High Performance Computing: Teil 1
SS 2013

Andreas F. Borchert
Universität Ulm

14. Mai 2013

Inhalte:

- Teil 1: Architekturen von Parallelrechnern, Techniken und Schnittstellen zur Parallelisierung (Threads, OpenMP, MPI und CUDA)
- Teil 2: Parallele numerische Verfahren für lineare Gleichungssysteme, Gebietszerlegung, Graph-Partitionierung, parallele Vorkonditionierer
- Teil 3: Paralleles Mehrgitter-Verfahren, symmetrische Eigenwertprobleme

- Die Vorlesung wird von Prof. Funken, Prof. Urban und mir bestritten.
- Es stehen uns 14 Vorlesungstage in diesem Semester zur Verfügung, die sich wie folgt aufteilen:
  - Teil 1: 5 Vorlesungen (Borchert)
  - Teil 2: 6 Vorlesungen (Funken)
  - Teil 3: 3 Vorlesungen (Urban)
  - Präsentationen: Anfang August (32. Woche)
- Die Übungen und das Praktikum werden von Markus Bantle und Kristina Steih betreut.

- Für die Prüfung ist die
  - erfolgreiche Teilnahme an den Übungen und Praktika und
  - ein Abschluss-Projekt erforderlich, über das vorgetragen wird.
- Einzelheiten dazu folgen noch
- `http://www.uni-ulm.de/mawi/mawi-numerik/lehre/sommersemester-2013/vorlesung-high-performance-computing.html`

Der 1965–1976 entwickelte Parallelrechner ILLIAC 4 (zunächst University of Illinois, dann NASA) symbolisiert mit seinen 31 Millionen US-Dollar Entwicklungskosten den Willen, keinen Aufwand zu scheuen, wenn es um bessere Architekturen für wissenschaftliches Rechnen geht.

- 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.

- 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 die zur Verfügung stehenden Ressourcen wachsen.

- Mit Pipelining werden Techniken bezeichnet, die eine sequentielle Abarbeitung beschleunigen, indem einzelne Arbeitsschritte wie beim Fließband parallelisiert werden.
- Im einfachsten Fall werden hintereinander im Speicher liegende Instruktionen sequentiell ausgeführt.
- Das kann als Instruktions-Strom organisiert werden, bei der die folgenden Instruktionen bereits aus dem Speicher geladen und dekodiert werden, während die aktuelle Instruktion noch ausgeführt wird.
- Bedingte Sprünge sind das Hauptproblem des Pipelining.
- Flynn hatte 1972 die Idee, neben Instruktionsströmen auch Datenströme in die Betrachtung paralleler Architekturen einzubeziehen.

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.

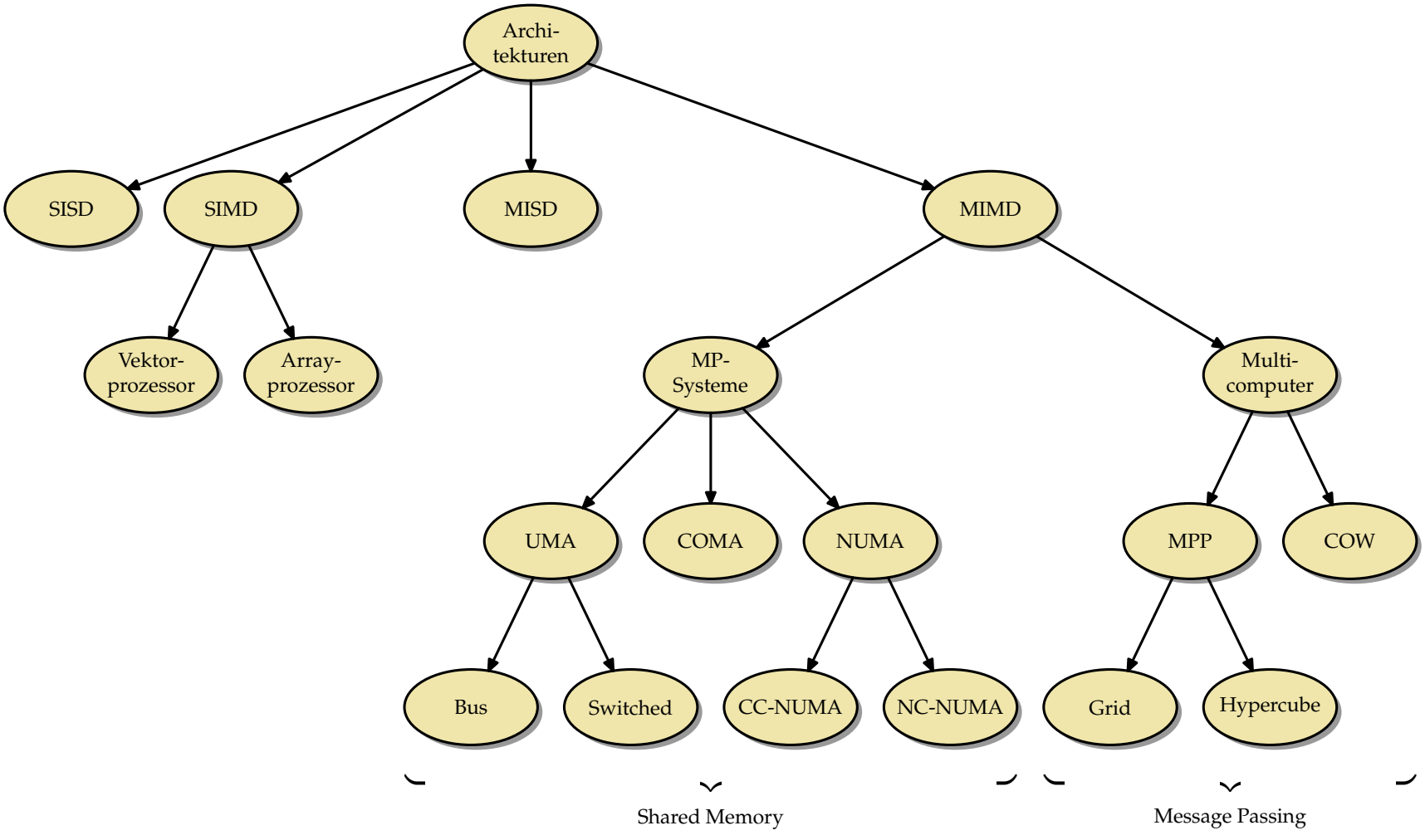
Hier gibt es zwei Varianten:

- Array-Prozessor: Eine Vielzahl von Prozessoren steht zur Verfügung mit zugehörigem Speicher, die diesen in einer Initialisierungsphase laden. Dann werden an alle Prozessoren Anweisungen 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.
- Vektor-Prozessor: Hier steht 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.

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.
- Bei der Pacioli handelt es sich um ein COW (*cluster of workstations*), das aus 36 Knoten besteht. Den einzelnen Knoten stehen jeweils zwei AMD-Opteron-Prozessoren zur Verfügung, eigener Speicher und eigener Plattenplatz. Die Knoten sind untereinander durch ein übliches Netzwerk (GbE) und zusätzlich durch ein Hochgeschwindigkeitsnetzwerk (Infiniband) verbunden.

- Die Hochwanner ist eine Intel-Dualcore-Maschine (2,80 GHz) mit einer Nvidia Quadro 600 Grafikkarte.
- Die Grafikkarte hat 1 GB Speicher, zwei Multiprozessoren und insgesamt 96 Recheneinheiten (SPs = *stream processors*).
- Die Grafikkarte ist 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 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.“

- Threads stehen als Abstraktion eines POSIX-konformen Betriebssystems zur Verfügung unabhängig von der tatsächlichen Ausstattung der Maschine.
- Auf einem Einprozessor-System wird mit Threads nur Nebenläufigkeit erreicht, indem die einzelnen Threads jeweils für kurze Zeitscheiben ausgeführt werden, bevor das Betriebssystem einen Wechsel zu einem anderen Thread einleitet (Kontextwechsel).
- Auf einem MP-System können je nach der Ausstattung auch mehrere Prozessoren zu einem Zeitpunkt einem Prozess zugehordnet werden, so dass dann die Threads echt parallel laufen.

- Dass Threads uneingeschränkt auf den gemeinsamen Speicher zugreifen können, ist zugleich ihr größter Vorteil als auch ihre größte Schwäche.
- Vorteilhaft ist die effiziente Kommunikation, da die Daten hierfür nicht kopiert und übertragen werden müssen, wie es sonst beim Austausch von Nachrichten notwendig wäre.
- Nachteilhaft ist die Fehleranfälligkeit, da die Speicherzugriffe synchronisiert werden müssen und die Korrektheit der Synchronisierung nicht so einfach sichergestellt werden kann, da dies sämtliche Speicherzugriffe betrifft.
- So müssen alle Funktionen oder Methoden, die auf Datenstrukturen zugreifen, damit rechnen, konkurrierend aufgerufen zu werden, und entsprechend darauf vorbereitet sein.

- Spezifikation der *Open Group*:
  http://www.opengroup.org/onlinepubs/007908799/xsh/threads.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 aktuellen C++-Standard ISO 14882-2012 (C++11) werden POSIX-Threads direkt unterstützt.
- Ältere C++-Übersetzer unterstützen dies noch nicht, aber die Boost-Schnittstelle für Threads ist recht ähnlich und kann bei älteren Systemen verwendet werden. (Alternativ kann auch die C-Schnittstelle in C++ verwendet werden, was aber recht umständlich ist.)
- Die folgende Einführung bezieht sich auf C++11. Bei g++ sollte also die Option „-std=gnu++11“ verwendet werden.

- Die ausführende Komponente eines Threads wird in C++ 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 Thread {
   public:
      Thread( /* parameters */ );
      void operator()() {
         // do something in dependence of the parameters
      }
   private:
      // parameters of this thread
};
```

- Eine Klasse für Funktionsobjekte muss den Funktionsoperator unterstützen, d.h. **void** *operator*()().
- Im privaten Bereich der Thread-Klasse können nun alle Parameter untergebracht werden, die für die Ausführung eines Threads benötigt werden.
- Der Konstruktor erhält die Parameter und kopiert diese in den privaten Bereich.
- Nun kann die parameterlose Funktion problemlos auf ihre Parameter zugreifen.

Repräsentierung eines Threads

fork-and-join.cpp

```
class Thread {
   public:
      Thread(int i) : id(i) {};
      void operator()() {
         cout << "thread " << id << " is operating" << endl;
      }

   private:
      const int id;
};
```

- In diesem einfachen Beispiel wird nur ein einziger Parameter für den einzelnen Thread verwendet: *id*
- (Ein Parameter, der die Identität des Threads festlegt, genügt in vielen Fällen bereits.)
- Für Demonstrationszwecke gibt der Funktionsoperator nur seine eigene *id* aus.
- So ein Funktionsobjekt kann auch ohne Threads erzeugt und benutzt werden:
  Thread t(7); *t*();

Threads erzeugen und auf das Ende warten

fork-and-join.cpp

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

using namespace std;

// class Thread...

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

fork-and-join2.cpp

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

- Wenn Threads in Datenstrukturen unterzubringen sind (etwa Arrays oder beliebigen Containern), dann können sie nicht zeitgleich mit einem Funktionsobjekt initialisiert werden.
- In diesem Falle existieren sie zunächst nur als leere Hülle.
- Wenn Thread-Objekte einander zugewiesen werden, dann wird ein Thread nicht dupliziert, sondern die Referenz auf den eigentlichen Thread wandert von einem Thread-Objekt zu einem anderen (Verschiebe-Semantik).
- 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
cout << "Joining..." << 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, int n) {
   assert(n > 0 && a <= b);
   double value = f(a)/2 + f(b)/2;
   double xleft;
   double x = a;
   for (int 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 SimpsonThread {
   public:
      SimpsonThread(double (*_f)(double),
                double _a, double _b, int _n,
                double* resultp) :
            f(_f), a(_a), b(_b), n(_n), rp(resultp) {
      }
      void operator()() {
         *rp = simpson(f, a, b, n);
      }
   private:
      double (*f)(double);
      double a, b;
      int 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-Verfahren über mehrere Threads

simpson.cpp

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

   thread threads[nofthreads];
   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.

Erzeugen der einzelnen Threads

simpson.cpp

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

- Für jedes Teilproblem wird ein entsprechendes Funktionsobjekt erzeugt, womit dann ein Thread erzeugt wird.

simpson.cpp

```
double sum = 0;
for (int 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) {
   istringstream arg(*argv++); --argc;
   if (!(arg >> N) || N <= 0) usage();
}
if (argc > 0) {
   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 *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.

