruktur mit den Eigenschaften einer der zur Verfügung stehenden GPUs.
- Es gibt eine Vielzahl von Nvidia-Karten mit sehr unterschiedlichen Fähigkeiten und Ausstattungen, so dass bei portablen Anwendungen diese u.U. zuerst überprüft werden müssen.
- Die Namen der CUDA-Programme enden normalerweise in „.cu“.

```
livingstone$ make
nvcc -o properties properties.cu
livingstone$ ./properties
one device present:
name: Quadro P620
compute capability: 6.1
total global memory: 2096103424
total constant memory: 65536
total shared memory per block: 49152
registers per block: 65536
L2 Cache Size: 524288
warp size: 32
mem pitch: 2147483647
max threads per block: 1024
max threads dim: 1024 1024 64
max grid dim: 2147483647 65535 65535
multi processor count: 4
kernel exec timeout enabled: no
device overlap: yes
integrated: no
can map host memory: yes
unified addressing: yes
livingstone$
```

```
livingstone$ make
nvcc -o properties properties.cu
```

- Übersetzt wird mit *nvcc*.
- Hierbei werden die für die GPU vorgesehenen Teile übersetzt und in Byte-Arrays verwandelt, die dann zusammen mit den für die CPU bestimmten Programmteilen dem *gcc* zur Übersetzung gegeben werden.
- Optionen wie „–gpu-architecture compute_60“ ermöglicht die Nutzung wichtiger später hinzu gekommener Erweiterungen.
- Wenn die Option „-code sm_60“ hinzukommt, wird beim Übersetzen auch der Code für die CPU erzeugt. Ansonsten geschieht dies bei neueren CUDA-Versionen erst zur Laufzeit im JIT-Verfahren.

simple.cu

```
__global__ void add(std::size_t len, double alpha, double* x, double* y) {
   for (std::size_t i = 0; i < len; ++i) {
      y[i] += alpha * x[i];
   }
}
```

- Funktionen (oder Methoden), die für die GPU bestimmt sind und von der CPU aus aufrufbar sind, werden Kernel-Funktionen genannt und mit dem CUDA-Schlüsselwort **__global__** gekennzeichnet.
- Die Funktion *add* in diesem Beispiel addiert den Vektor x multipliziert mit *alpha* zu y.
- Alle Zeiger verweisen hier auf GPU-Speicher.

simple.cu

```
  /* execute kernel function on GPU */
  add<<<1, 1>>>(N, 2.0, device_a, device_b);
```

- Eine Kernel-Funktion kann direkt von dem auf der CPU ausgeführten Programmtext aufgerufen werden.
- Zwischen dem Funktionsnamen und den Parametern wird in „<<<...>>>“ die Kernel-Konfiguration angegeben. Diese ist zwingend notwendig. Die einzelnen Konfigurationsparameter werden später erläutert.
- Elementare Datentypen können direkt übergeben werden (hier N und der Wert 2.0), Zeiger müssen aber bereits auf GPU-Speicher zeigen, d.h. dass ggf. Daten zuvor vom CPU-Speicher zum GPU-Speicher zu transferieren sind.

simple.cu

```
double a[N]; double b[N];
for (std::size_t i = 0; i < N; ++i) {
   a[i] = i; b[i] = i * i;
}

/* transfer vectors to GPU memory */
double* device_a;
CHECK_CUDA(cudaMalloc, (void**)&device_a, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, device_a, a, N * sizeof(double),
   cudaMemcpyHostToDevice);
double* device_b;
CHECK_CUDA(cudaMalloc, (void**)&device_b, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, device_b, b, N * sizeof(double),
   cudaMemcpyHostToDevice);
```

- Mit der CUDA-Bibliotheksfunktion *cudaMalloc* kann Speicher bei der GPU belegt werden.
- Es kann davon ausgegangen werden, dass alle Datentypen auf der CPU und der GPU in gleicher Weise repräsentiert werden. Anders als bei MPI kann die Übertragung ohne Konvertierungen erfolgen.

simple.cu

```
double a[N]; double b[N];
for (std::size_t i = 0; i < N; ++i) {
   a[i] = i; b[i] = i * i;
}

/* transfer vectors to GPU memory */
double* device_a;
CHECK_CUDA(cudaMalloc, (void**)&device_a, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, device_a, a, N * sizeof(double),
   cudaMemcpyHostToDevice);
double* device_b;
CHECK_CUDA(cudaMalloc, (void**)&device_b, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, device_b, b, N * sizeof(double),
   cudaMemcpyHostToDevice);
```

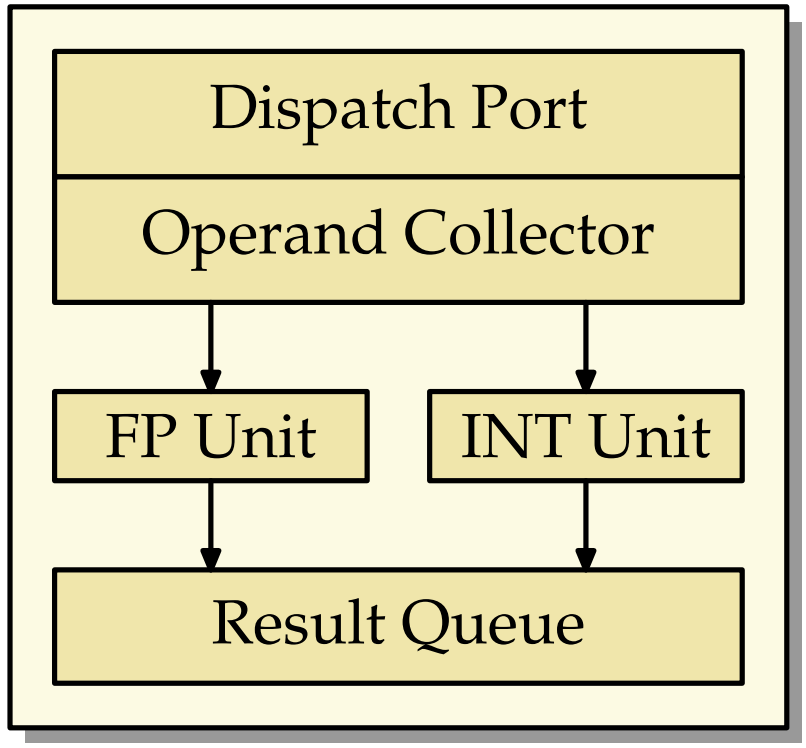
- Mit *cudaMemcpy* kann von CPU- in GPU-Speicher oder umgekehrt kopiert werden. Zuerst kommt (wie bei *memcpy*) das Ziel, dann die Quelle, dann die Zahl der Bytes. Zuletzt wird die Richtung angegeben.
- Ohne den letzten Parameter weiß *cudaMemcpy* nicht, um was für Zeiger es sich handelt.

simple.cu

```
/* transfer result vector from GPU device to host memory */
CHECK_CUDA(cudaMemcpy, b, device_b, N * sizeof(double),
   cudaMemcpyDeviceToHost);
/* free space allocated at GPU memory */
CHECK_CUDA(cudaFree, device_a); CHECK_CUDA(cudaFree, device_b);
```

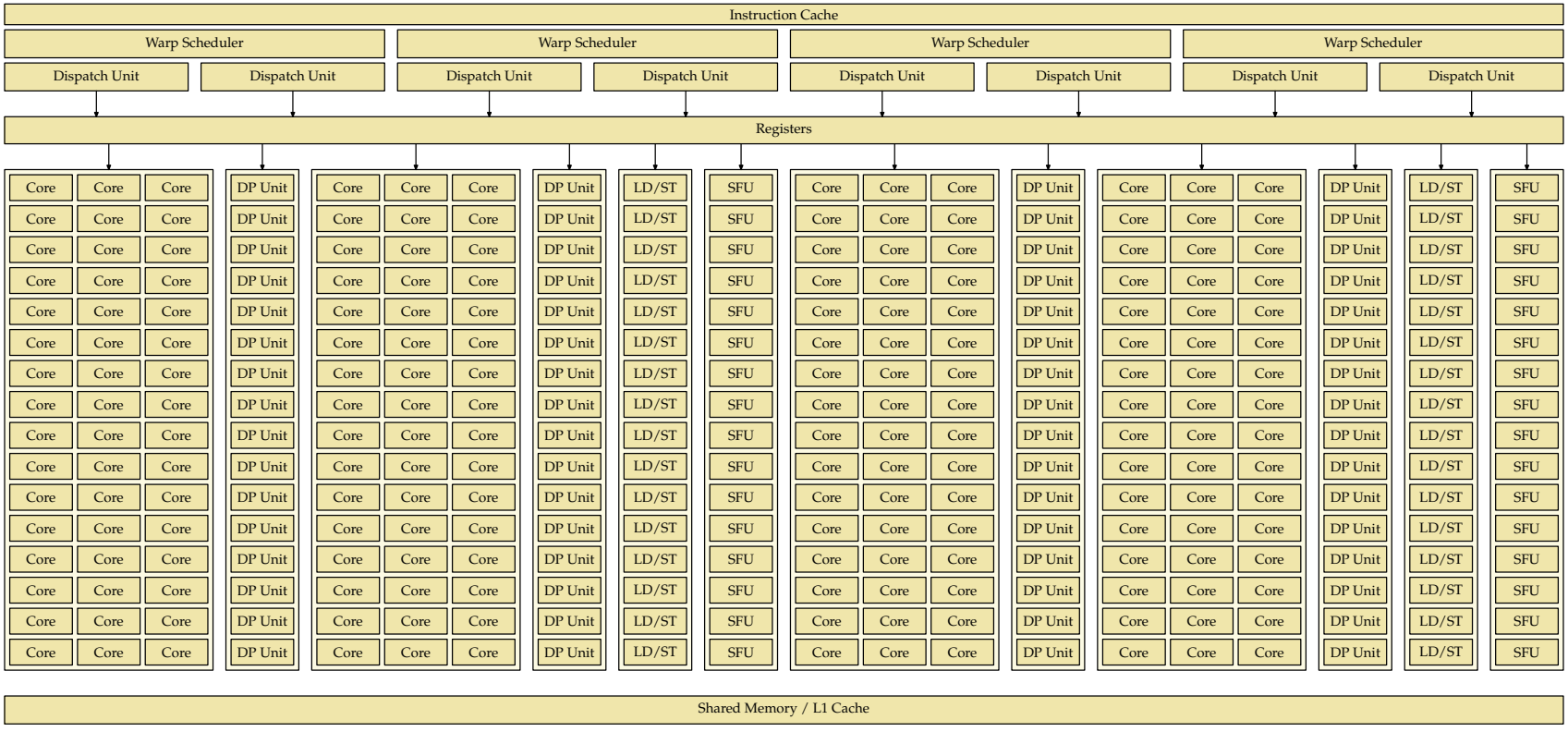
- Nach dem Aufruf der Kernel-Funktion kann das Ergebnis zurückkopiert werden.
- Kernel-Funktionen laufen asynchron, d.h. die weitere Ausführung des Programms auf der CPU läuft parallel zu der Verarbeitung auf der GPU. Sobald die Ergebnisse jedoch zurückkopiert werden, findet implizit eine Synchronisierung statt, d.h. es wird auf die Beendigung der Kernel-Funktion gewartet.
- Wenn der Kernel nicht gestartet werden konnte oder es zu Problemen während der Ausführung kam, wird dies über die Fehler-Codes bei der folgenden synchronisierenden CUDA-Operation mitgeteilt.
- Anschließend sollte der GPU-Speicher freigegeben werden, wenn er nicht mehr benötigt wird.