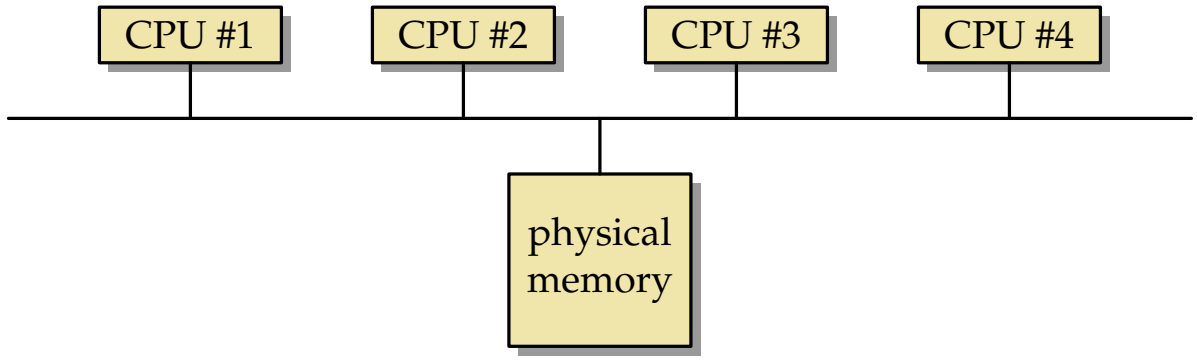
simpson.cpp

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

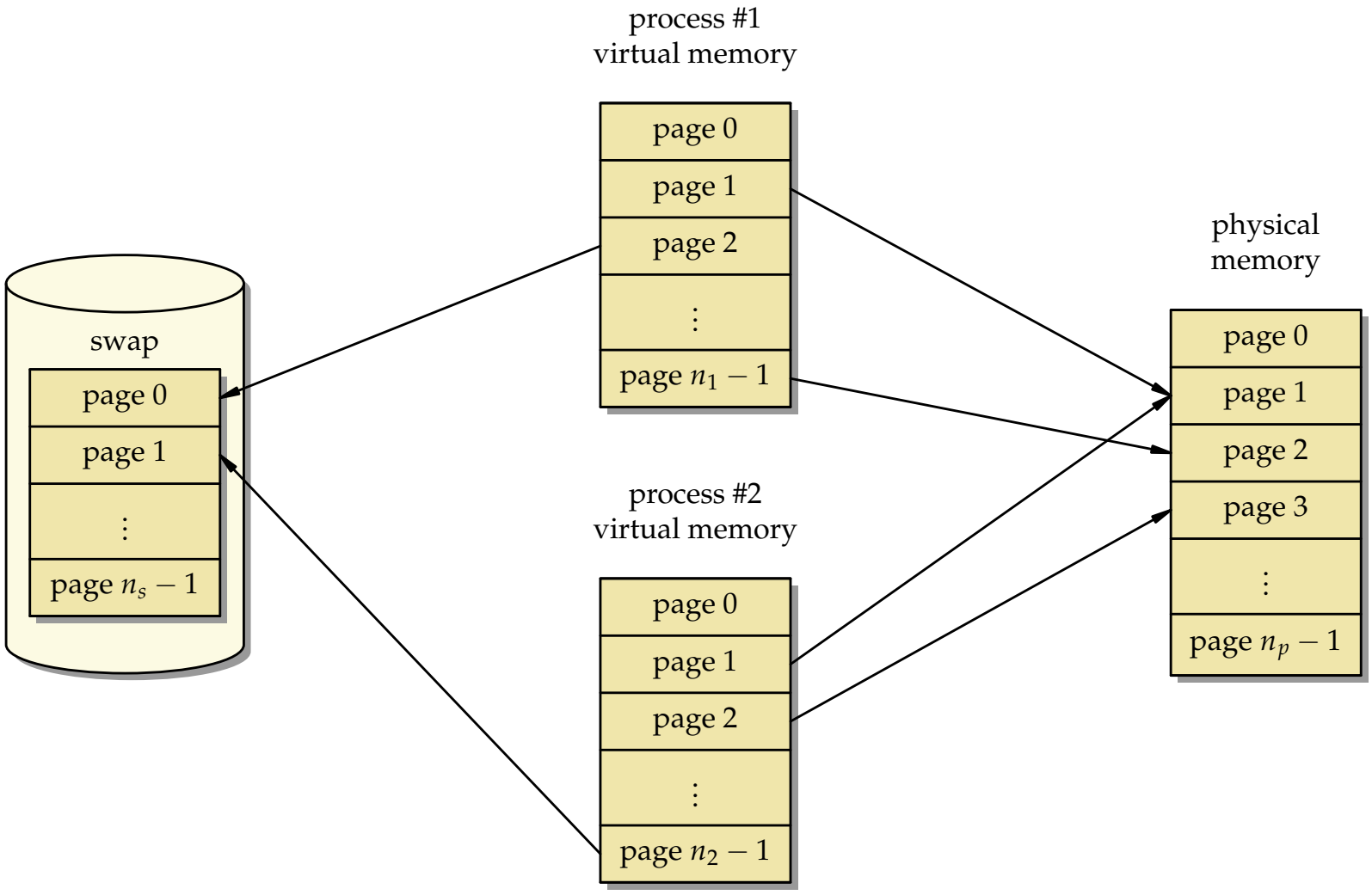
- Testen Sie Ihr Programm zuerst immer ohne Threads, indem die Funktion zur Lösung eines Teilproblems verwendet wird, um das Gesamtproblem unparallelisiert zu lösen. (Diese Variante ist hier auskommentiert.)
- *cout* ist in C++ die Standardausgabe, die mit dem <<-Operator auszugebende Werte erhält.
- *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.

- Grundsätzlich sollte bei solchen Anwendungen die höchstmöglichste Optimierungstufe gewählt werden. Wenn dies unterbleibt, erfolgen wesentlich mehr Speicherzugriffe als notwendig und die Parallelisierungsmöglichkeiten innerhalb einer CPU bleiben teilweise ungenutzt.
- Beim *g++* (GNU-Compiler für C++) ist die Option „-Ofast“ anzugeben.
- Vorsicht: Bei hoher Optimierung und gleichzeitiger Verwendung gemeinsamer Speicherbereiche durch Threads muss ggf. von der Speicherklasse **volatile** konsequent Gebrauch gemacht werden, wenn sonst keine Synchronisierungsfunktionen verwendet werden.

- Erste Ansätze zur Parallelisierung der Ausführung in Rechnern wurden erstaunlich früh vorgestellt.
- Die zuerst 1958 beschriebene Gamma-60-Maschine konnte mehrere Instruktionen entsprechend dem Fork-And-Join-Pattern parallel ausführen.
- Die Architektur eines MP-Systems in Verbindung mit dem Fork-And-Join-Pattern wurde wurde dann von Melvin E. Conway 1963 näher ausgeführt.
- Umgesetzt wurde dies danach u.a. von dem *Berkeley Timesharing System* in einer Form, die den heutigen Threads sehr nahekommt, wobei die damals noch als Prozesse bezeichnet wurden.
- Der erste Ansatz zur Synchronisierung jenseits des Fork-And-Join-Patterns erfolgte mit den Semaphoren durch Edsger W. Dijkstra, der gleichzeitig den Begriff der kooperierenden sequentiellen Prozesse prägte.



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


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 $n_2 - 1$

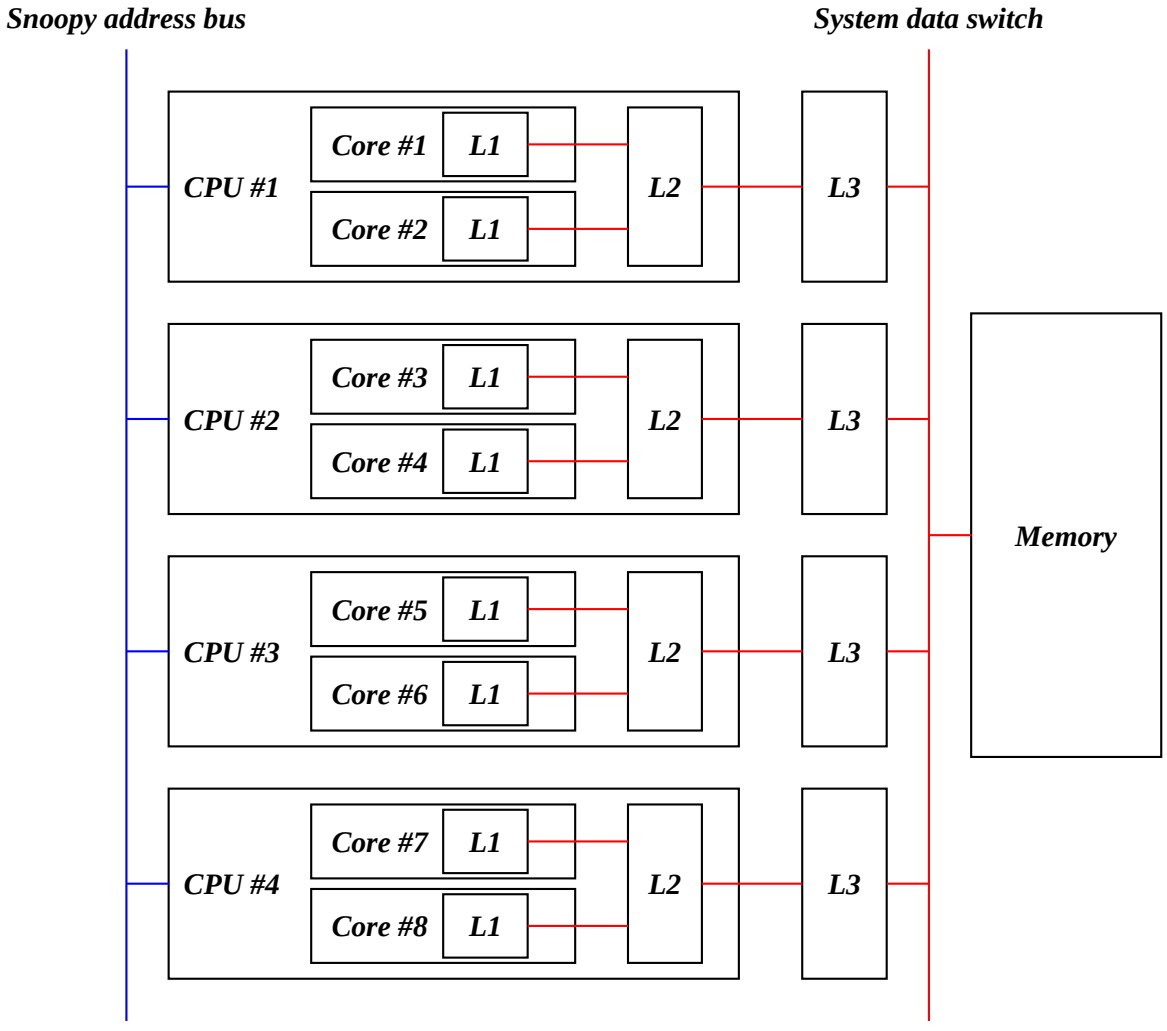
- Der Adressraum eines Prozesses ist eine virtuelle Speicherumgebung, die von dem Betriebssystem mit Unterstützung der jeweiligen Prozessorarchitektur (MMU = *memory management unit*) umgesetzt wird.
- Die virtuellen Adressen eines Prozesses werden dabei in physische Adressen des Hauptspeichers konvertiert.
- 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 Pacioli und Hochwanner 4 KiB (abzurufen über den Systemaufruf *getpagesize*()).
- Wenn nicht genügend physischer Hauptspeicher zur Verfügung steht, können auch einzelne Kacheln auf Platte ausgelagert werden (*swap space*), was zu erheblichen Zeitverzögerungen bei einem nachfolgendem Zugriff führt.

- 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 dem Initialisieren von *std*::*thread*-Objekten) 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.

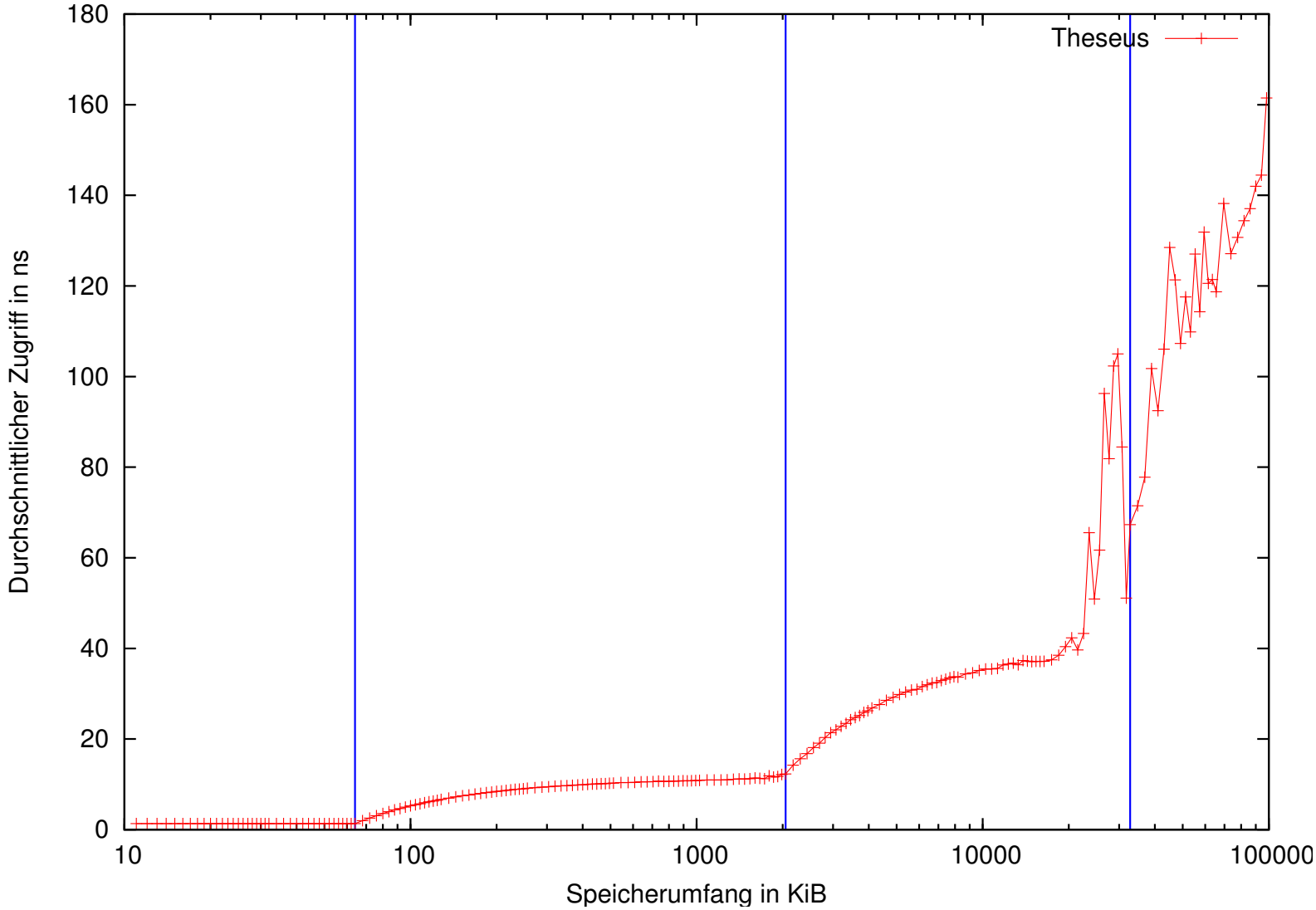
- Zugriffe einer CPU auf den primären Hauptspeicher sind vergleichsweise langsam. Obwohl Hauptspeicher generell schneller wurde, behielten die CPUs ihren Geschwindigkeitsvorsprung.
- Grundsätzlich ist Speicher direkt auf einer CPU deutlich schneller. Jedoch lässt sich Speicher auf einem CPU-Chip aus 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

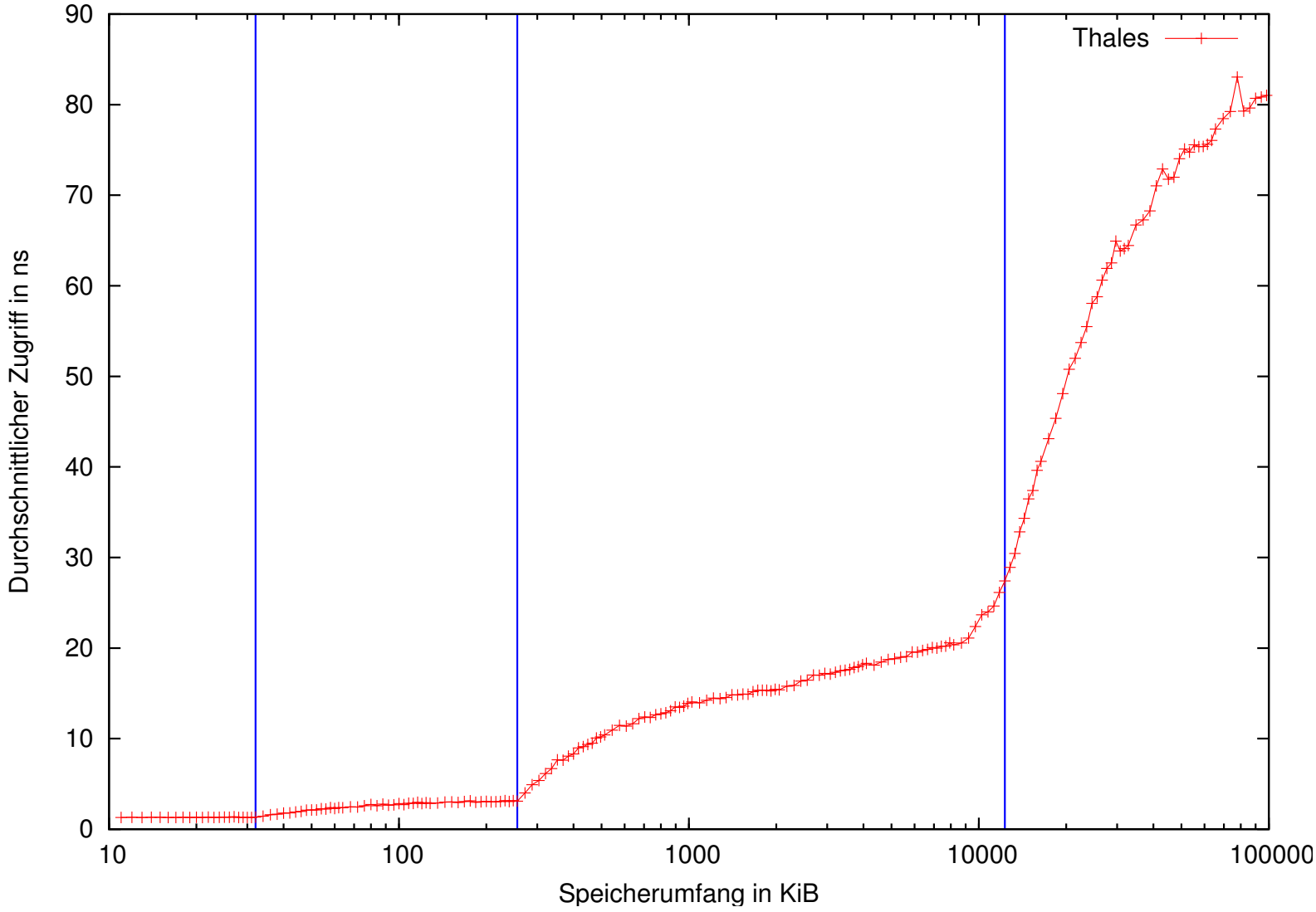


Gemessene Speicherzugriffszeiten: Theseus



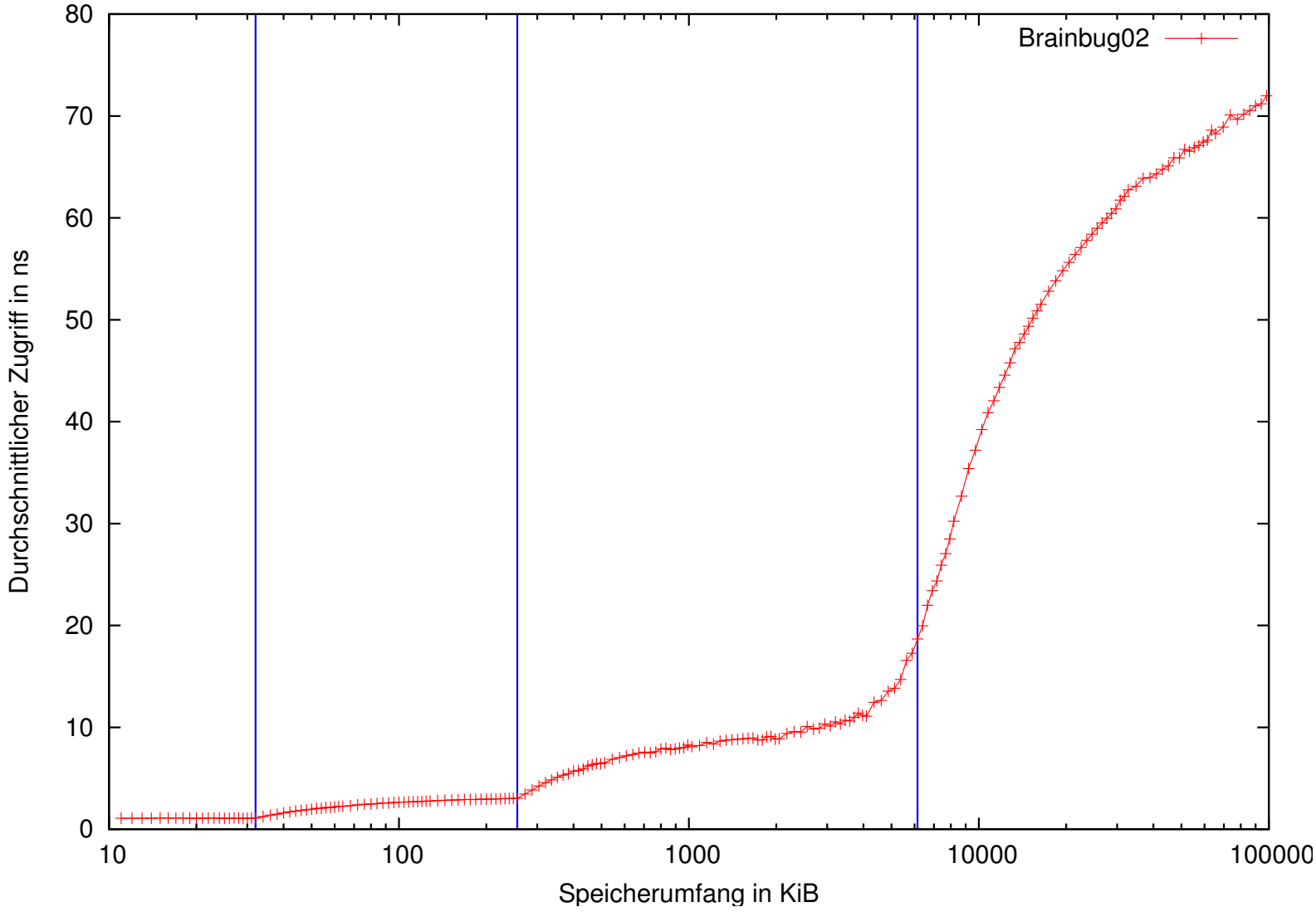
Theseus: 4 UltraSPARC IV+ Prozessoren mit je zwei Kernen
Caches: L1 (64 KiB), L2 (2 MiB), L3 (32 MiB)

Gemessene Speicherzugriffszeiten: Thales



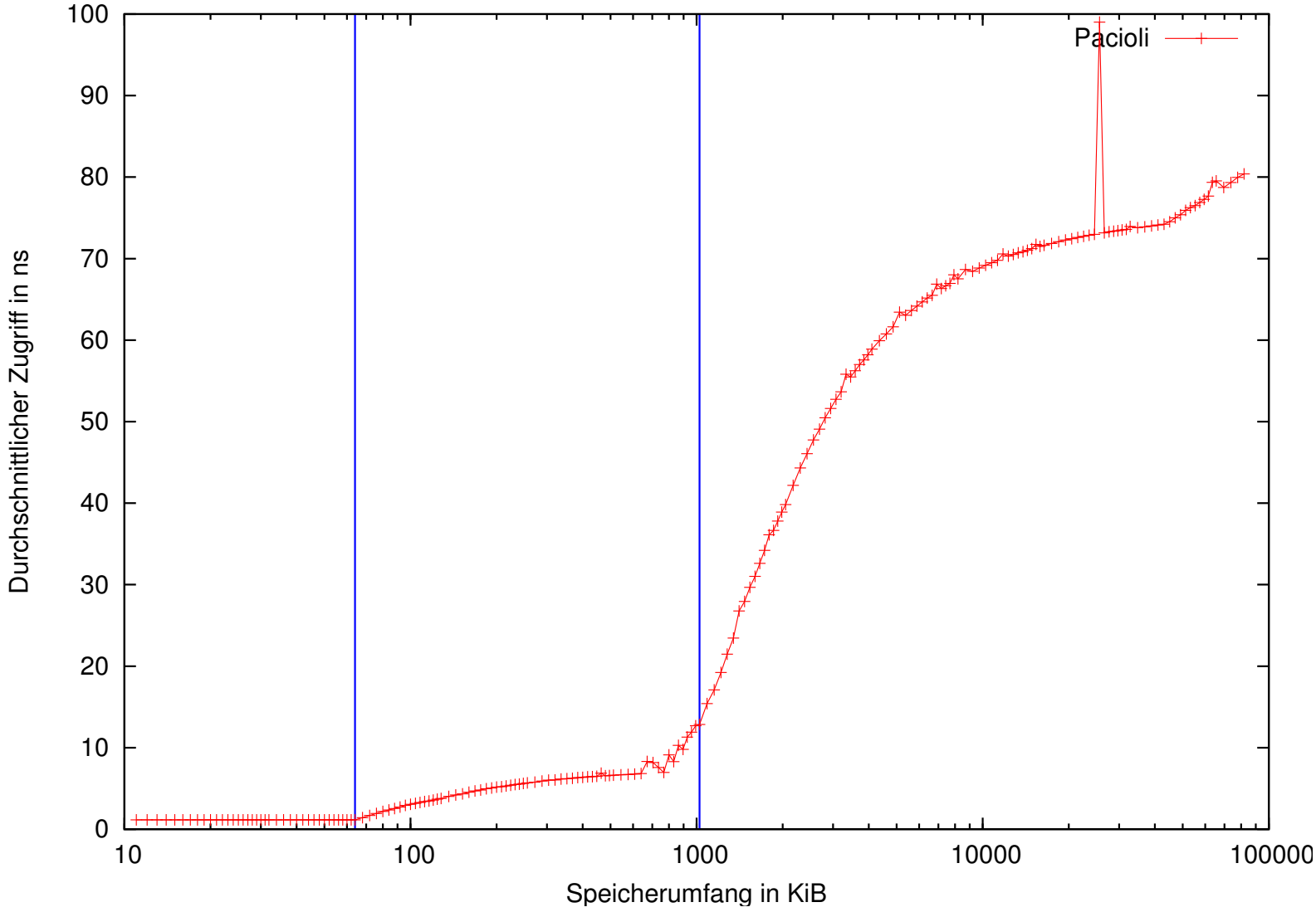
Theseus: 2 Intel X5650-Prozessoren mit je 6 Kernen mit je 2 Threads
Caches: L1 (32 KiB), L2 (256 KiB), L3 (12 MiB)