- Das erste Beispiel zeigt den typischen Ablauf vieler Anwendungen:
  - Speicher auf der GPU belegen
  - Daten zur GPU transferieren
  - Kernel-Funktion aufrufen
  - Ergebnisse zurücktransferieren
  - GPU-Speicher freigeben
- Eine echte Parallelisierung hat das Beispiel nicht gebracht, da die CPU auf die GPU wartete und auf der GPU nur ein einziger GPU-Kern aktiv war...



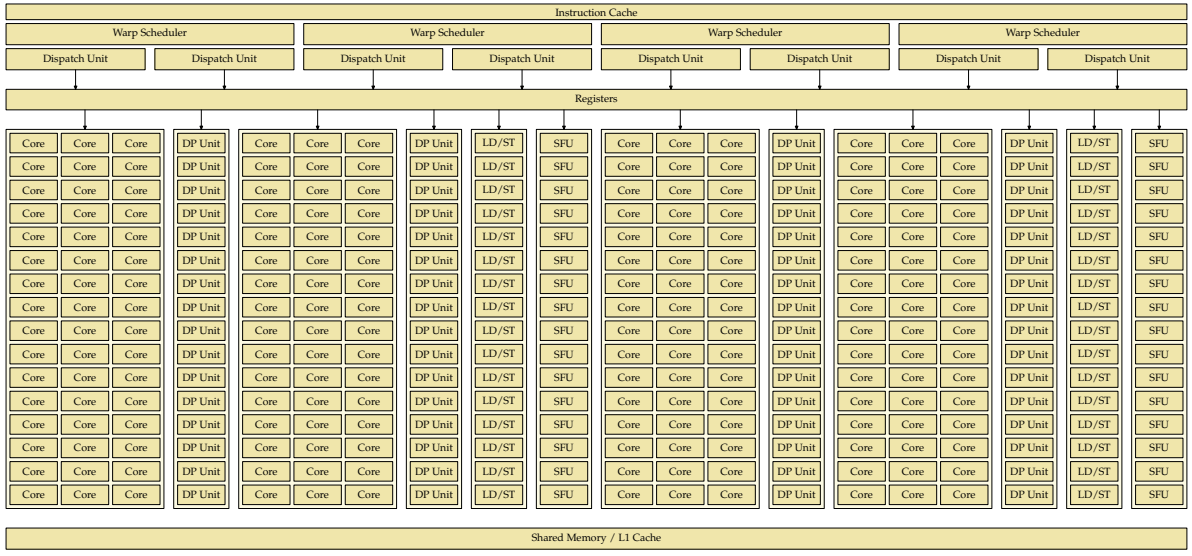
- Die elementaren Recheneinheiten einer GPU bestehen im wesentlichen nur aus zwei Komponenten, jeweils eine für arithmetische Operationen für Gleitkommazahlen und eine für ganze Zahlen.
- Mehr Teile hat ein GPU-Kern nicht. Die Instruktion und die Daten werden angeliefert (*Dispatch Port* und *Operand Collector*), und das Resultat wird bei der *Result Queue* abgeliefert.

Multiprozessor einer GPU



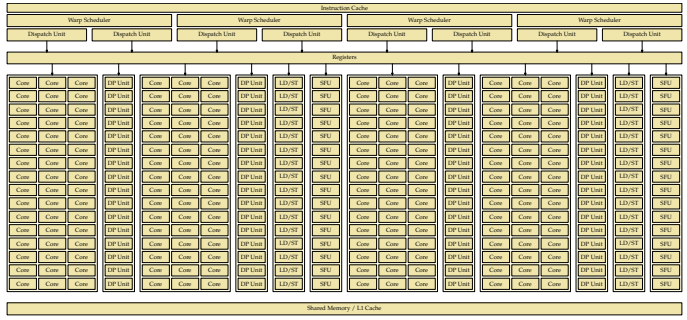
- Dieses Diagramm zeigt einen einzelnen Multiprozessor der Kepler-Mikroarchitektur mit 192 CUDA-Kernen, die mit einfacher Genauigkeit arbeiten („Core“) und 64 Kerne, die mit doppelter Genauigkeit operieren („DP Unit“).

Multiprozessor einer GPU



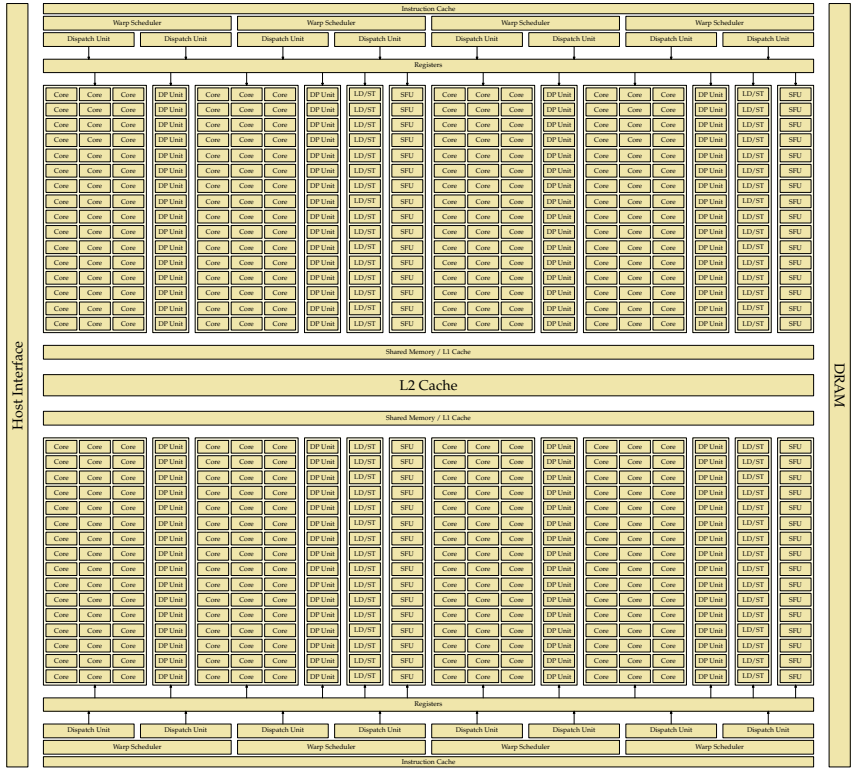
- Auf der Livingstone haben wir die Pascal-Mikroarchitektur, wobei jeder Multiprozessor 128 CUDA-Kerne besitzt.
- Hinzu kommen Einheiten zum parallelisierten Laden und Speichern (*LD/ST*) und Einheiten für spezielle Operationen wie beispielsweise *sin* oder *sqrt* (*SFU*).
- (Abgebildet ist hier die Kepler-Architektur.)

- Die Kerne operieren nicht unabhängig voneinander.
- Im Normalfall werden 32 Kerne zu einem Warp zusammengefasst. (Unterstützt werden auch halbe Warps mit 16 Kernen.)
- Alle Kerne eines Warps führen synchron die gleiche Instruktion aus – auf unterschiedlichen Daten (SIMD-Architektur: Array-Prozessor).
- Dabei werden jeweils zunächst die zu ladenden Daten parallel organisiert (durch die *LD/ST*-Einheiten) und dann über die Register den einzelnen Kernen zur Verfügung gestellt.



- Jeder Warp-Scheduler hat die Kontrolle über einen Warp mit 32 Threads.
- Die Kepler-Architektur bietet einen Warp-Scheduler mit zwei Dispatch-Units an, wobei jeder von diesen in der Lage ist, eine Instruktion für eine Gruppe von CUDA-Kernen oder speziellen Einheiten weiterzugeben in Abhängigkeit von der aktuellen Instruktion.
- Entsprechend können zwei Instruktionen bei einem Thread parallel ausgeführt werden.
- Instruktionen mit doppelter Genauigkeit müssen zweifach aussgeführt werden, je einmal für einen halben Warp mit 16 Threads, da jedem Warp-Scheduler nur 16 Kerne mit doppelter Genauigkeit zur Verfügung stehen.
- Jeder Kepler-Multiprozessor ist mit vier Warp-Schedulern ausgestattet.

GPU mit mehreren Multiprozessoren



- Je nach Ausführung der GPU werden mehrere Multiprozessoren zusammengefasst.
- Livingstone bietet 4 Multiprozessoren mit insgesamt 512 CUDA-Kernen an. Auf meinem iMac sind es zwei Multiprozessoren mit insgesamt 384 CUDA-Kernen (abgebildet).

- Der Instruktionssatz der Nvidia-GPUs ist proprietär und bis heute wurde abgesehen von einer kleinen Übersicht von Nvidia kein umfassendes öffentliches Handbuch dazu herausgegeben.
- Mit Hilfe des Werkzeugs *cuobjdump* lässt sich der Code disassemblieren und ansehen.
- Die Instruktionen haben entweder einen Umfang von 32 oder 64 Bits. 64-Bit-Instruktionen sind auf 64-Bit-Kanten.
- Arithmetische Instruktionen haben bis zu drei Operanden und ein Ziel, bei dem das Ergebnis abgelegt wird. Beispiel ist etwa eine Instruktion, die in einfacher Genauigkeit $d = a * b + c$ berechnet (FMAD).

Wie können bedingte Sprünge umgesetzt werden, wenn ein Warp auf eine if-Anweisung stößt und die einzelnen Threads des Warps unterschiedlich weitermachen wollen? (Zur Erinnerung: Alle Threads eines Warps führen immer die gleiche Instruktion aus.)

- Es stehen zwei Stacks zur Verfügung:
- Ein Stack mit Masken, bestehend aus 32 Bits, die festlegen, welche der 32 Threads die aktuellen Instruktionen ausführen.
- Ferner gibt es noch einen Stack mit Zieladressen.
- Bei einer bedingten Verzweigung legt jeder der Threads in der Maske fest, ob die folgenden Instruktionen ihn betreffen oder nicht. Diese Maske wird auf den Stack der Masken befördert.
- Die Zieladresse des Sprungs wird auf den Stack der Zieladressen befördert.
- Wenn die Zieladresse erreicht wird, wird auf beiden Stacks das oberste Element jeweils entfernt.

- Ein Block ist eine Abstraktion, die mehrere Warps zusammenfasst.
- Bei CUDA-Programmen werden Blöcke konfiguriert, die dann durch den jeweiligen Warp-Scheduler auf einzelne Warps aufgeteilt werden, die sukzessive zur Ausführung kommen.
- Ein Block läuft immer nur auf einem Multiprozessor bietet einen gemeinsamen Speicherbereich für alle zugehörigen Threads an.
- Threads eines Blockes können sich untereinander synchronisieren und über den gemeinsamen Speicher kommunizieren.
- Ein Block kann (bei uns auf Livingstone) bis zu 32 Warps bzw. 1024 Threads umfassen.

vector.cu

```
__global__ void add(double alpha, double* x, double* y) {
   std::size_t tid = threadIdx.x;
   y[tid] += alpha * x[tid];
}
```

- Die **for**-Schleife ist entfallen. Stattdessen führt die Kernel-Funktion jetzt nur noch eine einzige Addition durch.
- Mit *threadIdx.x* erfahren wir hier, der wievielte Thread (beginnend ab 0) unseres Blocks wir sind.
- Die Kernel-Funktion wird dann für jedes Element aufgerufen, wobei dies im Rahmen der Möglichkeiten parallelisiert wird.

vector.cu

```
/* execute kernel function on GPU */
add<<<1, N>>>(2.0, device_a, device_b);
```

- Beim Aufruf der Kernel-Funktion wurde jetzt die Konfiguration verändert.
- $<<<1, N>>>$ bedeutet jetzt, dass ein Block mit N Threads gestartet wird.
- Entsprechend starten die einzelnen Threads mit Werten von 0 bis $N-1$ für *threadIdx.x*.
- Da die Zahl der Threads pro Block beschränkt ist, kann N hier nicht beliebig groß werden.

vector.ptx

```
        ld.param.f64    %fd1, [_Z3adddPdS__param_0];
        ld.param.u64    %rd1, [_Z3adddPdS__param_1];
        ld.param.u64    %rd2, [_Z3adddPdS__param_2];
        cvta.to.global.u64      %rd3, %rd2;
        cvta.to.global.u64      %rd4, %rd1;
        mov.u32         %r1, %tid.x;
        mul.wide.u32    %rd5, %r1, 8;
        add.s64         %rd6, %rd4, %rd5;
        ld.global.f64   %fd2, [%rd6];
        add.s64         %rd7, %rd3, %rd5;
        ld.global.f64   %fd3, [%rd7];
        fma.rn.f64      %fd4, %fd2, %fd1, %fd3;
        st.global.f64   [%rd7], %fd4;
        ret;
```

- PTX steht für *Parallel Thread Execution* und ist eine Assembler-Sprache für einen virtuellen GPU-Prozessor. Dies ist die erste Zielsprache des Übersetzers für den für die GPU bestimmten Teil.