Gemessene Speicherzugriffszeiten: Brainbug02



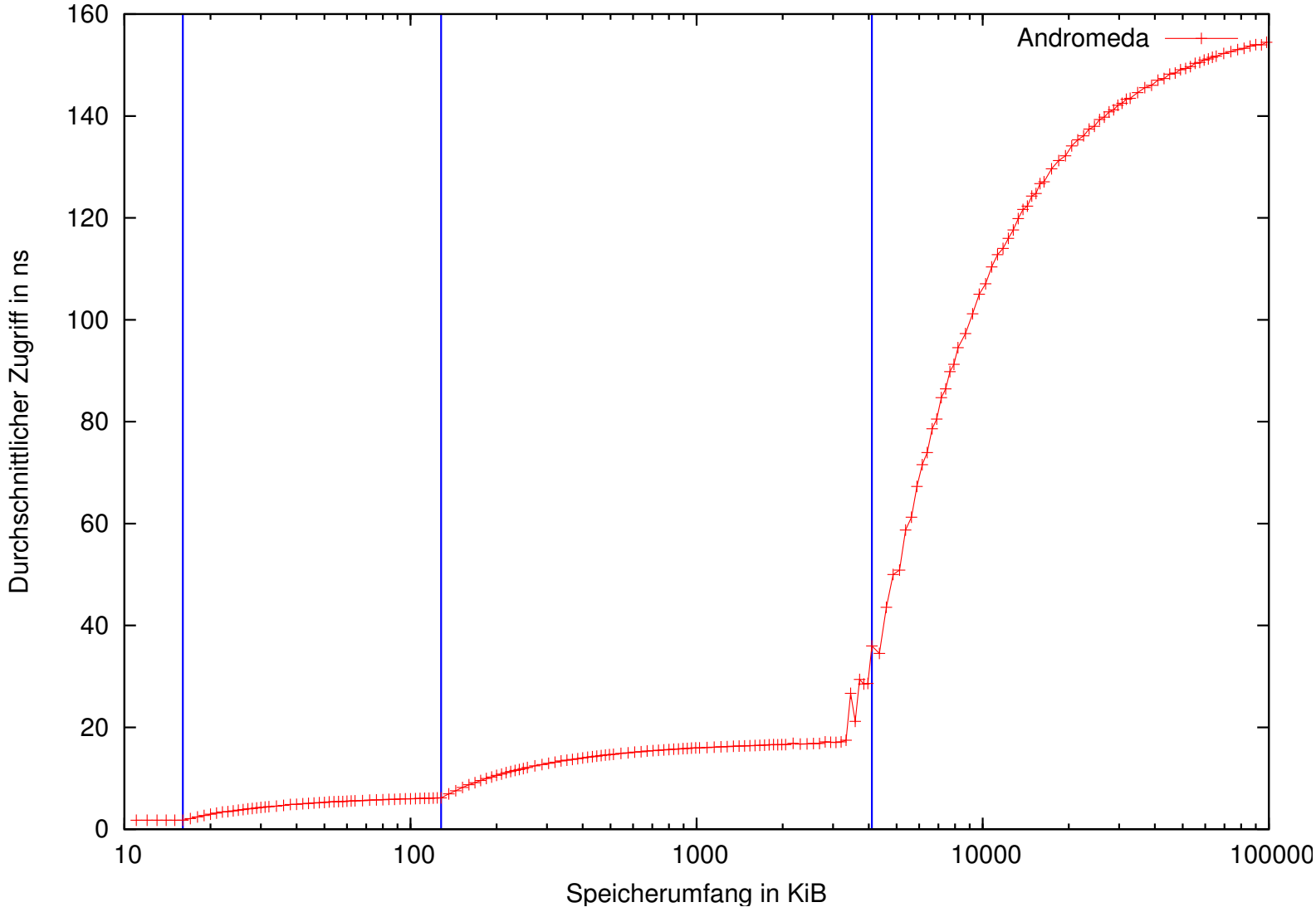
Brainbug02: Intel Quadcore i5-2500S CPU
Caches: L1 (32 KiB), L2 (256 KiB), L3 (6 MiB)

Gemessene Speicherzugriffszeiten: Pacioli



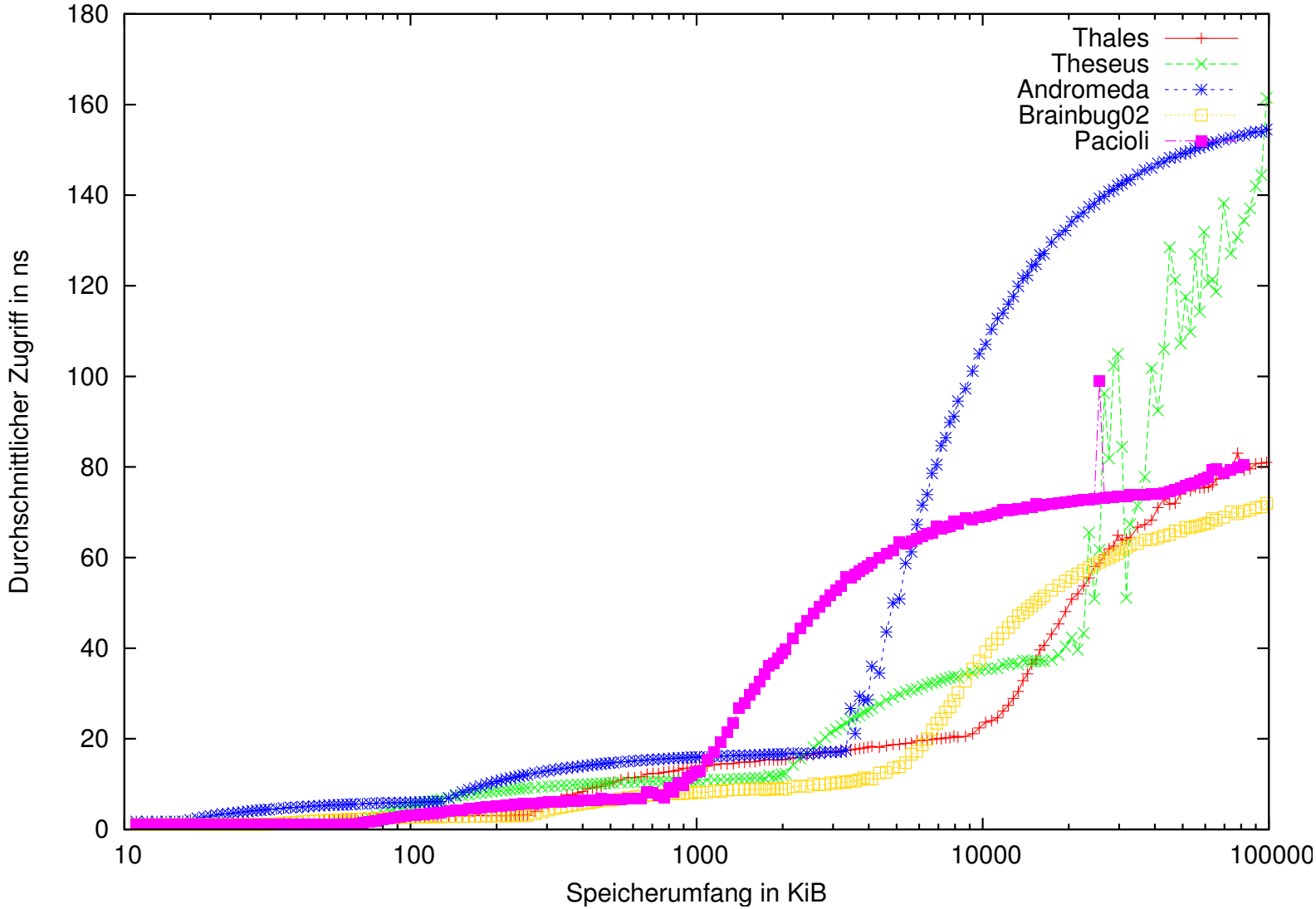
Pacioli: AMD Opteron 252
Caches: L1 (64 KiB), L2 (1 MiB)

Gemessene Speicherzugriffszeiten: Andromeda

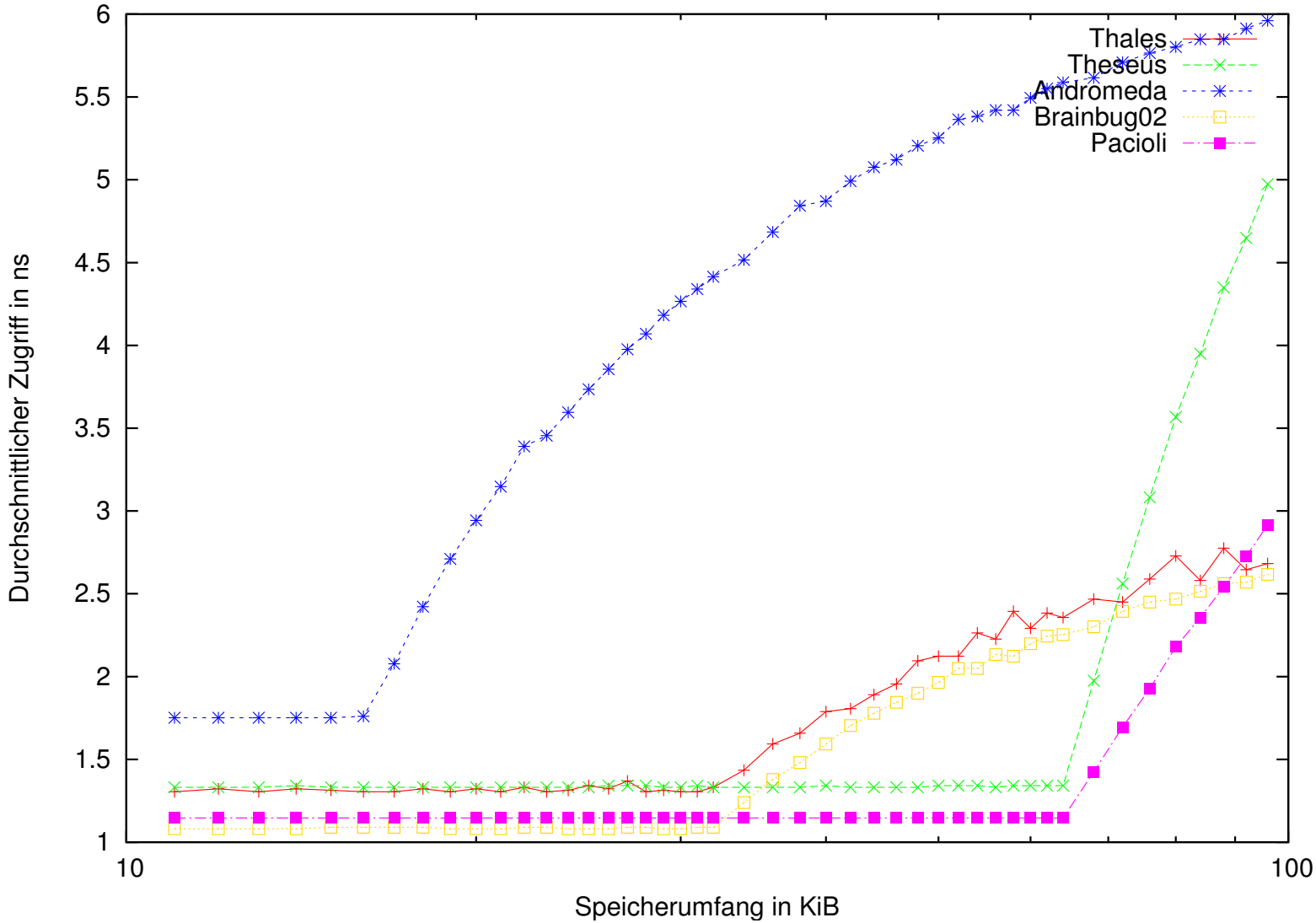


Andromeda: 2 SPARC-T4-Prozessoren mit je 8 Kernen mit je 8 Threads
Caches: L1 (16 KiB), L2 (128 KiB), L3 (4 MiB)

Gemessene Speicherzugriffszeiten: Vergleich



Gemessene Speicherzugriffszeiten im L1-Bereich



- 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 der Theseus sind es beispielsweise 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*).