- PTX steht für *Parallel Thread Execution* und ist eine Assembler-Sprache für einen virtuellen GPU-Prozessor. Dies ist die erste Zielsprache des Übersetzers für den für die GPU bestimmten Teil.
- Der PTX-Instruktionssatz ist öffentlich:
  `http://docs.nvidia.com/cuda/pdf/ptx_isa_6.1.pdf`
- PTX wurde entwickelt, um eine portable vom der jeweiligen Grafikkarte unabhängige virtuelle Maschine zu haben, die ohne größeren Aufwand effizient für die jeweiligen GPUs weiter übersetzt werden kann.
- PTX lässt sich mit Hilfe des folgenden Kommandos erzeugen:
  `nvcc -o vector.ptx --ptx --gpu-architecture compute_60 vector.cu`

vector.disas

```
/*0008*/                   MOV R1, c[0x0][0x20] ;
/*0010*/                   S2R R0, SR_TID.X ;
/*0018*/                   SHL R4, R0.reuse, 0x3 ;
/*0028*/                   SHR.U32 R0, R0, 0x1d ;
/*0030*/                   IADD R6.CC, R4, c[0x0][0x148] ;
/*0038*/                   IADD.X R7, R0, c[0x0][0x14c] ;
/*0048*/         {         IADD R4.CC, R4, c[0x0][0x150] ;
/*0050*/                   LDG.E.64 R2, [R6]          }
/*0058*/                   IADD.X R5, R0, c[0x0][0x154] ;
/*0068*/                   LDG.E.64 R8, [R4] ;
/*0070*/                   DFMA R2, R2, c[0x0][0x140], R8 ;
/*0078*/                   STG.E.64 [R4], R2 ;
/*0088*/                   EXIT ;
```

- Mit Hilfe von *ptxas* kann der PTX-Text für eine konkrete GPU in Maschinen-Code übersetzt werden (CUBIN):
  `ptxas --output-file vector.cubin --gpu-name sm_60 vector.ptx`

- In neueren Versionen stellt Nvidia mit *cuobjdump* ein Werkzeug zur Verfügung, der den CUBIN-Code wieder disassembliert:
  `cuobjdump --dump-sass vector.cubin >vector.disas`
- Bei uns auf Livingstone ist in der zugehörigen Dokumentation die Übersicht zu Pascal relevant.

bigvector.cu

```
unsigned int max_threads_per_block() {
   int device; CHECK_CUDA(cudaGetDevice, &device);
   struct cudaDeviceProp device_prop;
   CHECK_CUDA(cudaGetDeviceProperties, &device_prop, device);
   return device_prop.maxThreadsPerBlock;
}
```

- Wenn die Zahl der Elemente größer ist als die maximale Zahl der Threads pro Block, wird es notwendig, die Aufgabe auf mehrere Blöcke aufzusplitten.

bigvector.cu

```
/* execute kernel function on GPU */
size_t max_threads = hpc::cuda::get_max_threads_per_block();
if (N <= max_threads) {
   add<<<1, N>>>(2.0, device_a, device_b);
} else {
   add<<<(N+max_threads-1)/max_threads, max_threads>>>(2.0,
      device_a, device_b);
}
```

bigvector.cu

```
__global__ void add(double alpha, double* x, double* y) {
   std::size_t tid = threadIdx.x + blockIdx.x * blockDim.x;
   if (tid < N) {
      y[tid] += alpha * x[tid];
   }
}
```

- *threadIdx.x* gibt jetzt nur noch den Thread-Index innerhalb eines Blocks an und ist somit alleine nicht mehr geeignet, die eindeutige Thread-ID im Bereich von 0 bis $N-1$ zu bestimmen.
- *blockIdx.x* liefert jetzt zusätzlich noch den Block-Index und *blockDim.x* die Zahl der Threads pro Block.
- Wenn N nicht durch *max_threads* teilbar ist, dann erhalten wir Threads, die nichts zu tun haben. Um einen Zugriff außerhalb der Array-Grenzen zu vermeiden, benötigen wir eine entsprechende **if**-Anweisung.

In der allgemeinen Form akzeptiert die Konfiguration vier Parameter. Davon sind die beiden letzten optional:
$<<<$ *Dg*, *Db*, *Ns*, *S* $>>>$

- *Dg* legt die Dimensionierung des Grids fest (ein- oder zwei- oder dreidimensional).
- *Db* legt die Dimensionierung eines Blocks fest (ein-, zwei- oder dreidimensional).
- *Ns* legt den Umfang des gemeinsamen Speicherbereichs per Block fest (per Voreinstellung 0 bzw. vom Übersetzer bestimmt).
- *S* erlaubt die Verknüpfung mit einem Stream (per Voreinstellung keine).
- *Dg* und *Db* sind beide vom Typ *dim3*, der mit eins bis drei ganzen Zahlen initialisiert werden kann.
- Vorgegebene Beschränkungen sind bei der Dimensionierung zu berücksichtigen. Sonst kann der Kernel nicht gestartet werden.

In den auf der GPU laufenden Funktionen stehen spezielle Variablen zur Verfügung, die die Identifizierung bzw. Einordnung des eigenen Threads ermöglichen:

threadIdx.x	x-Koordinate innerhalb des Blocks
threadIdx.y	y-Koordinate innerhalb des Blocks
threadIdx.z	z-Koordinate innerhalb des Blocks
blockDim.x	Dimensionierung des Blocks für x
blockDim.y	Dimensionierung des Blocks für y
blockDim.z	Dimensionierung des Blocks für z
blockIdx.x	x-Koordinate innerhalb des Gitters
blockIdx.y	y-Koordinate innerhalb des Gitters
blockIdx.z	z-Koordinate innerhalb des Gitters
gridDim.x	Dimensionierung des Gitters für x
gridDim.y	Dimensionierung des Gitters für y
gridDim.z	Dimensionierung des Gitters für z

simpson.cu

```
// to be integrated function
__device__ double f(double x) {
   return 4 / (1 + x*x);
}

// numerical integration according to the Simpson rule
// for f over the i-th subinterval of [a,b]
__global__ void simpson(double a, double b, double* sums) {
   const std::size_t N = blockDim.x * gridDim.x;
   const std::size_t i = threadIdx.x + blockIdx.x * blockDim.x;
   double xleft = a + (b - a) / N * i;
   double xright = xleft + (b - a) / N;
   double xmid = (xleft + xright) / 2;
   sums[i] = (xright - xleft) / 6 * (f(xleft) + 4 * f(xmid) + f(xright));
}
```

- Die Kernel-Funktion berechnet hier jeweils nur ein Teilintervall und ermittelt mit Hilfe von $blockDim.x * gridDim.x$, wieviel Teilintervalle es gibt.
- Funktionen, die auf der GPU nur von anderen GPU-Funktionen aufzurufen sind, werden mit dem Schlüsselwort **__device__** gekennzeichnet.

simpson.cu

```
double* device_sums;
CHECK_CUDA(cudaMalloc, (void**)&device_sums, N * sizeof(double));
simpson<<<nof_blocks, blocksize>>>(a, b, device_sums);

double sums[N];
CHECK_CUDA(cudaMemcpy, sums, device_sums,
   N * sizeof(double), cudaMemcpyDeviceToHost);
CHECK_CUDA(cudaFree, device_sums);
double sum = 0;
for (std::size_t i = 0; i < N; ++i) {
   sum += sums[i];
}
```

- Es gibt keine vorgegebenen Aggregierungs-Operatoren, so dass diese „von Hand“ durchgeführt werden müssen.

simpson2.cu

```
constexpr std::size_t THREADS_PER_BLOCK = 1024; // must be a power of 2

// numerical integration according to the Simpson rule
// for f over the i-th subinterval of [a,b]
__global__ void simpson(double a, double b, double* sums) {
   /* compute approximative sum for our sub-interval */
   const std::size_t N = blockDim.x * gridDim.x;
   const std::size_t i = threadIdx.x + blockIdx.x * blockDim.x;
   // ...
   double sum = (xright - xleft) / 6 * (f(xleft) +
      4 * f(xmid) + f(xright));

   /* store it into the per-block shared array of sums */
   std::size_t me = threadIdx.x;
   __shared__ double sums_per_block[THREADS_PER_BLOCK];
   sums_per_block[me] = sum;

   // ...
}
```

- Innerhalb eines Blocks ist eine Synchronisierung und die Nutzung eines gemeinsamen Speicherbereiches möglich.
- Das eröffnet die Möglichkeit der blockweisen Aggregierung.

simpson2.cu

```
/* store it into the per-block shared array of sums */
unsigned int me = threadIdx.x;
__shared__ double sums_per_block[THREADS_PER_BLOCK];
sums_per_block[me] = sum;
```