- Diese Optimierungsechnik des Übersetzers bemüht sich darum, die Instruktionen (soweit dies entsprechend der Datenflussanalyse möglich ist) so anzuordnen, dass in der Prozessor-Pipeline keine Stockungen auftreten.
- Das lässt sich nur in Abhängigkeit des konkret verwendeten Prozessors optimieren, da nicht selten verschiedene Prozessoren der gleichen Architektur mit unterschiedlichen Pipelines arbeiten.
- Ein recht großer Gewinn wird erzielt, wenn ein vom Speicher geladener Wert erst sehr viel später genutzt wird.
- Beispiel: `x = a[i] + 5; y = b[i] + 3;`
  Hier ist es für den Übersetzer sinnvoll, zuerst die Ladebefehle für `a[i]` und `b[i]` zu generieren und erst danach die beiden Additionen durchzuführen und am Ende die beiden Zuweisungen.

axpy.c

```
// y = y + alpha * x
void axpy(int n, double alpha, const double* x, double* y) {
   for (int i = 0; i < n; ++i) {
      y[i] += alpha * x[i];
   }
}
```

- Dies ist eine kleine Blattfunktion, die eine Vektoraddition umsetzt. Die Länge der beiden Vektoren ist durch n gegeben, x und y zeigen auf die beiden Vektoren.
- Aufrufkonvention:

Variable	Register
n	%o0
alpha	%o1 und %o2
x	%o3
y	%o4

Resultat von gcc -O3

axpy.s

```
        add     %sp, -120, %sp
        cmp     %o0, 0
        st      %o1, [%sp+96]
        st      %o2, [%sp+100]
        ble     .LL5
        ldd     [%sp+96], %f12
        mov     0, %g2
        mov     0, %g1
.LL4:
        ldd     [%g1+%o3], %f10
        ldd     [%g1+%o4], %f8
        add     %g2, 1, %g2
        fmuld   %f12, %f10, %f10
        cmp     %o0, %g2
        faddd   %f8, %f10, %f8
        std     %f8, [%g1+%o4]
        bne     .LL4
        add     %g1, 8, %g1
.LL5:
        jmp     %o7+8
        sub     %sp, -120, %sp
```

- Ein *loop unrolling* fand hier nicht statt, wohl aber ein *instruction scheduling*.

Resultat von cc -fast

- Der C-Compiler von Sun generiert für die gleiche Funktion 241 Instruktionen (im Vergleich zu den 19 Instruktionen beim gcc).
- Der innere Schleifenkern mit 81 Instruktionen behandelt 8 Iterationen gleichzeitig. Das orientiert sich exakt an der Größe der *cache lines* der Architektur: $8 * \textbf{sizeof}(\textbf{double}) == 64$.
- Mit Hilfe der prefetch-Instruktion wird dabei jeweils noch zusätzlich dem Cache der Hinweis gegeben, die jeweils nächsten 8 Werte bei x und y zu laden.
- Der Code ist deswegen so umfangreich, weil
  - die Randfälle berücksichtigt werden müssen, wenn n nicht durch 8 teilbar ist und
  - die Vorbereitung recht umfangreich ist, da der Schleifenkern von zahlreichen bereits geladenen Registern ausgeht.

- 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 beispielsweise den anderen mitteilt, wenn Änderungen durchgeführt werden.
- 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

```
void axpy(int n, double alpha,
      const double* x, int incX, double* y, int incY) {
   for (int 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.

Verpackung der Vektoraddition als Thread

vectors.cpp

```
class AxpyThread {
   public:
      AxpyThread(int _n, double _alpha, double* _x, int _incX,
         double* _y, int _incY) :
            n(_n), alpha(_alpha), x(_x), incX(_incX),
            y(_y), incY(_incY) {
      }
      void operator()() {
         axpy(n, alpha, x, incX, y, incY);
      }
   private:
      int n;
      double alpha;
      double* x; int incX;
      double* y; int incY;
};
```

- Wie gehabt werden die Parameter wieder in eine Klasse verpackt.
- Doch wie werden die zu addierenden Vektoren auf die einzelnen Threads aufgeteilt?

vectors.cpp

```
void mt_axpy(int n, double alpha, double* x, int incX, double* y,
             int incY, int nofthreads) {
   assert(n > 0 && nofthreads > 0);
   thread axpy_thread[nofthreads];

   // spawn threads and pass parameters
   int chunk = n / nofthreads;
   int remainder = n % nofthreads;
   int nextX = 0; int nextY = 0;
   for (int i = 0; i < nofthreads; ++i) {
      int len = chunk;
      if (i < remainder) ++len;
      axpy_thread[i] = thread(AxpyThread(len, alpha,
         x + nextX * incX, incX, y + nextY * incY, incY));
      nextX += len; nextY += len;
   }

   // wait for all threads to finish
   for (int i = 0; i < nofthreads; ++i) {
      axpy_thread[i].join();
   }
}
```

Alternierende Aufteilung

bad-vectors.cpp

```
void mt_axpy(int n, double alpha, double* x, int incX, double* y,
             int incY, int nofthreads) {
   assert(n > 0 && nofthreads > 0);
   thread axpy_thread[nofthreads];

   // spawn threads and pass parameters
   int chunk = n / nofthreads;
   int remainder = n % nofthreads;
   int nextX = 0; int nextY = 0;
   for (int i = 0; i < nofthreads; ++i) {
      int len = chunk;
      if (i < remainder) ++len;
      axpy_thread[i] = thread(AxpyThread(len, alpha,
         x + i * incX, incX * nofthreads,
         y + i * incY, incY * nofthreads));
      nextX += len; nextY += len;
   }

   // wait for all threads to finish
   for (int i = 0; i < nofthreads; ++i) {
      axpy_thread[i].join();
   }
}
```

Laufzeiten je nach Aufteilung


```
theseus$ time vectors 10000000 1

real    0m0.78s
user    0m0.57s
sys     0m0.16s
theseus$ time bad-vectors 10000000 8

real    0m0.70s
user    0m2.33s
sys     0m0.17s
theseus$ time vectors 10000000 8

real    0m0.44s
user    0m0.66s
sys     0m0.17s
theseus$
```

- Der erste Parameter spezifiziert die Länge der Vektoren $n = 10^7$, der zweite die Zahl der Threads. (Dieses Beispiel könnte noch besser skalieren, wenn auch die Initialisierung der Vektoren parallelisiert worden wäre.)
- Die ungünstige alternierende Aufteilung führt dazu, dass Threads um die gleichen *cache lines* konkurrieren, so dass erhebliche Wartezeiten entstehen, die insgesamt den Vorteil der vielen Threads fast vollständig aufhoben.

```
theseus$ cputrack -c pic0=L2_rd_miss,pic1=L2L3_snoop_inv_sh \
> vectors 10000000 8
   time lwp      event      pic0      pic1
  0.302   2 lwp_create         0         0
  0.321   3 lwp_create         0         0
  0.382   4 lwp_create         0         0
  0.382   2   lwp_exit       224      1192
  0.382   3   lwp_exit       143       556
  0.442   5 lwp_create         0         0
  0.442   4   lwp_exit       134       278
  0.504   6 lwp_create         0         0
  0.504   5   lwp_exit        46       642
  0.562   7 lwp_create         0         0
  0.562   6   lwp_exit       128       373
  0.623   8 lwp_create         0         0
  0.623   7   lwp_exit       116       909
  0.664   9 lwp_create         0         0
  0.683   8   lwp_exit       126       400
  0.719   9   lwp_exit       172       486
  0.720   1       exit      4659      5437
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 5.437, während bei der ungünstigen Aufteilung diese auf 815.276 ansteigt...

```
theseus$ cputrack -c pic0=L2_rd_miss,pic1=L2L3_snoop_inv_sh \
> bad-vectors 10000000 8
   time lwp      event      pic0      pic1
  0.294   2 lwp_create         0         0
  0.301   3 lwp_create         0         0
  0.432   4 lwp_create         0         0
  0.518   5 lwp_create         0         0
  0.518   2   lwp_exit      1076    195702
  0.518   3   lwp_exit       186      1222
  0.692   6 lwp_create         0         0
  0.693   4   lwp_exit      4508    304384
  0.820   7 lwp_create         0         0
  0.821   5   lwp_exit       109      2176
  0.932   8 lwp_create         0         0
  0.932   6   lwp_exit       196       856
  0.947   9 lwp_create         0         0
  0.948   7   lwp_exit       131       959
  1.007   1       tick      3230       646
  1.067   8       tick       276    308152
  1.077   9       tick        25       702
  1.079   8   lwp_exit       369    308404
  1.110   9   lwp_exit       123       893
  1.112   1       exit     10263    815276
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 Kern extra.)

- 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/`

openmp-vectors.cpp

```
void axpy(int n, double alpha, const double* x, int incX,
      double* y, int incY) {
#pragma omp parallel for
   for (int 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.

```
theseus$ time openmp-vectors 100000000

real    0m7.81s
user    0m5.79s
sys     0m1.52s
theseus$ OMP_NUM_THREADS=8 time openmp-vectors 100000000

real        4.1
user        6.8
sys         1.5
theseus$ cd ../pthreads-vectors/
theseus$ time vectors 100000000 8

real    0m4.26s
user    0m6.76s
sys     0m1.54s
theseus$
```

- 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.

- 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.