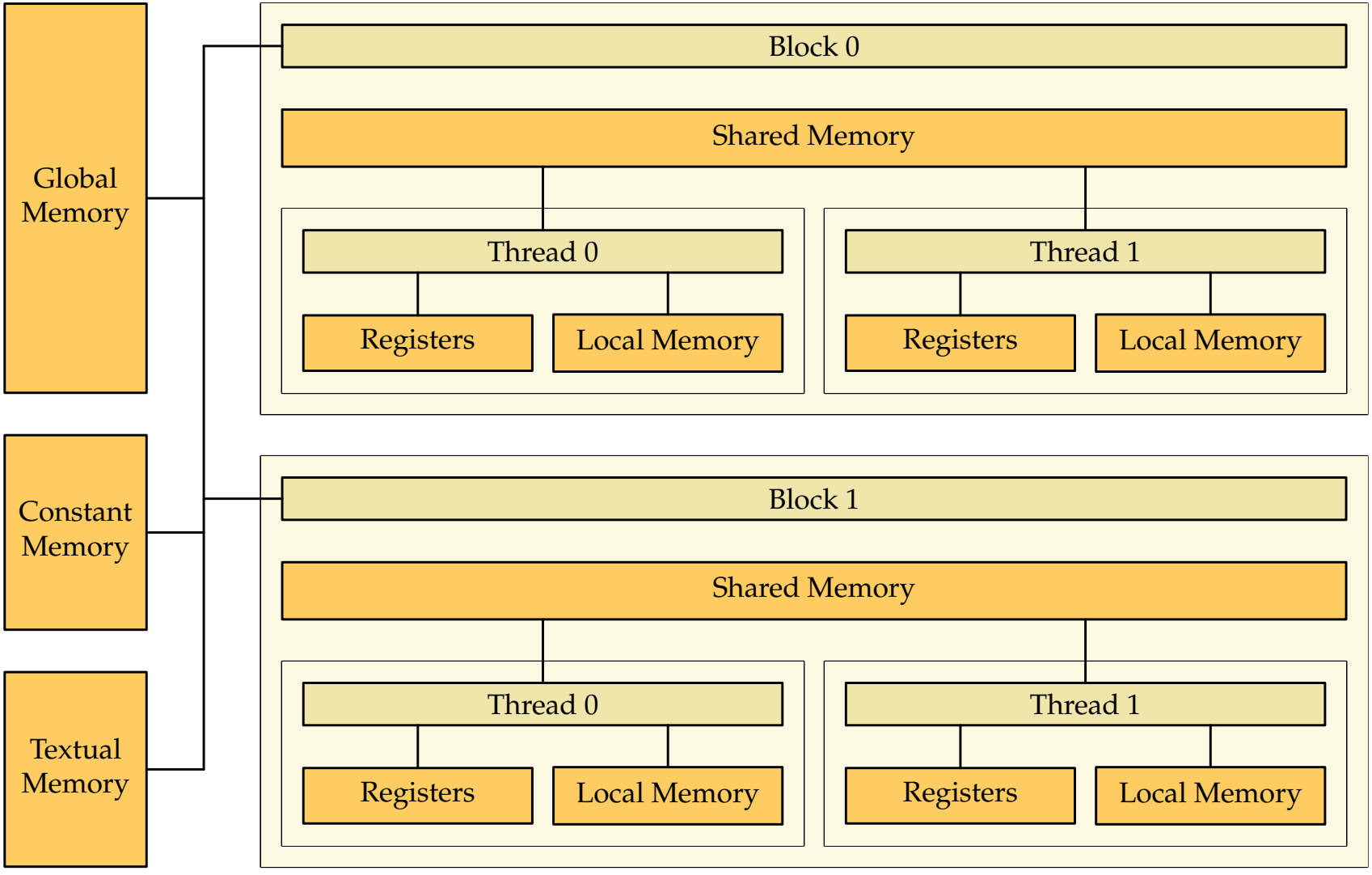
- Mit **__shared__** können Variablen gekennzeichnet werden, die allen Threads eines Blocks gemeinsam zur Verfügung stehen.
- Die Zugriffe auf diese Bereiche sind schneller als auf den globalen GPU-Speicher.
- Allerdings ist die Kapazität begrenzt. Auf Livingstone stehen nur 48 KiB zur Verfügung.

simpson2.cu

```
/* aggregate sums within a block */
std::size_t index = blockDim.x / 2;
while (index) {
   __syncthreads();
   if (me < index) {
      sums_per_block[me] += sums_per_block[me + index];
   }
   index /= 2;
}
/* publish result */
if (me == 0) {
   sums[blockIdx.x] = sums_per_block[0];
}
```

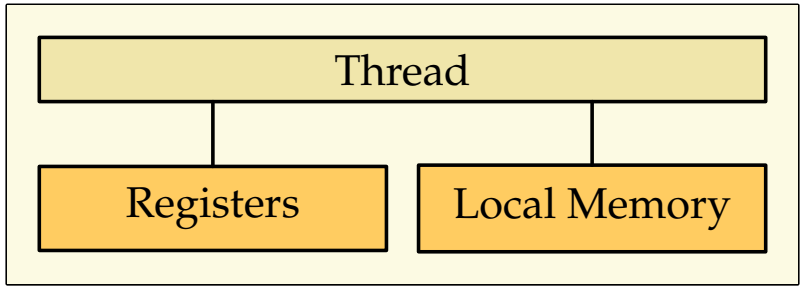
- Zwar werden die jeweils einzelnen Gruppen eines Blocks zu Warps zusammengefasst, die entsprechend der SIMD-Architektur synchron laufen, aber das umfasst nicht den gesamten Block.
- Wenn alle Threads eines globalen Blocks synchronisiert werden sollen, geht dies mit der Funktion *__syncthreads*, die den aufrufenden Thread solange blockiert, bis alle Threads des Blocks diese Funktion aufrufen.

Speicherbereiche einer GPU

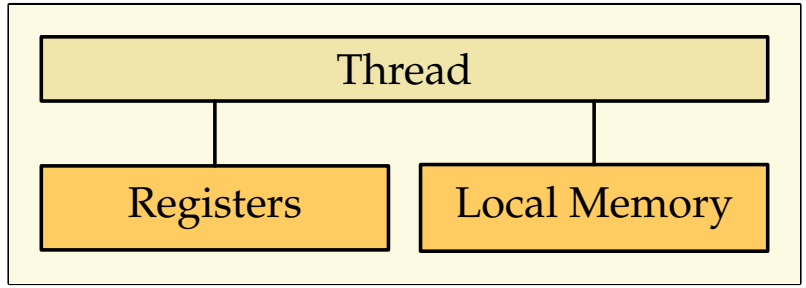


```
livingstone$ make
nvcc -o simpson -I/home/numerik/pub/pp/ss19/lib --ptxas-options=-
v simpson.cu
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z7simpsonddPd' for 'sm_30'
ptxas info    : Function properties for _Z7simpsonddPd
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 26 registers, 344 bytes cmem[0], 40 bytes cmem[2]
livingstone$
```

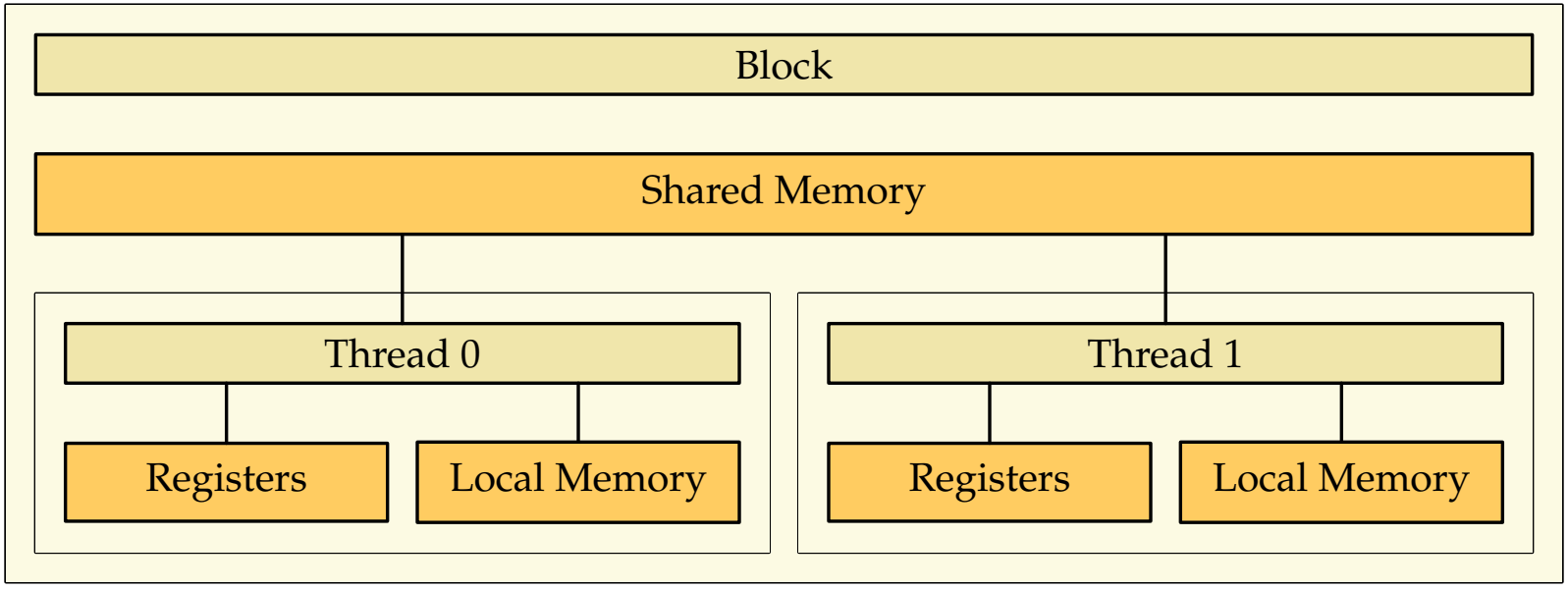
- *ptxas* dokumentiert den Verbrauch der einzelnen Speicherbereiche für eine Kernel-Funktion, wenn die entsprechende Verbose-Option gegeben wird.
- *gmem* steht hier für *global memory*, *cmem* für *constant memory*, das in Abhängigkeit der jeweiligen GPU-Architektur in einzelne Bereiche aufgeteilt wird.
- Lokaler Speicher wird verbraucht durch das *stack frame* und das Sichern von Registern (*spill stores*). Die *spill stores* und *spill loads* werden aber nur statisch an Hand des erzeugten Codes gezählt.



- 32-Bit-Register gibt es für ganzzahlige Datentypen oder Gleitkommazahlen.
- Lokale Variablen innerhalb eines Threads werden soweit wie möglich in Registern abgelegt.
- Wenn sehr viel Register benötigt werden, kann dies dazu führen, dass weniger Threads in einem Block zur Verfügung stehen als das maximale Limit angibt.
- Die Livingstone bietet beispielsweise 65536 Register per Block. Wenn das Maximum von 1024 Threads pro Block ausgeschöpft wird, verbleiben nur 64 Register für jeden Thread.



- Für den lokalen Speicher wird tatsächlich globaler Speicher verwendet.
- Es gibt allerdings spezielle cache-optimierte Instruktionen für das Laden und Speichern von und in den lokalen Speicher. Dies lässt sich optimieren, weil das Problem der Cache-Kohärenz wegfällt.
- Normalerweise erfolgen Lese- und Schreibzugriffe entsprechend nur aus bzw. in den L1-Cache. Wenn jedoch Zugriffe auf globalen Speicher notwendig werden, dann ist dies um ein Vielfaches langsamer als der gemeinsame Speicherbereich.
- Benutzt wird der lokale Speicher für den Stack, wenn die Register ausgehen und für lokale Arrays, bei denen Indizes zur Laufzeit berechnet werden.



- Nach den Registern bietet der für jeweils einen Block gemeinsame Speicher die höchste Zugriffsgeschwindigkeit.
- Die Kapazität ist sehr begrenzt. Auf Livingstone stehen nur 48 KiB per Block zur Verfügung, insgesamt jedoch 96 KiB per Multiprozessor (bei dem möglicherweise mehrere Blöcke parallel laufen).

- Der gemeinsame Speicher ist zyklisch in Bänke (*banks*) aufgeteilt. Das erste Wort im gemeinsamen Speicher (32 Bit) gehört zur ersten Bank, das zweite Wort zur zweiten Bank usw. Auf Livingstone gibt es 32 solcher Bänke. Das 33. Wort gehört dann wieder zur ersten Bank.
- Zugriffe eines Warps auf unterschiedliche Bänke erfolgen gleichzeitig. Zugriffe auf die gleiche Bank müssen serialisiert werden, wobei je nach Architektur Broad- und Multicasts möglich sind, d.h. ein Wort kann gleichzeitig an alle oder mehrere Threads eines Warps verteilt werden.

- Der globale Speicher ist für alle Threads und (mit Hilfe von *cudaMemcpy*) auch von der CPU aus zugänglich.
- Anders als beim regulären Hauptspeicher findet bei dem globalen GPU-Speicher kein Paging statt. D.h. es gibt nicht virtuell mehr Speicher als physisch zur Verfügung steht.
- Auf Livingstone stehen 1,95 GiB zur Verfügung.
- Zugriffe erfolgen über L1 und L2, wobei (bei unseren GPUs) Cache-Kohärenz nur über den globalen L2-Cache hergestellt wird, d.h. Schreib-Operationen schlagen sofort auf den L2 durch.
- Der L2 hat auf Livingstone 512 KiB.
- Globale Variablen können mit **__global__** deklariert werden oder dynamisch belegt wereden.

Zugriffe auf globalen Speicher sind bei älteren GPU-Architekturen unter den folgenden Bedingungen schnell:

- Der Zugriff erfolgt auf Worte, die mindestens 32 Bit groß sind.
- Die Zugriffsadressen müssen aufeinanderfolgend sein entsprechend der Thread-IDs innerhalb eines Blocks.
- Das erste Wort muss auf einer passenden Speicherkante liegen:

Wortgröße	Speicherkante
32 Bit	64 Byte
64 Bit	128 Byte
128 Bit	256 Byte

Bei der Pascal-Architektur (bei uns auf Livingstone) erfolgen die Zugriffe normalerweise nur über den L2, für Zugriffe durch den L1 müssen spezielle Übersetzungsoptionen gewählt werden.

- Die Cache-Lines bei L1 und L2 betragen jeweils 128 Bytes. (Entsprechend ergibt sich ein Alignment von 128 Bytes.)
- Eine Cache-Line besteht aus 4 Segmenten zu je 32 Bytes, bei einem Cache-Miss werden Segmente gefüllt, jedoch nicht zwangsläufig die gesamte Cache-Line.
- Wenn die gewünschten Daten im L1 liegen, dann kann innerhalb einer Transaktion eine Cache-Line mit 128 Bytes übertragen werden.
- Wenn die Daten nicht im L1, jedoch im L2 liegen, dann können per Transaktion 32 Byte übertragen werden.
- Die Restriktion, dass die Zugriffe konsekutiv entsprechend der Thread-ID erfolgen müssen, damit es effizient abläuft, entfällt. Es kommt nur noch darauf an, dass alle durch einen Warp gleichzeitig erfolgenden Zugriffe in eine Cache-Line passen.

- Wird von dem Übersetzer verwendet (u.a. für die Parameter der Kernel-Funktion) und es sind auch eigene Deklarationen mit dem Schlüsselwort **__constant__** möglich.
- Auf Livingstone stehen hierfür 64 KiB zur Verfügung.
- Die Zugriffe erfolgen optimiert, weil keine Cache-Kohärenz gewährleistet werden muss.
- Schreibzugriffe sind zulässig, aber innerhalb eines Kernels wegen der fehlenden Cache-Kohärenz nicht sinnvoll.

tracer.cu

```
__constant__ char sphere_storage[sizeof(Sphere)*SPHERES];
```

- Variablen im konstanten Speicher werden mit **___constant___** deklariert.
- Datentypen mit nicht-leeren Konstruktoren oder Destruktoren werden in diesem Bereich jedoch nicht unterstützt, so dass hier nur die entsprechende Fläche reserviert wird.
- Mit *cudaMemcpyToSymbol* kann dann von der CPU eine Variable im konstanten Speicher beschrieben werden.

tracer.cu

```
Sphere host_spheres[SPHERES];
// fill host_spheres...
// copy spheres to constant memory
CHECK_CUDA(cudaMemcpyToSymbol, sphere_storage,
   host_spheres, sizeof(host_spheres));
```

- Matrix-Matrix-Multiplikationen sind hochgradig parallelisierbar.
- Bei der Berechung von $C \leftarrow \alpha AB + \beta C$ kann beispielsweise die Berechnung von $c_{i,j}$ an einen einzelnen Thread delegiert werden.
- Da größere Matrizen nicht mehr in einen Block (mit bei uns maximal 1024 Threads) passen, ist es sinnvoll, die gesamte Matrix in Blocks zu zerlegen.
- Dazu bieten sich 16×16 Blöcke mit 256 Threads an.

gemm1.hpp

```
template<
   typename Alpha, typename Beta,
   template<typename> class MatrixA,
   template<typename> class MatrixB,
   template<typename> class MatrixC,
   typename T,
   Require<
      DeviceGe<MatrixA<T>>, DeviceView<MatrixA<T>>,
      DeviceGe<MatrixB<T>>, DeviceView<MatrixB<T>>,
      DeviceGe<MatrixC<T>>, DeviceView<MatrixC<T>>,
   > = true
>
__global__ void gemm_kernel(const Alpha alpha,
      const MatrixA<T> A,  const MatrixB<T> B,
      const Beta beta, MatrixC<T> C) {
   std::size_t i = threadIdx.x + blockIdx.x * blockDim.x;
   std::size_t j = threadIdx.y + blockIdx.y * blockDim.y;

   if (i < C.numRows() && j < C.numCols()) {
      T sum{};
      for (std::size_t k = 0; k < A.numCols(); ++k) {
         sum += A(i, k) * B(k, j);
      }
      sum *= alpha;
      sum += beta * C(i, j);
      C(i, j) = sum;
   }
}
```

gemm1.hpp

```
std::size_t i = threadIdx.x + blockIdx.x * blockDim.x;
std::size_t j = threadIdx.y + blockIdx.y * blockDim.y;

if (i < C.numRows() && j < C.numCols()) {
   T sum{};
   for (std::size_t k = 0; k < A.numCols(); ++k) {
      sum += A(i, k) * B(k, j);
   }
   sum *= alpha;
   sum += beta * C(i, j);
   C(i, j) = sum;
}
```

- Dies ist die triviale Implementierung, bei der jeder Thread $c_{row,col}$ direkt berechnet.
- Die Threads eines Warps bzw. Half-Warps unterscheiden sich nur durch *threadIdx.x*, haben *threadIdx.y* jedoch gemeinsam.
- Der Zugriff auf A und möglicherweise auch auf B und C sind hier ineffizient, da ein Warp bzw. Half-Warp nicht auf konsekutiv im Speicher liegende Werte zugreift.

gemm1.hpp

```
template<
   typename Alpha, typename Beta,
   template<typename> class MatrixA,
   template<typename> class MatrixB,
   template<typename> class MatrixC,
   typename T,
   Require<
      DeviceGe<MatrixA<T>>,
      DeviceGe<MatrixB<T>>,
      DeviceGe<MatrixC<T>>
   > = true
>
void cuda_gemm(const Alpha alpha,
      const MatrixA<T>& A,  const MatrixB<T>& B,
      const Beta beta, MatrixC<T>& C) {
   assert(A.numRows() == C.numRows() && A.numCols() == B.numRows() &&
      B.numCols() == C.numCols());
   constexpr std::size_t blockdim = 16;
   dim3 block(blockdim, blockdim);
   std::size_t M = C.numRows();
   std::size_t N = C.numCols();
   std::size_t K = A.numCols();
   using namespace hpc::aux;
   dim3 grid(ceildiv(M, blockdim), ceildiv(N, blockdim));
   gemm_kernel<<<grid, block>>>(alpha,
      A.view(), B.view(), beta, C.view());
}
```

gemm1.hpp

```
gemm_kernel<<<grid, block>>>(alpha,
   A.view(), B.view(), beta, C.view());
```

- An die Kernel-Funktion werden hier Views übergeben, d.h. Matrix-Objekte ohne eigene Datenhaltung.
- Die Views werden *by value* übergeben und enthalten nur die Dimensionierung der Matrix, einen Zeiger auf die Daten und Informationen zur Organisation der Matrix im Speicher.

- Prinzipiell können Matrix-Klassen eingerichtet werden, die Matrizen je nach Ausprägung entweder auf der GPU oder auf der CPU halten.
- Kopieroperationen ermöglichen das Verschieben der Daten. Effizient gelingt dies nur, wenn die jeweiligen Matrizen gleichartig im Speicher organisiert sind.
- Objekte des Typs *DeviceGeMatrix* leben nur auf der CPU-Seite, deren Daten liegen aber auf der GPU-Seite.
- Die zugehörigen Views des Typs *DeviceGeMatrixView* können sowohl auf der CPU- als auch der GPU-Seite verwendet werden, wobei wiederum die Daten nur auf der GPU-Seite zugänglich sind.

test-gemm.cu

```
using T = double;
constexpr std::size_t M = 512;
constexpr std::size_t N = 512;
constexpr std::size_t K = 512;
T alpha = 1.0; T beta = 1.5;

using namespace hpc::cuda;
using namespace hpc::matvec;

GeMatrix<T> A_host(M, K, Order::RowMajor);
GeMatrix<T> B_host(K, N, Order::RowMajor);
GeMatrix<T> C1_host(M, N, Order::RowMajor);
GeMatrix<T> C2_host(M, N, Order::RowMajor);

DeviceGeMatrix<T> A_dev(M, K, Order::RowMajor);
DeviceGeMatrix<T> B_dev(K, N, Order::RowMajor);
DeviceGeMatrix<T> C2_dev(M, N, Order::RowMajor);

init_matrix(A_host); copy(A_host, A_dev);
init_matrix(B_host); copy(B_host, B_dev);
init_matrix(C1_host);copy(C1_host, C2_dev);

auto start = std::chrono::high_resolution_clock::now();
cuda_gemm(alpha, A_dev, B_dev, beta, C2_dev);
CHECK_CUDA(cudaDeviceSynchronize); // wait for the kernel to finish
auto finish = std::chrono::high_resolution_clock::now();

copy(C2_dev, C2_host);
```

gemm2.hpp

```
std::size_t i = threadIdx.y + blockIdx.y * BLOCK_DIM;
std::size_t j = threadIdx.x + blockIdx.x * BLOCK_DIM;

/* ... */

if (i < C.numRows() && j < C.numCols()) {
   C(i, j) = sum;
}
```

- Im folgenden gehen wir davon aus, dass sowohl A, B als auch C jeweils in *row major* organisiert sind.
- Dann ist es sinnvoll, den Spaltenindex, der auf benachbarte Speicherzellen zugreift, mit *threadIdx.x* zu verknüpfen.
- Damit wird sichergestellt, dass ein Warp bzw. Half-Warp auf benachbarte Daten zugreift.

gemm2.hpp

```
__shared__ T ablock[BLOCK_DIM][BLOCK_DIM];
__shared__ T bblock[BLOCK_DIM][BLOCK_DIM];
```

- Wenn kein konsekutiver Zugriff erfolgt, kann es sich lohnen, dies über Datenstruktur abzuwickeln, die allen Threads eines Blocks gemeinsam ist.
- Die Idee ist, dass dieses Array gemeinsam von allen Threads eines Blocks konsekutiv gefüllt wird.
- Der Zugriff auf das gemeinsame Array ist recht effizient und muss nicht mehr konsekutiv sein.
- Die Matrix-Matrix-Multiplikation muss dann aber blockweise organisiert werden.

gemm2.hpp

```
std::size_t K = A.numCols();
std::size_t rounds = (K + BLOCK_DIM - 1) / BLOCK_DIM;
T sum{};
for (std::size_t round = 0; round < rounds; ++round) {
   T val;
   if (i < A.numRows() && round*BLOCK_DIM + threadIdx.x < A.numCols()) {
      val = A(i, round*BLOCK_DIM + threadIdx.x);
   } else {
      val = 0;
   }
   ablock[threadIdx.y][threadIdx.x] = val;
   if (round*BLOCK_DIM + threadIdx.x < B.numRows() && j < B.numCols()) {
      val = B(round*BLOCK_DIM + threadIdx.y, j);
   } else {
      val = 0;
   }
   bblock[threadIdx.y][threadIdx.x] = val;
   __syncthreads();
   #pragma unroll
   for (std::size_t k = 0; k < BLOCK_DIM; ++k) {
      sum += ablock[threadIdx.y][k] * bblock[k][threadIdx.x];
   }
   __syncthreads();
}
```

- Bislang wurden überwiegend alle CUDA-Aktivitäten sequentiell durchgeführt, abgesehen davon, dass die Kernel-Funktionen parallelisiert abgearbeitet werden und der Aufruf eines Kernels asynchron erfolgt.
- In vielen Fällen bleibt so Parallelisierungspotential ungenutzt.
- CUDA-Streams sind eine Abstraktion, mit deren Hilfe mehrere sequentielle Abläufe definiert werden können, die voneinander unabhängig sind und daher prinzipiell parallelisiert werden können.
- Ferner gibt es Synchronisierungsoperationen und das Behandeln von Ereignissen mit CUDA-Streams.