OpenMP-Probleme beim Simpson-Verfahren

simpson.cpp

```
double simpson(double (*f)(double), double a, double b, int n) {
   assert(n > 0 && a <= b);
   double value = f(a)/2 + f(b)/2;
   double xleft;
   double x = a;
   for (int 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

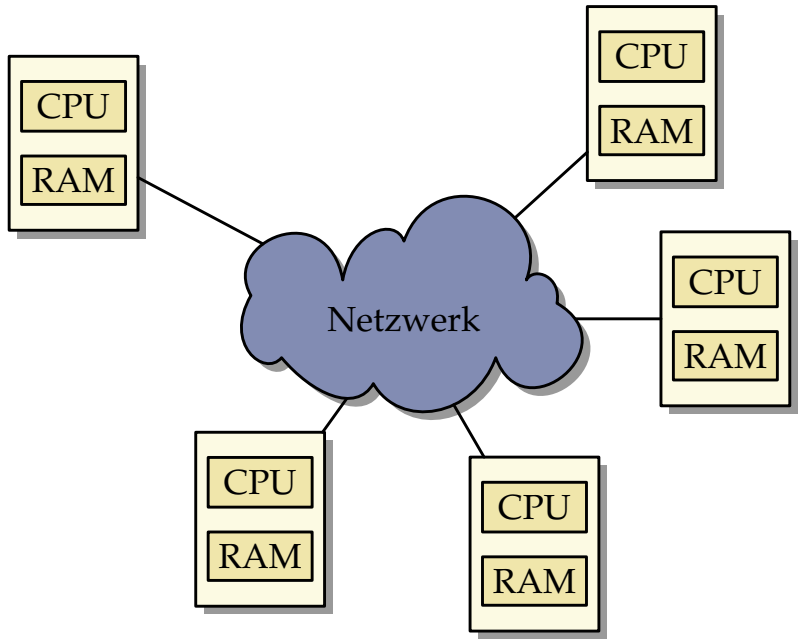
```
double simpson(double (*f)(double), double a, double b, int n) {
   assert(n > 0 && a <= b);
   double value = f(a)/2 + f(b)/2;
   double xleft;
   double x = a;
   double sum = 0;
#pragma omp parallel for \
      private(xleft) \
      lastprivate(x) \
      reduction(+:sum)
   for (int i = 1; i < n; ++i) {
      xleft = a + (i-1) * (b - a) / n;
      x = a + i * (b - a) / n;
      sum += f(x) + 2 * f((xleft + x)/2);
   }
   value += sum;
   value += 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

```
double mt_simpson(double (*f)(double), double a, double b, int n) {
   assert(n > 0 && a <= b);
   double sum = 0;
#pragma omp parallel reduction(+:sum)
   {
      int nofthreads = omp_get_num_threads();
      int nofintervals = n / nofthreads;
      int remainder = n % nofthreads;
      int i = omp_get_thread_num();
      int interval = nofintervals * i;
      int intervals = nofintervals;
      if (i < remainder) {
         ++intervals;
         interval += i;
      } else {
         interval += remainder;
      }
      double xleft = a + interval * (b - a) / n;
      double 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.



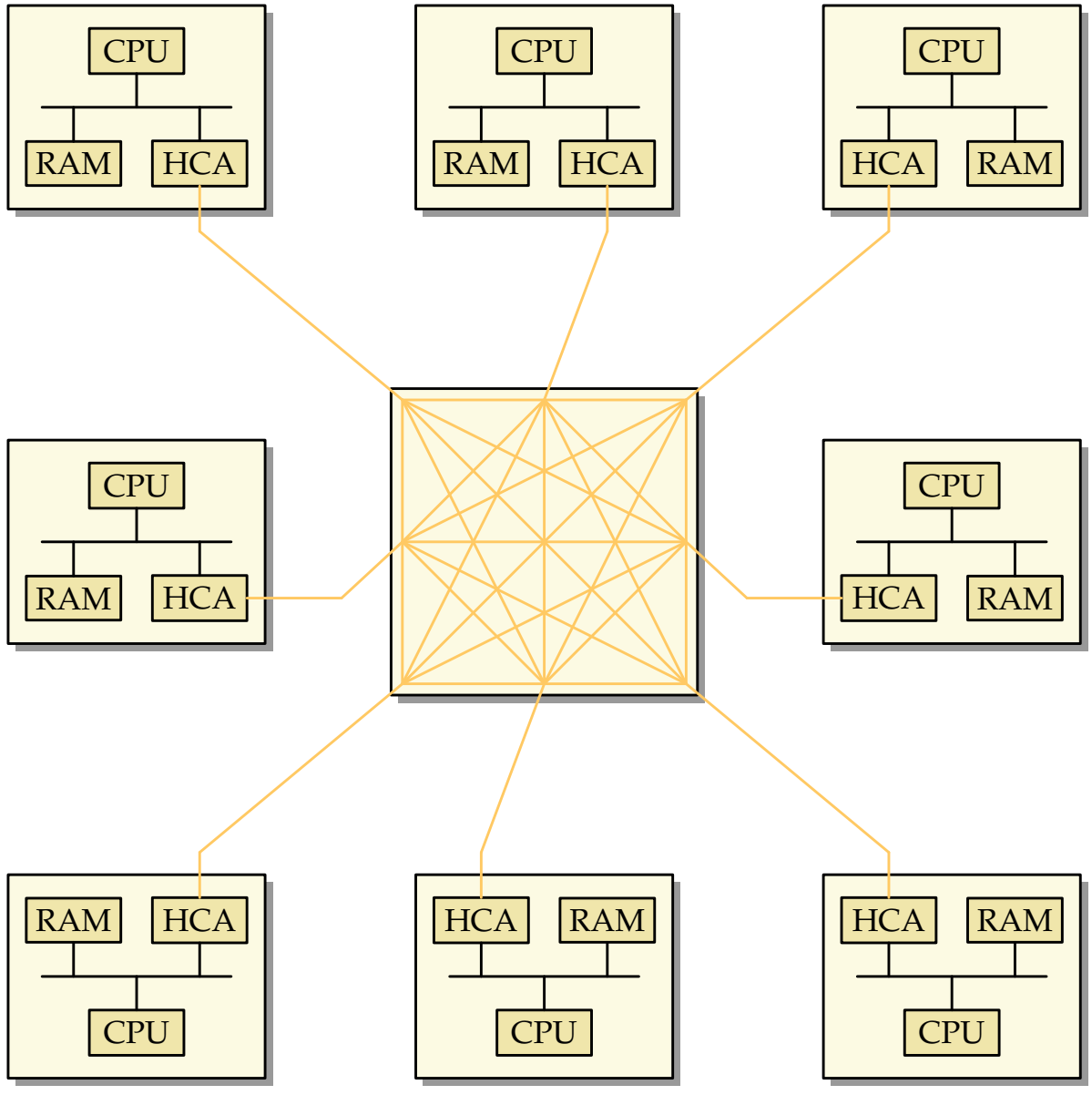
- Multicomputer bestehen aus einzelnen Rechnern mit eigenem Speicher, die über ein Netzwerk miteinander verbunden sind.
- Ein direkter Zugriff auf fremden Speicher ist nicht möglich.
- Die Kommunikation kann daher nicht über gemeinsame Speicherbereiche erfolgen. Stattdessen geschieht dies durch den Austausch 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. $150\mu s$ bei GbE auf Pacioli, ca. $500\mu s$ 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 die populärste Vernetzung bei Supercomputern: Zwei Supercomputer der TOP-10 und 210 der TOP-500 verwenden Infiniband (Stand: November 2012).
- 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 den 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 werden auf Programmebene (mit MPI) Latenzzeiten von unter 5 μs erreicht.

Fallbeispiel Titan

- Mit einer Rechenleistung von 17,59 Petaflop/s ist die Installation seit November 2012 der leistungsstärkste Rechner.
- Titan besteht aus 18.688 einzelnen Knoten des Typs Cray XK7, die jeweils mit einem AMD-Opteron-6274-Prozessor mit 16 Kernen und einer Nvidia-Tesla-K20X-GPU bestückt sind.
- Jede der GPUs hat 2.688 SPs (Stream-Prozessoren).
- Jeweils vier Knoten werden zu einer Blade zusammengefasst, die in insgesamt 200 Schränken verbaut sind.
- Für je zwei Knoten gibt es für die Kommunikation jeweils einen *Cray-Gemini-interconnect*-Netzwerkknoten, mit 10 Außenverbindungen, die topologisch in einem drei-dimensionalen Torus organisiert sind (vier redundant ausgelegte Verbindungen jeweils in den x- und z-Dimensionen und zwei in der y-Dimension, die topologisch in einem drei-dimensionalen Torus organisiert sind.
- Die Verbindungsstruktur wurde speziell für MPI ausgelegt und erlaubt auch zusätzlich die dreidimensionale Adressierung fremder Speicherbereiche.

- 12 Knoten mit jeweils 2 AMD-Opteron-252-Prozessoren (2,6 GHz) und 8 GiB Hauptspeicher.
- 20 Knoten mit jeweils 2 AMD-Opteron-2218-Prozessoren mit jeweils zwei Kernen und 8 GiB Hauptspeicher.
- 4 Knoten mit 2 Intel-Xeon-E5-2630-CPUs mit jeweils sechs Kernen und 128 GiB Hauptspeicher.
- Zu jedem Knoten gibt es eine lokale Festplatte.
- Über einen Switch sind sämtliche Knoten untereinander über Infiniband vernetzt.
- Zusätzlich steht noch ein GbE-Netzwerk zur Verfügung, an das jeder der Knoten angeschlossen ist.
- Pacioli bietet zwar auch IP-Schnittstellen für Infiniband an, aber diese sind nicht annähernd so effizient wie die direkte Schnittstelle.

- 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. Kürzlich (im September 2012) erschien die Version 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: `http://www.boost.org/doc/libs/1_53_0/doc/html/mpi.html`).
- Es stehen mehrere Open-Source-Implementierungen zur Verfügung:
  - OpenMPI: `http://www.open-mpi.org/` (wird von Sun verwendet, auf Theseus und Thales)
  - MPICH: `http://www.mpich.org/`
  - MVAPICH: `http://mvapich.cse.ohio-state.edu/` (spezialisiert auf Infiniband, auf Pacioli)

- 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*) oder individuell versandt werden.
- Die Kommunikation kann sowohl synchron als auch asynchron erfolgen.
- Die Prozesse können in einzelne Gruppen aufgesplittet werden.

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 = nofprocesses;
      } else {
         istringstream arg(argv[1]);
         if (!(arg >> n) || n <= 0) usage();
      }
   }
   // ...

   MPI_Finalize();

   if (rank == 0) {
      cout << setprecision(14) << sum << 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, 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 = nofprocesses;
   } else {
      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.

Ergebnisse berechnen und aggregieren

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(f, 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	**signed char**
MPI_SIGNED_CHAR	**signed char**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_SHORT	**signed short**
MPI_INT	**signed int**
MPI_LONG	**signed long**
MPI_UNSIGNED_SHORT	**unsigned short**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned long**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**
MPI_LONG_DOUBLE	**long double**
MPI_WCHAR	*wchar_t*
MPI_BOOL	*bool*
MPI_COMPLEX	*Complex*<**float**>
MPI_DOUBLE_COMPLEX	*Complex*<**double**>
MPI_LONG_DOUBLE_COMPLEX	*Complex*<**long double**>

Makefile

```
CXX :=          mpiCC
CXXFLAGS :=     -fast -library=stlport4
CC :=           mpiCC
CFLAGS :=       -fast -library=stlport4
```

- Die Option *mpi* sollte in *~/.options* genannt werden. Ggf. hinzufügen und erneut anmelden.
- Dann ist */opt/SUNWhpc/HPC8.2.1c/sun/bin* relativ weit vorne im Suchpfad.
- Statt den C++-Compiler von Sun mit *CC* direkt aufzurufen, wird stattdessen *mpiCC* verwendet, das alle MPI-spezifischen Header-Dateien und Bibliotheken automatisch zugänglich macht.
- Die Option *-fast* schaltet alle Optimierungen ein. Die Warnung, die deswegen ausgegeben wird, kann ignoriert werden.
- Hinweis: *mpiCC* unterstützt noch nicht den aktuellen C++-Standard von 2012.

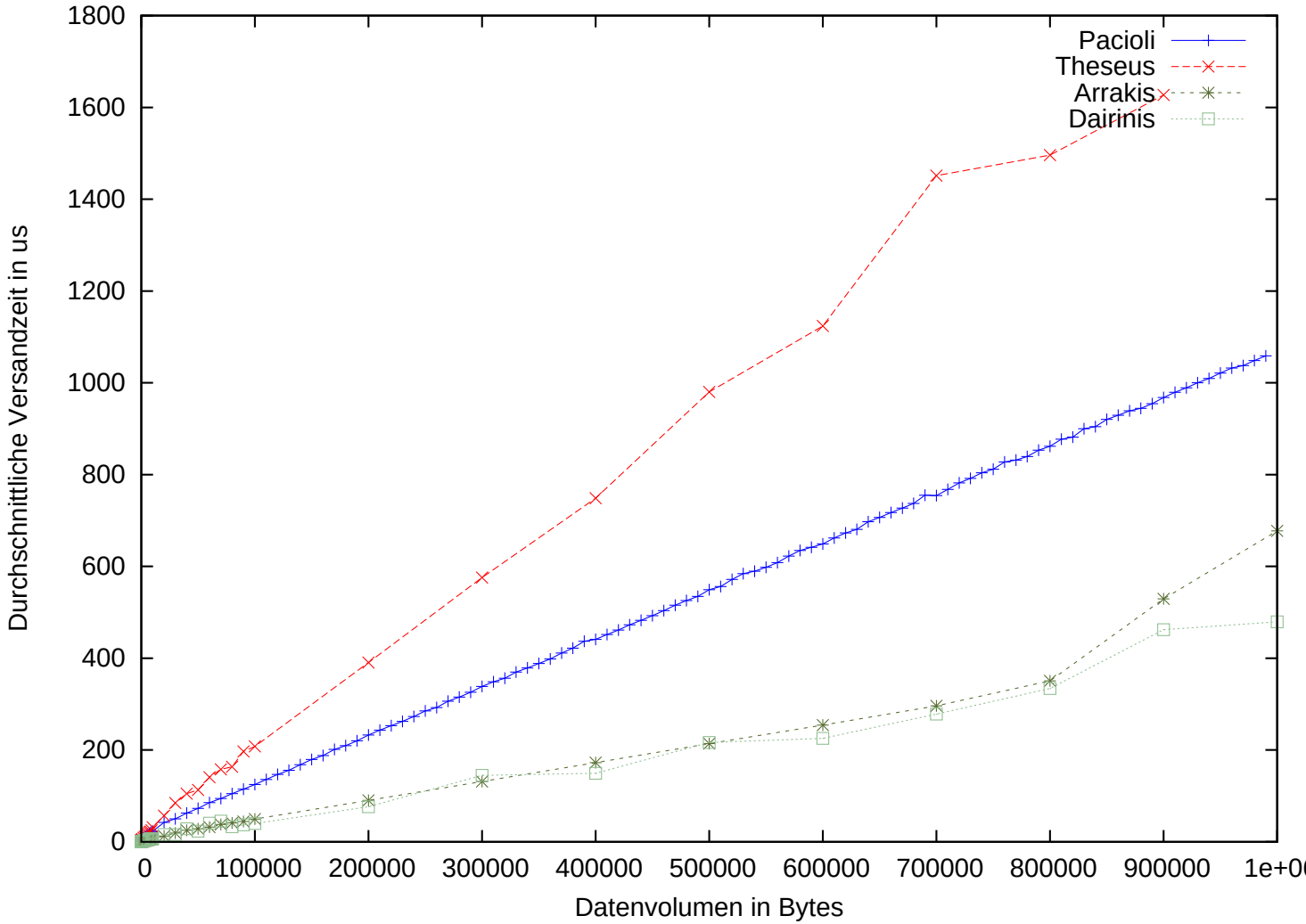
```
theseus$ f
Makefile  mpi-simpson.C
theseus$ make mpi-simpson
mpiCC -fast    mpi-simpson.C   -o mpi-simpson
CC: Warning: -xarch=native has been explicitly specified, or implicitly specif
theseus$ time mpirun -np 1 mpi-simpson 10000000
3.1415926535902

real    0m0.95s
user    0m0.87s
sys     0m0.03s
theseus$ time mpirun -np 4 mpi-simpson 10000000
3.1415926535897

real    0m0.39s
user    0m1.02s
sys     0m0.14s
theseus$
```

- 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.

```
theseus$ cat my-machines
malaga
rom
prag
lille
theseus$ time mpirun -hostfile my-machines -np 4 mpi-simpson 10000000
3.1415926535897

real    0m3.03s
user    0m0.31s
sys     0m0.06s
theseus$
```

- Die Option *-hostfile* ermöglicht 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; Arrakis: 2 Prozesse (AMD-Opteron Dual-Core, 3 GHz); Dairinis: 4 Prozesse (Intel Quad-Core, 2,5 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. (Arrakis: 0.8 μs; Dairinis: 0.9 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 Sklaven, die Aufträge entgegennehmen, verarbeiten und das Ergebnis zurücksenden.
- Dies wird an einem Beispiel der Matrix-Vektor-Multiplikation demonstriert, wobei diese Technik in diesem konkreten Fall nicht viel bringt.

mpi-gemv.cpp

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

   int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   int nofslaves; MPI_Comm_size(MPI_COMM_WORLD, &nofslaves);
   --nofslaves; assert(nofslaves > 0);

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

   MPI_Finalize();
}
```

Rolle eines Sklaven

mpi-gemv.cpp

```
static void gemv_slave() {
   int n;
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   double* x = new double[n];
   MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   double* row = new double[n];
   // receive tasks and process them
   for(;;) {
      // receive next task
      MPI_Status status;
      MPI_Recv(row, 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);
   }
   // release allocated memory
   delete[] x; delete[] row;
}
```

mpi-gemv.cpp

```
int n;
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
double* x = new double[n];
MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- Zu Beginn werden die Größe des Vektors und der Vektor selbst übermittelt.
- Da alle Sklaven 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, 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 nofslaves) {
   // broadcast parameters that are required by all slaves
   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 slaves
   int* tasks = new int[nofslaves];
   // ...

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

   // release allocated memory
   delete[] tasks;
}
```