Folgende Aktivitäten können mit Hilfe von CUDA-Streams unabhängig voneinander parallel laufen:

- CPU und GPU können unabhängig voneinander operieren
- der Transfer von Daten und die Ausführung von Kernel-Funktionen.
- Mehrere Kernel können auf der gleichen GPU konkurrierend ausgeführt werden (bei Livingstone können 32 Kernel parallel laufen).
- Wenn mehrere GPUs zur Verfügung stehen, können diese ebenfalls parallel laufen.

Insbesondere bietet es sich an, den Datentransfer und die Ausführung der Kernel-Funktionen zu parallelisieren. Dabei können insbesondere Datentransfers vom Hauptspeicher zur GPU und in umgekehrter Richtung von der GPU zum Hauptspeicher ungestört parallel laufen.

Sobald ein CUDA-Stream erzeugt worden ist, können einzelne Operationen oder der Aufruf einer Kernel-Funktion einem Stream zugeordnet werden:

cudaError_t cudaStreamCreate (cudaStream_t stream)*
: Erzeugt einen neuen Stream. Bei *cudaStream_t* handelt es sich um einen Zeiger auf eine nicht-öffentliche Datenstruktur, die beliebig kopiert werden kann.

cudaError_t cudaStreamSynchronize (cudaStream_t stream)
: Wartet bis alle Aktivitäten des Streams beendet sind.

cudaError_t cudaStreamDestroy (cudaStream_t stream)
: Wartet auf die Beendigung der mit dem Stream verbundenen Aktivitäten und anschließende Freigabe der zum Stream gehörenden Ressourcen.

Für die Datentransfers stehen asynchrone Operationen zur Verfügung, die einen Stream als Parameter erwarten:

cudaError_t cudaMemcpyAsync(**void**∗ *dst*, **const void**∗ *src*, *size_t count*, **enum** *cudaMemcpyKind kind*, *cudaStream_t stream*)

Funktioniert analog zu *cudaMemcpy*, synchronisiert jedoch nicht und reiht den Datentransfer in die zu dem Stream gehörende Sequenz ein.

Beim Aufruf eines Kernels kann bei dem letzten Parameter der Konfiguration ein Stream angegeben werden:

- $<<<$ *Dg*, *Db*, *Ns*, *S* $>>>$
- Der letzte Parameter *S* ist der Stream.
- Bei *Ns* kann im Normalfall einfach 0 angegeben werden.

- Grundsätzlich kann auch ein 0-Zeiger (bzw. **nullptr**) als Stream übergeben werden.
- In diesem Fall werden ähnlich wie bei *cudaDeviceSynchronize* erst alle noch nicht abgeschlossenen CUDA-Operationen abgewartet, bevor die Operation beginnt.
- Das erfolgt aber asynchron, so dass die CPU dessen ungeachtet weiter fortfahren kann.
- Datentransfers und der Aufruf von Kernel-Funktionen ohne die Angabe eines Streams implizieren immer die Verwendung des NULL-Streams. Entsprechend wird in diesen Fällen implizit synchronisiert.
- Wenn versucht wird, mit Streams zu parallelisieren, ist darauf zu achten, dass nicht versehentlich durch die implizite Verwendung eines NULL-Streams eine Synchronisierung erzwungen wird.

Wenn mehrere voneinander unabhängige Kernel-Funktionen hintereinander aufzurufen sind, die jeweils Daten von der CPU benötigen und Daten zurückliefern, lohnt sich u.U. ein Pipelining mit Hilfe von Streams:

- Für jeden Aufruf einer Kernel-Funktion wird ein Stream angelegt.
- Jedem Stream werden drei Operationen zugeordnet:
  - Datentransfer zur GPU
  - Aufruf der Kernel-Funktion
  - Datentransfer zum Hauptspeicher

Wenn der Zeitaufwand für die Datentransfers geringer ist als für die eigentliche Berechnung auf der GPU fällt dieser dank der Parallelisierung weg bei der Berücksichtigung der Gesamtzeit, abgesehen von dem ersten und letzten Datentransfer.

hpc/cuda/copy.hpp

```
template<
   template<typename> class MatrixA,
   template<typename> class MatrixB,
   typename T,
   Require<HostGe<MatrixA<T>>, DeviceGe<MatrixB<T>>> = true
>
void copy(const MatrixA<T>& A, MatrixB<T>& B, Stream& stream) {
   assert(A.numRows() == B.numRows() && A.numCols() == B.numCols() &&
      A.incRow() == B.incRow() && A.incCol() == B.incCol() &&
      consecutively_stored(A) && consecutively_stored(B));
   CHECK_CUDA(cudaMemcpyAsync, B.data(), A.data(),
      A.numRows() * A.numCols() * sizeof(T), cudaMemcpyHostToDevice,
      stream);
}
```

hpc/cuda/copy.hpp

```
template<
   template<typename> class MatrixA,
   template<typename> class MatrixB,
   typename T,
   Require<DeviceGe<MatrixA<T>>, HostGe<MatrixB<T>>> = true
>
void copy(const MatrixA<T>& A, MatrixB<T>& B, Stream& stream) {
   assert(A.numRows() == B.numRows() && A.numCols() == B.numCols() &&
      A.incRow() == B.incRow() && A.incCol() == B.incCol() &&
      consecutively_stored(A) && consecutively_stored(B));
   CHECK_CUDA(cudaMemcpyAsync, B.data(), A.data(),
      A.numRows() * A.numCols() * sizeof(T), cudaMemcpyDeviceToHost,
      stream);
}
```

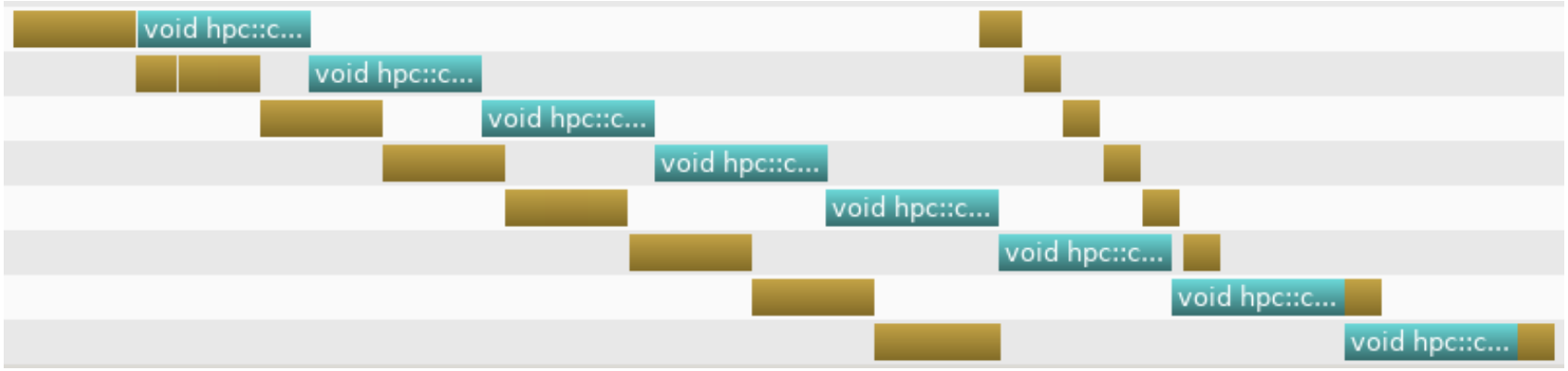
gemm-streamed.cu

```
Stream stream[COUNT];
for (std::size_t index = 0; index < COUNT; ++index) {
   copy(*A_host[index], *A_dev[index], stream[index]);
   copy(*B_host[index], *B_dev[index], stream[index]);
   copy(*C_host[index], *C_dev[index], stream[index]);
   cuda_gemm(alpha, *A_dev[index], *B_dev[index], beta, *C_dev[index],
      stream[index]);
}
for (std::size_t index = 0; index < COUNT; ++index) {
   copy(*C_dev[index], *C_host[index], stream[index]);
}
CHECK_CUDA(cudaDeviceSynchronize); // wait for the kernel to finish
```

- *COUNT* Matrix-Matrix-Multiplikationen werden hier parallel abgewickelt.
- Das entspricht hier dem Fork-And-Join-Pattern, wobei *cudaDeviceSynchronize* dem *join* entspricht.

Matrix-Multiplikationen mit Pipelining


```
livingstone$ nvprof --print-gpu-summary gemm-streamed 2>&1 | cut -b1-104
==13911== NVPROF is profiling process 13911, command: gemm-streamed
8 gemm operations on gpu took 0.103952 s for (1024 x 1024) (1024 x 1024)
time per gemm operation: 0.012994 s
==13911== Profiling application: gemm-streamed
==13911== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   51.45%  93.201ms         8  11.650ms  11.095ms  11.785ms  void hpc::cuda::gemm_kernel
                   37.05%  67.123ms        24  2.7968ms  2.7804ms  3.0706ms  [CUDA memcpy HtoD]
                   11.50%  20.824ms         8  2.6030ms  2.5525ms  2.9126ms  [CUDA memcpy DtoH]
livingstone$ nvprof --print-gpu-summary gemm-serialized 2>&1 | cut -b1-104
==13926== NVPROF is profiling process 13926, command: gemm-serialized
8 gemm operations on gpu took 0.179611 s for (1024 x 1024) (1024 x 1024)
time per gemm operation: 0.0224514 s
==13926== Profiling application: gemm-serialized
==13926== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   50.24%  88.114ms         8  11.014ms  11.003ms  11.022ms  void hpc::cuda::gemm_kernel
                   38.08%  66.788ms        24  2.7828ms  2.7798ms  2.8056ms  [CUDA memcpy HtoD]
                   11.68%  20.476ms         8  2.5595ms  2.5569ms  2.5715ms  [CUDA memcpy DtoH]
livingstone$ nvprof --print-gpu-summary gemm-without-transfers 2>&1 | cut -b1-104
==13940== NVPROF is profiling process 13940, command: gemm-without-transfers
8 gemm operations on gpu took 0.0881717 s for (1024 x 1024) (1024 x 1024)
time per gemm operation: 0.0110215 s
==13940== Profiling application: gemm-without-transfers
==13940== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   50.25%  88.114ms         8  11.014ms  11.005ms  11.022ms  void hpc::cuda::gemm_kernel
                   38.09%  66.784ms        24  2.7827ms  2.7798ms  2.8080ms  [CUDA memcpy HtoD]
                   11.66%  20.455ms         8  2.5568ms  2.5557ms  2.5578ms  [CUDA memcpy DtoH]
livingstone$
```



Das Werkzeug *nvvp* erlaubt es, das Scheduling der einzelnen Streams zu visualisieren:

- Jede Zeile entspricht einem Stream.
- Die x-Achse entspricht der Zeitachse.
- Jeder der Streams führt zunächst die erste Kopieraktion aus (*host to device*, in Ocker dargestellt), dann den Kernel (in türkiser Farbe) und schließlich die zweite Kopieraktion (*device to host*, wieder in Ocker).
- In der überwiegenden Zeit läuft immer eine der Kopieraktionen parallel zu einem der Kernel-Aufrufe.