- Zu Beginn werden die beiden Parameter n und x, die für alle Sklaven gleich sind, mit *Bcast* verteilt.
- Danach erhält jeder der Sklaven 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 slaves
   // remember the task for each of the slaves
int* tasks = new int[nofslaves];
int next_task = 0;
for (int slave = 1; slave <= nofslaves; ++slave) {
   if (next_task < n) {
      int row = next_task++; // pick next remaining task
      MPI_Send(A[row], n, MPI_DOUBLE, slave, NEXT_ROW,
         MPI_COMM_WORLD);
      // remember which task was sent out to whom
      tasks[slave-1] = row;
   } else {
      // there is no work left for this slave
      MPI_Send(0, 0, MPI_DOUBLE, slave, FINISH, MPI_COMM_WORLD);
   }
}
```

- Die Sklaven 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 slave = status.MPI_SOURCE;
   int row = tasks[slave-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, MPI_DOUBLE, slave, NEXT_ROW,
         MPI_COMM_WORLD);
      tasks[slave-1] = row;
   } else {
      // send notification that there is no more work to be done
      MPI_Send(0, 0, MPI_DOUBLE, slave, FINISH, MPI_COMM_WORLD);
   }
}
```

mpi-gemv.cpp

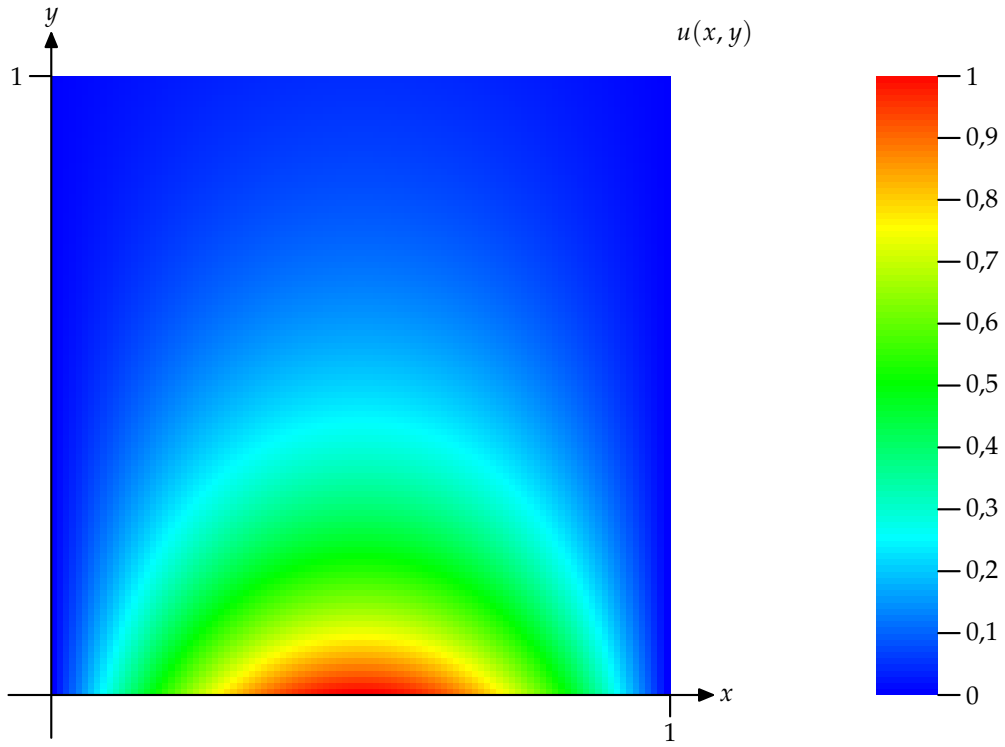
```
      MPI_Status status;
      MPI_Recv(&value, 1, MPI_DOUBLE,
         MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      int slave = status.MPI_SOURCE;
```

- Mit *MPI_ANY_SOURCE* wird angegeben, dass ein beliebiger Sender akzeptiert wird.
- Hier ist die Identifikation des Sklaven wichtig, damit das Ergebnis korrekt in y eingetragen werden kann. Dies erfolgt hier mit *status.Get_source*().

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 $u(0, y) = u(1, y) = 0$.

$$
\begin{array}{ccccc}
A_{0,N-1} & - & A_{1,N-1} & \cdots & A_{N-2,N-1} - A_{N-1,N-1} \\
| & & | & & | \quad\quad | \\
A_{0,N-2} & - & A_{1,N-2} & \cdots & A_{N-2,N-2} - A_{N-1,N-2} \\
\vdots & & \vdots & & \vdots \quad\quad \vdots \\
A_{0,1} & - & A_{1,1} & \cdots & A_{N-2,1} - A_{N-1,1} \\
| & & | & & | \quad\quad | \\
A_{0,0} & - & A_{1,0} & \cdots & A_{N-2,0} - A_{N-1,0}
\end{array}
$$

- 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.

np-jacobi.cpp

```
// single Jacobi iteration step
double single_jacobi_iteration(double** A, double** B, int n, int m) {
   for (int i = 1; i <= n; ++i) {
      for (int j = 1; j <= m; ++j) {
         B[i][j] = 0.25 * (A[i-1][j] + A[i][j-1] +
                           A[i][j+1] + A[i+1][j]);
      }
   }
   double maxdiff = 0;
   for (int i = 1; i <= n; ++i) {
      for (int j = 1; j <= m; ++j) {
         double diff = fabs(A[i][j] - B[i][j]);
         if (diff > maxdiff) maxdiff = diff;
         A[i][j] = B[i][j];
      }
   }
   return maxdiff;
}
```

- n entspricht hier $N-2$. A repräsentiert A_k und B die Näherungslösung des folgenden Iterationsschritts A_{k+1}.

np-jacobi.cpp

```
void initialize_A(double& Aij, int i, int j, int N) {
   const static double E_POWER_MINUS_PI = pow(M_E, -M_PI);
   if (j == 0) {
      Aij = sin(M_PI * ((double)i/(N-1)));
   } else if (j == N-1) {
      Aij = sin(M_PI * ((double)i/(N-1))) * E_POWER_MINUS_PI;
   } else {
      Aij = 0;
   }
}
```

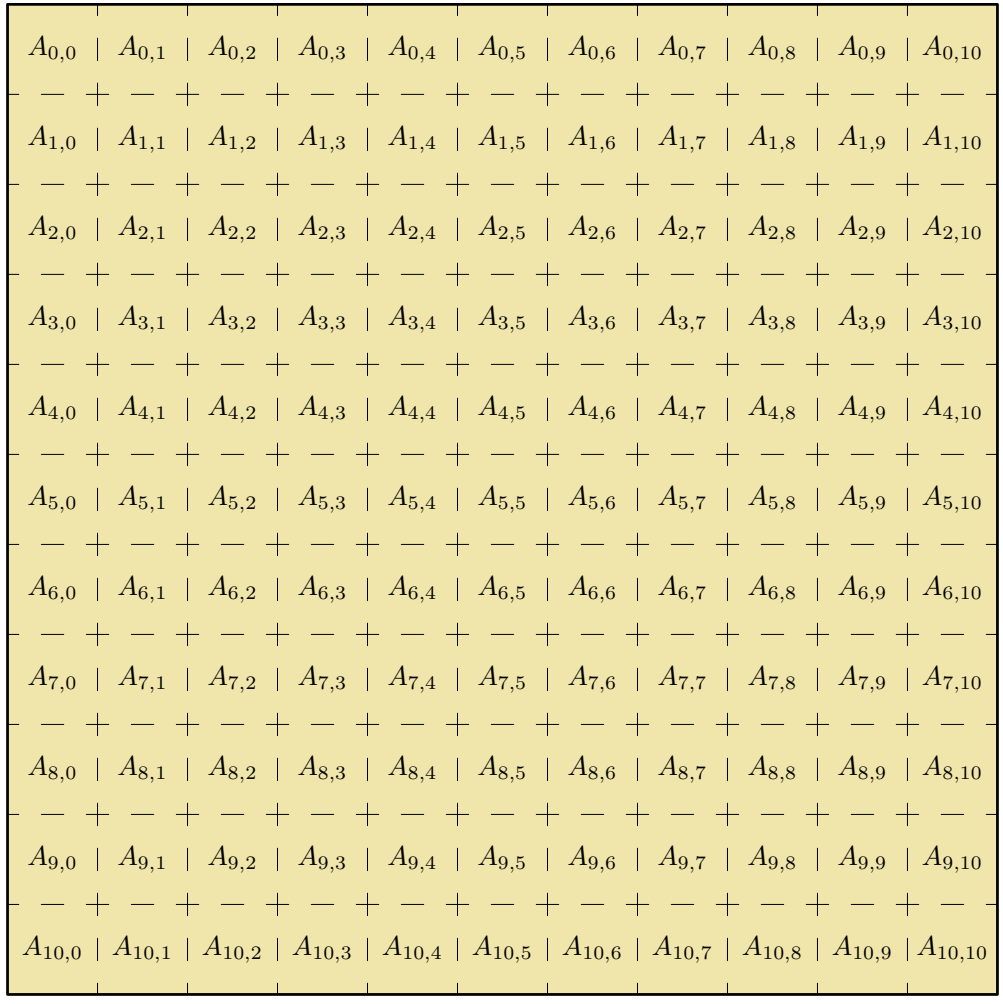
- Der gesamte Innenbereich wird mit 0 initialisiert.
- Für den Rand gelten die Randbedingungen $u(x,0) = sin(\pi x)$, $u(x,1) = sin(\pi x)e^{-\pi}$ und $u(0,y) = u(1,y) = 0$.

Nicht-parallelisierte Umsetzung

np-jacobi.cpp

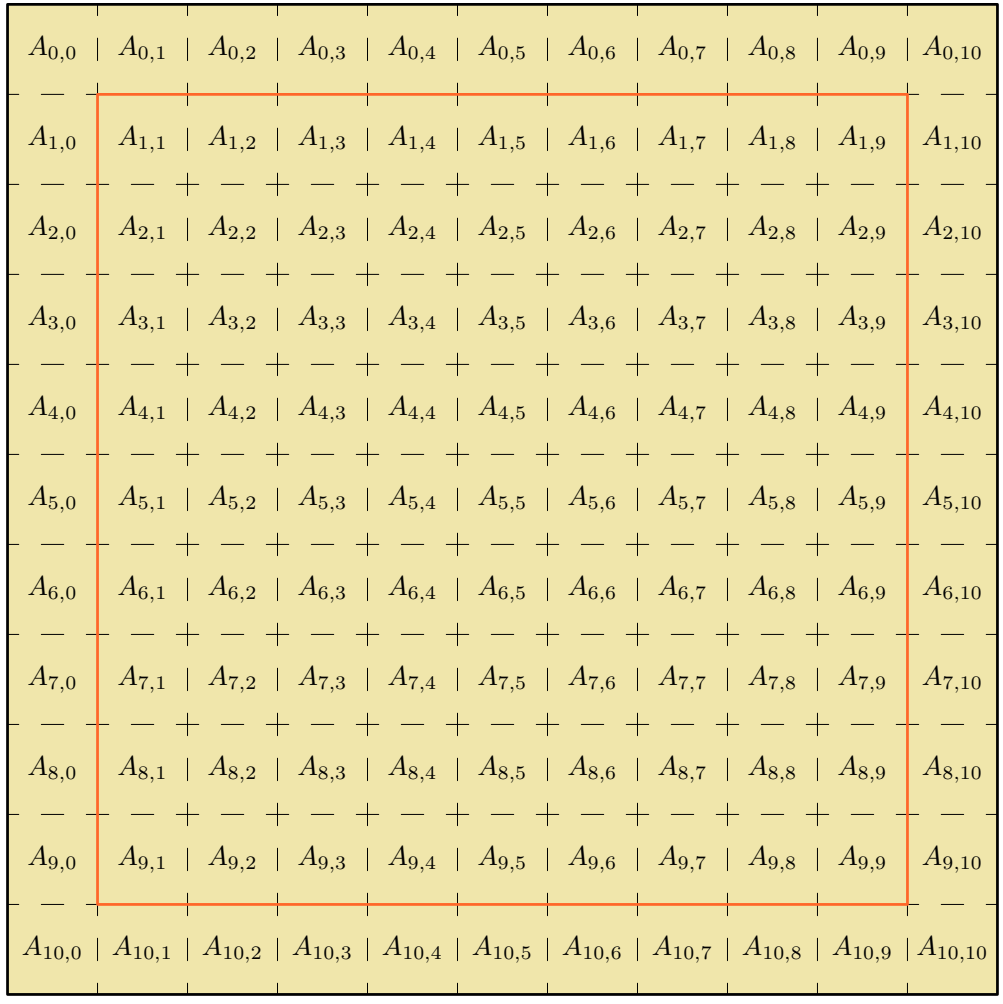
```
double** run_jacobi_iteration(int N, double eps) {
   int n = N-2;
   double** A = new double*[N]; assert(A);
   for (int i = 0; i < N; ++i) {
      A[i] = new double[N]; assert(A[i]);
      for (int j = 0; j < N; ++j) {
         initialize_A(A[i][j], i, j, N);
      }
   }
   double** B = new double*[N-1];
   for (int i = 1; i < N-1; ++i) {
      B[i] = new double[N-1]; assert(B[i]);
   }

   double maxdiff;
   do {
      maxdiff = single_jacobi_iteration(A, B, n, n);
   } while (maxdiff > eps);

   for (int i = 1; i < N-1; ++i) {
      delete[] B[i];
   }
   delete[] B;

   return A;
}
```