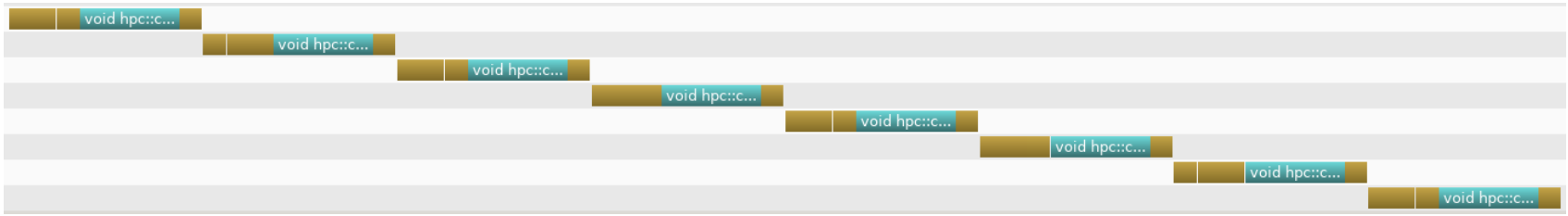
Alternative Variante

gemm-badly-streamed.cu

```
Stream stream[COUNT];
for (std::size_t index = 0; index < COUNT; ++index) {
   copy(*A_host[index], *A_dev[index], stream[index]);
   copy(*B_host[index], *B_dev[index], stream[index]);
   copy(*C_host[index], *C_dev[index], stream[index]);
   cuda_gemm(alpha, *A_dev[index], *B_dev[index], beta, *C_dev[index],
      stream[index]);
   copy(*C_dev[index], *C_host[index], stream[index]);
}
```

- Diese Variante liefert das gleiche Resultat und kommt nur mit einer *for*-Schleife aus.
- Ist sie aber auch genauso gut?

```
livingstone$ nvprof --print-gpu-summary gemm-badly-streamed 2>&1 | cut -b1-104
==16244== NVPROF is profiling process 16244, command: gemm-badly-streamed
8 gemm operations on gpu took 0.1816 s for (1024 x 1024) (1024 x 1024)
time per gemm operation: 0.0227 s
==16244== Profiling application: gemm-badly-streamed
==16244== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   51.27%  91.855ms         8  11.482ms  11.055ms  11.752ms  void hpc::cuda::gemm_kernel
                   37.31%  66.853ms        24  2.7855ms  2.7791ms  2.8216ms  [CUDA memcpy HtoD]
                   11.42%  20.463ms         8  2.5578ms  2.5502ms  2.5712ms  [CUDA memcpy DtoH]
livingstone$ nvprof --print-gpu-summary gemm-streamed 2>&1 | cut -b1-104
==16259== NVPROF is profiling process 16259, command: gemm-streamed
8 gemm operations on gpu took 0.0994354 s for (1024 x 1024) (1024 x 1024)
time per gemm operation: 0.0124294 s
==16259== Profiling application: gemm-streamed
==16259== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   50.24%  88.701ms         8  11.088ms  11.056ms  11.110ms  void hpc::cuda::gemm_kernel
                   37.98%  67.066ms        24  2.7944ms  2.7788ms  3.0685ms  [CUDA memcpy HtoD]
                   11.78%  20.802ms         8  2.6002ms  2.5541ms  2.9147ms  [CUDA memcpy DtoH]
livingstone$
```



- Zu einer Parallelisierung kommt es hier überhaupt nicht.
- Das liegt daran, dass wir hier nur über eine einzige *copy engine* verfügen. Die Queue wird somit strikt in der Reihenfolge abgearbeitet, wie die Jobs eingegangen sind. Entsprechend wartet der erste *device to host* Kopierjob erst auf die Fertigstellung des ersten Kernel-Aufrufs. Erst danach kommen die *host to device* Kopierjobs für die nächste Matrix.
- Es werden in der Queue leider keine Jobs nach vorne gezogen, wenn deren Abhängigkeiten bereits alle erfüllt sind.

- Der GPU steht über die PCIe-Schnittstelle *direct memory access* (DMA) zur Verfügung.
- Dies wäre recht schnell, kommt aber normalerweise nicht zum Zuge, da dazu sichergestellt sein muss, dass die entsprechenden Kacheln im Hauptspeicher nicht zwischenzeitlich vom Betriebssystem ausgelagert werden.
- Alternativ ist es möglich, ausgewählte Bereiche des Hauptspeichers zu reservieren, so dass diese nicht ausgelagert werden können (*pinned memory*).
- Davon sollte zurückhaltend Gebrauch gemacht werden, da dies ein System in die Knie zwingen kann, wenn zuviele physische Kacheln reserviert sind.

Nicht auslagerbarer Speicher (*pinned memory*) muss mit speziellen Funktionen belegt und freigegeben werden:

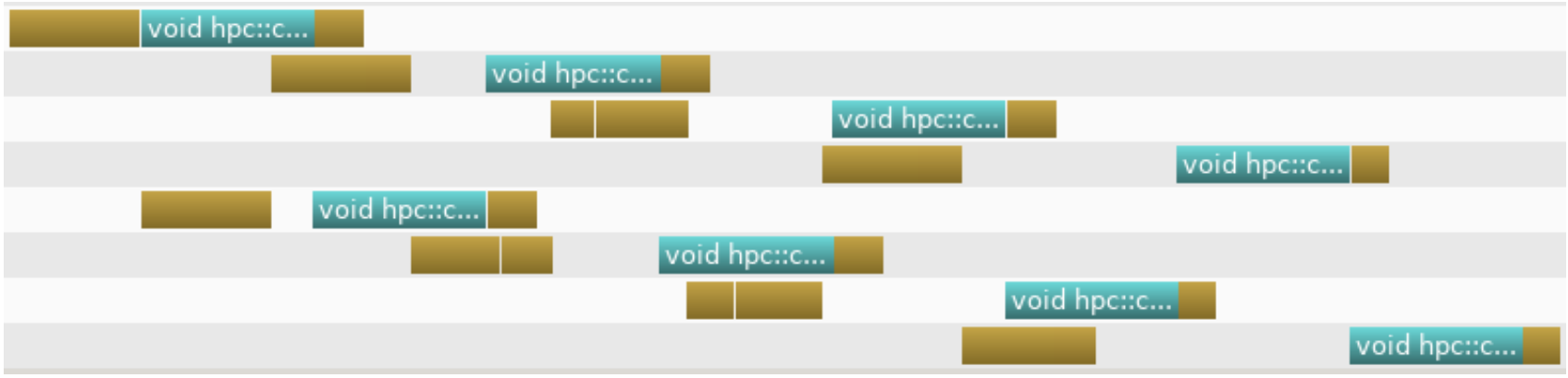
cudaError_t cudaMallocHost(**void**∗∗ *ptr*, *size_t size*)
: belegt ähnlich wie *malloc* Hauptspeicher, wobei hier sichergestellt wird, dass dieser nicht ausgelagert wird.

cudaError_t cudaFreeHost(**void**∗)
: gibt den mit *cudaMallocHost* oder *cudaHostAlloc* reservierten Speicher wieder frei.

hpc/cuda/pinned-buffer.hpp

```
template<typename T>
struct PinnedBuffer {
   void* const hostptr;
   T* const aligned_hostptr;
   PinnedBuffer(std::size_t length, std::size_t alignment = alignof(T)) :
         hostptr(cuda_host_malloc(compute_aligned_size<T>(length, alignment)))
         aligned_hostptr(align_ptr<T>(hostptr, alignment)) {
   }
   ~PinnedBuffer() {
      CHECK_CUDA(cudaFreeHost, hostptr);
   }
   T* data() const {
      return aligned_hostptr;
   }
   /* ... */
};
```

- Als entsprechende RAII-Klasse steht *PinnedBuffer* zur Verfügung.
- Darauf aufbauend gibt es *PinnedDenseVector* und *PinnedGeMatrix* mitsamt den zugehörigen Views, die sich ansonsten genauso wie *DenseVector* und *GeMatrix* verhalten.



- Durch die Verwendung von *PinnedGeMatrix* an Stelle von *GeMatrix* sieht sich die GPU in der Lage, Kopieraktionen zu parallelisieren, wenn sich die Transferrichtungen voneinander unterscheiden.
- Bei diesem Beispiel gibt es jedoch keinen Zeitgewinn, da immer nur eine Kernel-Funktion ausgeführt wird und am Ende noch das Resultat des letzten Kernel-Funktionsaufrufs zu übertragen ist.

Neuere Grafikkarten und Versionen der CUDA-Schnittstelle (einschließlich Livingstone) erlauben die Abbildung nicht auslagerbaren Hauptspeichers in den Adressraum der GPU:

cudaError_t cudaHostAlloc(**void**∗∗ *ptr*, *size_t size*, **unsigned int** *flags*)
belegt Hauptspeicher, der u.a. in den virtuellen Adressraum der GPU abgebildet werden kann. Folgende miteinander kombinierbare Optionen gibt es:

cudaHostAllocDefault emuliert *cudaMallocHost*, d.h. der Speicher wird nicht abgebildet.

cudaHostAllocPortable macht den Speicherbereich allen Grafikkarten zugänglich (falls mehrere zur Verfügung stehen).

cudaHostAllocMapped bildet den Hauptspeicher in den virtuellen Adressraum der GPU ab

cudaHostAllocWriteCombined ermöglicht u.U. eine effizientere Lesezugriffe der GPU zu Lasten der Lesegeschwindigkeit auf der CPU.

Neuere CUDA-Versionen unterstützen *unified virtual memory* (UVM), bei dem die Zeiger auf der CPU- und GPU-Seite für abgebildeten Hauptspeicher identisch sind. (Dies gilt auch dann, wenn wenn die CPU mit 64-Bit- und die GPU mit 32-Bit-Zeigern arbeitet.)
Ohne UVM müssen die Zeiger abgebildet werden:

cudaError_t cudaHostGetDevicePointer(**void**∗∗ *pDevice*, **void**∗ *pHost*, **unsigned int** *flags*)

liefert für Hauptspeicher, der in den Adressraum der GPU abgebildet ist (*cudaDeviceMapHost* wurde angegeben) den entsprechenden Zeiger in den Adressraum der GPU. Bei *unified virtual memory* (UVM) sind beide Zeiger identisch. (Bei den *flags* ist nach dem aktuellen Stand der API immer 0 anzugeben.)

Ob UVM unterstützt wird oder nicht, lässt sich über das Feld *unifiedAddressing* aus der **struct** *cudaDeviceProp* ermitteln, die mit *cudaGetDeviceProperties* gefüllt werden kann.

- Bei integrierten Systemen wird jeglicher Kopieraufwand vermieden (siehe das Feld *integrated* in der **struct** *cudaDeviceProp*).
- Bei nicht-integrierten Systemen werden die Daten jeweils implizit per *direct memory access* transferiert.
- Wenn der Speicher auf der GPU sonst nicht ausreicht.
- Wenn jede Speicherzelle nicht mehr als einmal in konsekutiver Weise gelesen oder geschrieben wird (ansonsten sind Zugriffe durch einen Cache effizienter).
- Reine Schreibzugriffe sind günstiger, da hier die Synchronisierung wegfällt, d.h. die Umsetzung einer Schreib-Operation und die Fortsetzung der Kernel-Funktion erfolgen parallel.
- Bei Lesezugriffen ist der Vorteil geringer, da hier gewartet werden muss.

Datenstruktur für abgebildeten Hauptspeicher

hpc/cuda/mapped-buffer.hpp

```
template<typename T>
struct MappedBuffer {
   void* const hostptr;
   T* const aligned_hostptr;
   void* const devptr;
   T* const aligned_devptr;

   MappedBuffer(std::size_t length, std::size_t alignment = alignof(T)) :
         hostptr(cuda_host_malloc(compute_aligned_size<T>(length, alignment))),
         aligned_hostptr(align_ptr<T>(hostptr, alignment)),
         devptr(convert_host_to_device_ptr(hostptr)),
         aligned_devptr(align_devptr(hostptr, aligned_hostptr, devptr)) {
   }

   ~MappedBuffer() {
      CHECK_CUDA(cudaFreeHost, hostptr);
   }

   HOST_DEV
   T* host_data() const {
      return aligned_hostptr;
   }

   HOST_DEV
   T* dev_data() const {
      return aligned_devptr;
   }

   /* ... */
};
```

hpc/cuda/mapped-buffer.hpp

```
inline void* cuda_host_malloc(std::size_t size) {
   void* hostptr = nullptr;
   CHECK_CUDA(cudaHostAlloc, &hostptr, size, cudaHostAllocMapped);
   return hostptr;
}

inline void* convert_host_to_device_ptr(void* hostptr) {
   void* devptr = nullptr;
   CHECK_CUDA(cudaHostGetDevicePointer, &devptr, hostptr, 0);
   return devptr;
}

template<typename T>
inline T* align_devptr(void* hostptr, T* aligned_hostptr, void* const devptr) {
   auto offset = aligned_hostptr - static_cast<T*>(hostptr);
   return static_cast<T*>(devptr) + offset;
}
```

hpc/cuda/buffer.hpp

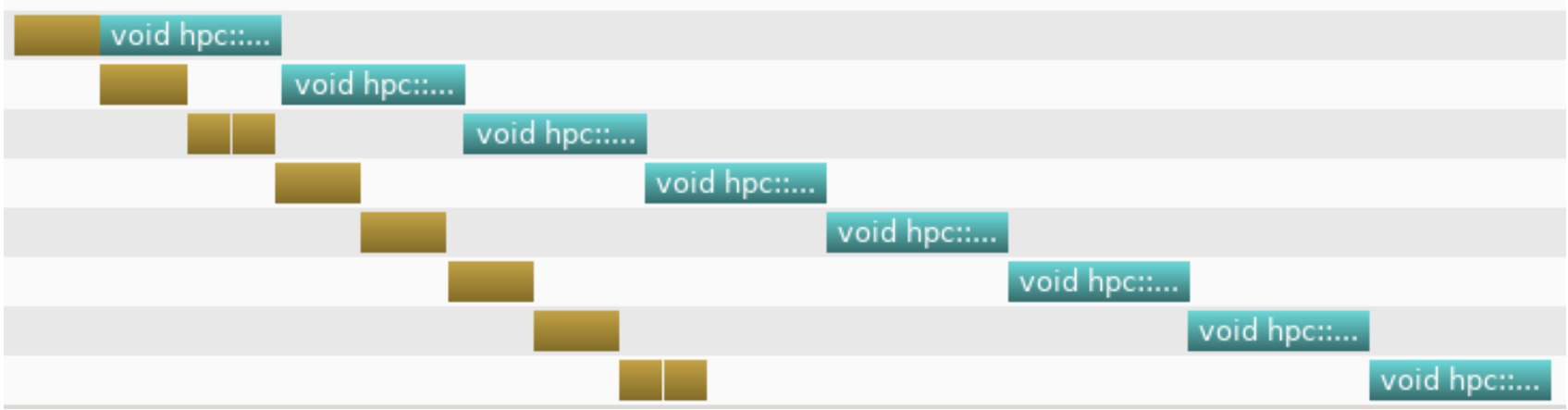
```
template<typename T>
constexpr std::size_t
compute_aligned_size(std::size_t length, std::size_t wanted_alignment) {
   return length * sizeof(T) + std::max(wanted_alignment, alignof(T));
}
```

Variante mit abgebildetem Speicher

gemm-mapped-and-streamed.cu

```
std::unique_ptr<GeMatrix<T>> A_host[COUNT];
std::unique_ptr<GeMatrix<T>> B_host[COUNT];
std::unique_ptr<DeviceGeMatrix<T>> A_dev[COUNT];
std::unique_ptr<DeviceGeMatrix<T>> B_dev[COUNT];
std::unique_ptr<MappedGeMatrix<T>> C[COUNT];

for (std::size_t index = 0; index < COUNT; ++index) {
   A_host[index] = std::make_unique<GeMatrix<T>>(M, K, Order::RowMajor);
   B_host[index] = std::make_unique<GeMatrix<T>>(K, N, Order::RowMajor);
   C[index] = std::make_unique<MappedGeMatrix<T>>(M, N, Order::RowMajor);
   init_matrix(*A_host[index]);
   init_matrix(*B_host[index]);
   init_matrix(*C[index]);
   A_dev[index] = std::make_unique<DeviceGeMatrix<T>>(M, K, Order::RowMajor);
   B_dev[index] = std::make_unique<DeviceGeMatrix<T>>(K, N, Order::RowMajor);
}

auto start = std::chrono::high_resolution_clock::now();
Stream stream[COUNT];
for (std::size_t index = 0; index < COUNT; ++index) {
   copy(*A_host[index], *A_dev[index], stream[index]);
   copy(*B_host[index], *B_dev[index], stream[index]);
   cuda_gemm(alpha, *A_dev[index], *B_dev[index], beta, *C[index],
      stream[index]);
}
CHECK_CUDA(cudaDeviceSynchronize); // wait for the kernel to finish
auto finish = std::chrono::high_resolution_clock::now();
```

```
livingstone$ nvprof --print-gpu-summary gemm-streamed 2>&1 | cut -b1-104
==24807== NVPROF is profiling process 24807, command: gemm-streamed
8 gemm operations on gpu took 0.105245 s for (1024 x 1024) (1024 x 1024)
time per gemm operation: 0.0131556 s
==24807== Profiling application: gemm-streamed
==24807== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   51.81%  94.515ms         8  11.814ms  11.777ms  11.832ms  void hpc::cuda::gemm_kernel
                   36.77%  67.082ms        24  2.7951ms  2.7799ms  3.0637ms  [CUDA memcpy HtoD]
                   11.42%  20.829ms         8  2.6036ms  2.5577ms  2.9210ms  [CUDA memcpy DtoH]
livingstone$ nvprof --print-gpu-summary gemm-mapped-and-streamed 2>&1 | cut -b1-104
==24821== NVPROF is profiling process 24821, command: gemm-mapped-and-streamed
8 gemm operations on gpu took 0.0940759 s for (1024 x 1024) (1024 x 1024)
time per gemm operation: 0.0117595 s
==24821== Profiling application: gemm-mapped-and-streamed
==24821== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   66.23%  88.828ms         8  11.104ms  11.064ms  11.130ms  void hpc::cuda::gemm_kernel
                   33.77%  45.283ms        16  2.8302ms  2.7809ms  2.9374ms  [CUDA memcpy HtoD]
livingstone$
```



- Die expliziten asynchronen Kopieraktionen *device to host* sind jetzt weggefallen.
- Entsprechend wird es jetzt etwas schneller fertig, da wir nicht mehr auf eine letzte Kopieraktion warten müssen.
- Wichtig ist, dass nur C abgebildet wird. Wenn A und B ebenfalls abgebildet wären, dann würde das zu katastrophalen Zeiten führen.

Die CUDA-Schnittstelle bietet auch die Möglichkeit, auf konventionelle Weise belegten Speicher (etwa mit **new** oder *malloc*) nachträglich gegen Auslagerung zu schützen:

cudaError_t cudaHostRegister(**void**∗ *ptr*, *size_t size*, **unsigned int** *flags*)
schützt die Speicherfläche, auf die *ptr* verweist, vor einer Auslagerung. Zwei Optionen werden unterstützt:

cudaHostRegisterPortable die Speicherfläche wird von allen GPUs als nicht auslagerbar erkannt.

cudaHostRegisterMapped die Speicherfläche wird in den Adressraum der GPU abgebildet. (Achtung: Selbst bei UVM kann nicht auf *cudaHostGetDevicePointer* verzichtet werden.)

cudaError_t cudaHostUnregister(**void**∗ *ptr*)
beendet den Schutz vor Auslagerung.

Die CUDA-Schnittstelle unterstützt Ereignisse, die der Synchronisierung und der Zeitmessung dienen:

cudaError_t cudaEventCreate(cudaEvent_t event)*
: legt ein Ereignis-Objekt an mit der Option *cudaEventDefault*, d.h. Zeitmessungen sind möglich, eine Synchronisierung erfolgt jedoch im *busy-wait*-Verfahren.

cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream)
: fügt in die zu *stream* gehörende Ausführungssequenz die Anweisung hinzu, das Eintreten des Ereignisses zu signalisieren.

cudaError_t cudaEventDestroy(cudaEvent_t event)
: gibt die mit dem Ereignis-Objekt verbundenen Ressourcen wieder frei.

CUDA-Event-Operationen zur Synchronisierung und Zeitmessung:

cudaError_t cudaEventSynchronize(cudaEvent_t event)

der aufrufende Thread wartet, bis das Ereignis eingetreten ist. Wenn jedoch *cudaEventRecord* vorher noch nicht aufgerufen worden ist, kehrt dieser Aufruf sofort zurück. Wenn die Option *cudaEventBlockingSync* nicht gesetzt wurde, wird im *busy-wait*-Verfahren gewartet.

*cudaError_t cudaEventElapsedTime(***float**** ms, cudaEvent_t start, cudaEvent_t end)*

liefert die Zeit in Millisekunden, die zwischen den Ereignissen *start* und *end* vergangen ist. Die Auflösung der Zeit beträgt etwa eine halbe Mikrosekunde. Diese Zeitmessung ist genauer als konventionelle Methoden, weil hierfür die Uhr auf der GPU verwendet wird.

```
cudaEvent_t start_event; CHECK_CUDA(cudaEventCreate, &start_event);
cudaEvent_t end_event; CHECK_CUDA(cudaEventCreate, &end_event);
CHECK_CUDA(cudaEventRecord, start_event);

// kernel invocations, data transfers etc.

CHECK_CUDA(cudaEventRecord, end_event);
CHECK_CUDA(cudaDeviceSynchronize); // wait for everything to finish

float timeInMillisecs;
CHECK_CUDA(cudaEventElapsedTime, &timeInMillisecs,
   start_event, end_event);
std::cerr << "GPU time in ms: " << timeInMillisecs << std::endl;
CHECK_CUDA(cudaEventDestroy, start_event);
CHECK_CUDA(cudaEventDestroy, end_event);
```

- Zu beachten ist hier, dass *cudaEventRecord* asynchron abgewickelt wird und das Ereignis erst signalisiert, wenn alle laufenden CUDA-Operationen abgeschlossen sind. Die CPU muss sich also danach immer noch mit *cudaDeviceSynchronize* synchronisieren.
- Zeitmessungen sollten immer auf Ereignissen beruhen, die mit dem NULL-Stream verbunden sind. Das Messen der Realzeit einzelner CUDA-Streams ist nicht sinnvoll, da sich jederzeit andere Operationen dazwischenschieben können.