Eindimensionale Partitionierung



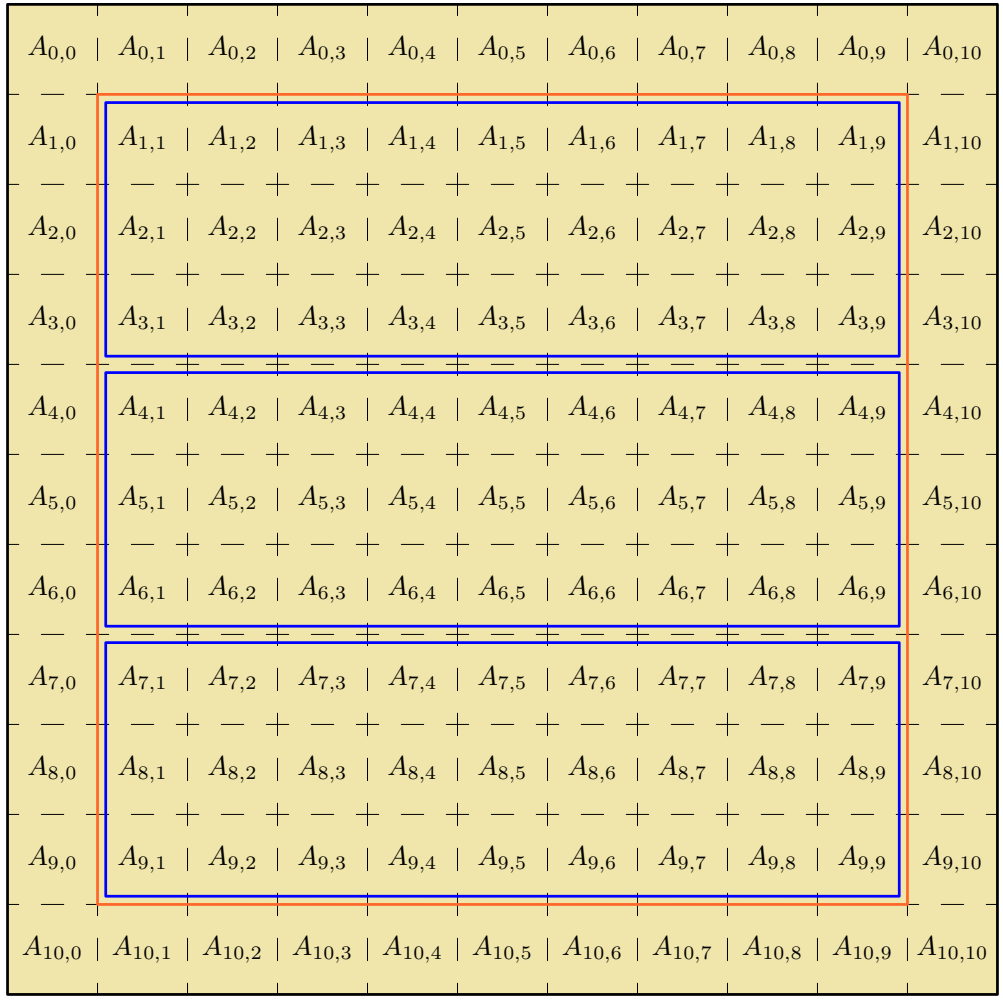
- Gegeben sei eine initialisierte Matrix A.

Eindimensionale Partitionierung

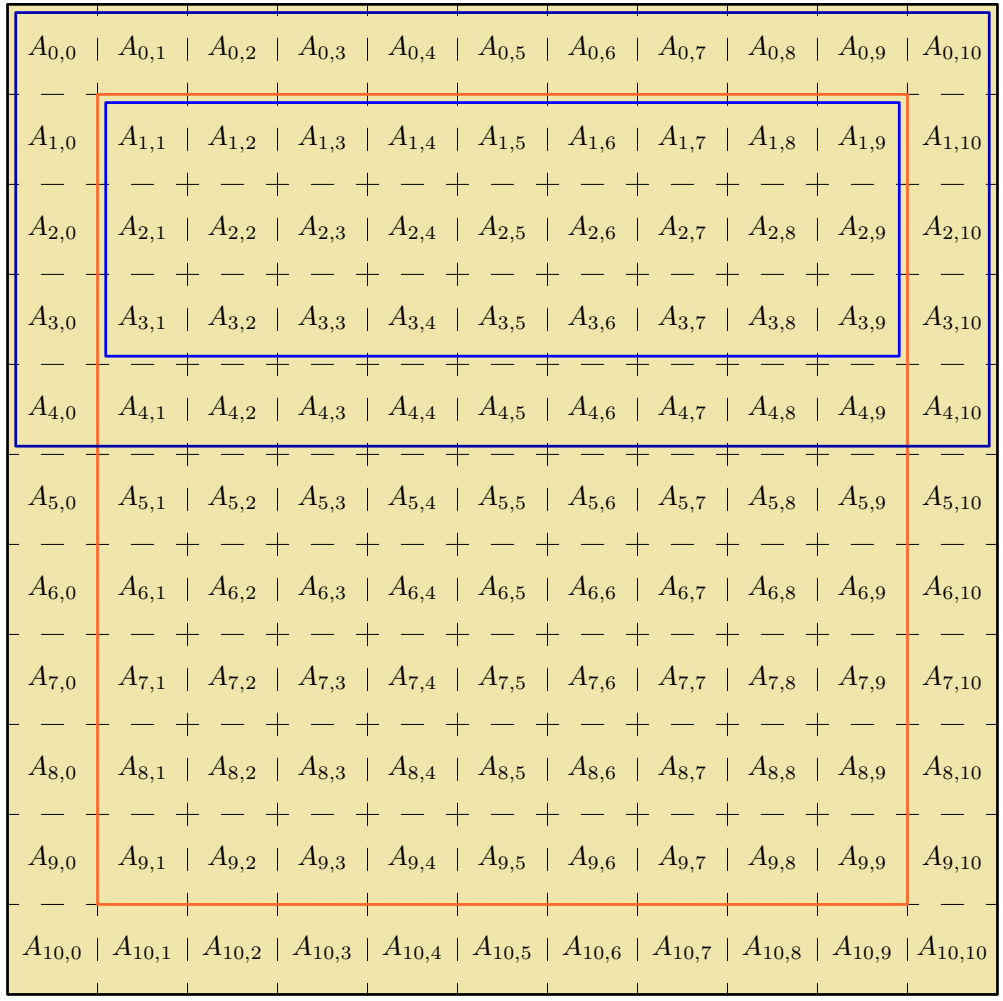


- Der Rand von A ist fest vorgegeben, der innere Teil ist näherungsweise zu bestimmen.

Eindimensionale Partitionierung

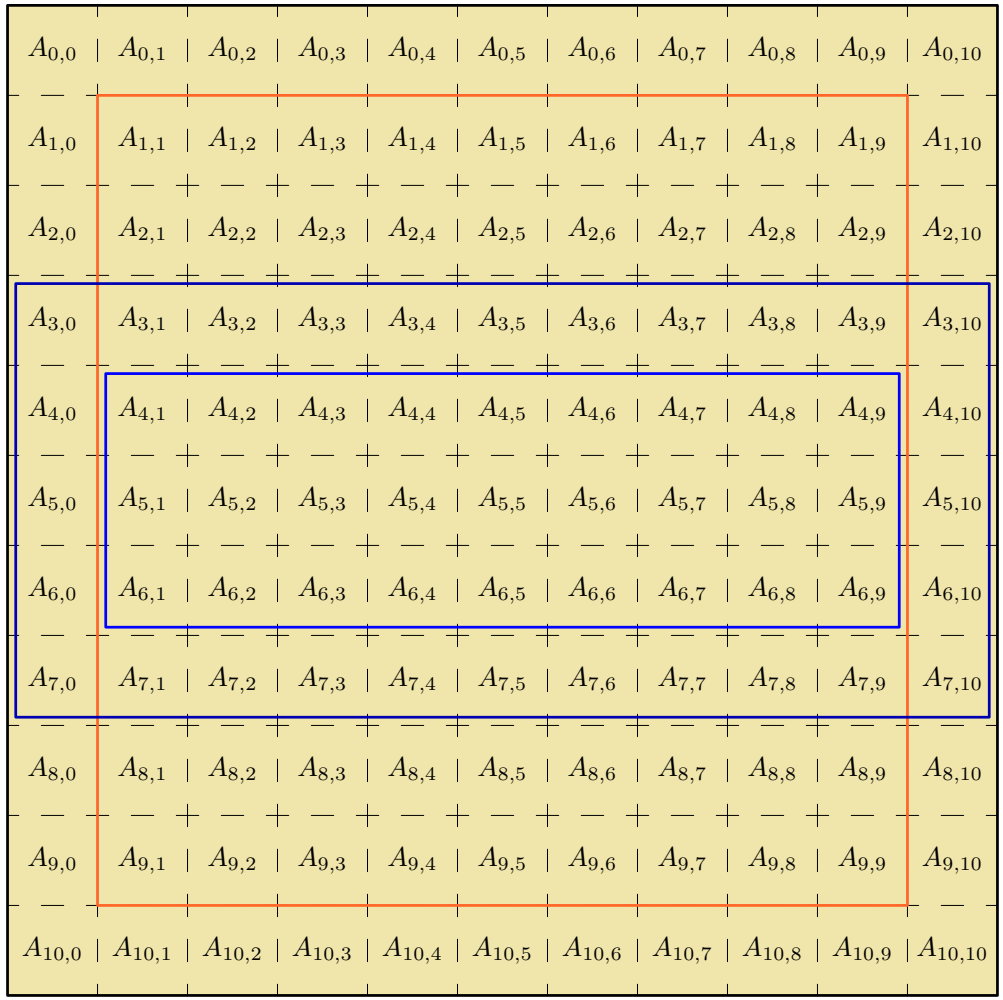


- Der innere Teil von A ist auf m Prozesse (hier $m = 3$) gleichmäßig aufzuteilen.



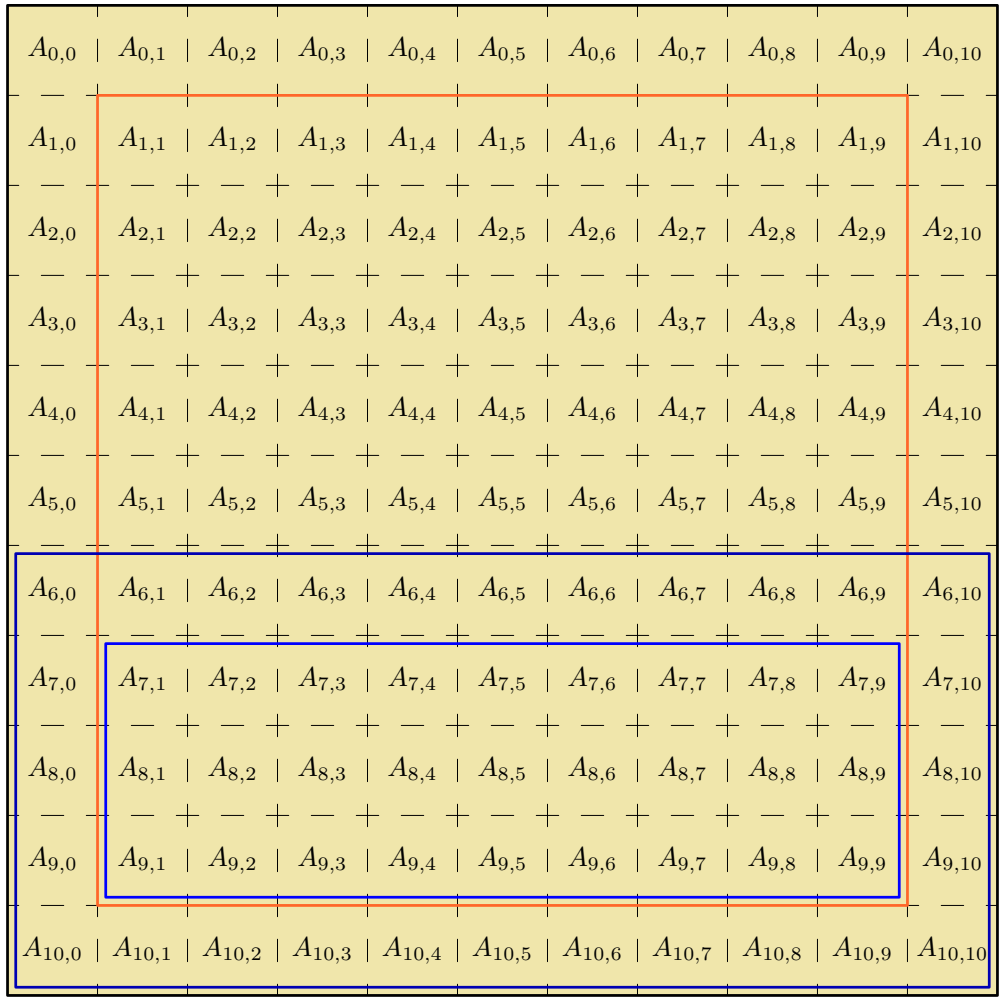
- Jeder der Prozesse benötigt lesenderweise den Rand. Soweit es sich nicht um den äußeren Rand handelt, muss dieser vom jeweiligen Nachbarn nach jedem Iterationsschritt erneut organisiert werden.

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.

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.

Kommunikationsaufwand bei einer ein-dimensionalen Partitionierung

- 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.

```
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*

mpi-deadlock.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 = 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) cout << "len = " << len << " survived" << 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.

Die Gefahr von Deadlocks bei *MPI_Send*


```
dairinis$ 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
^Cmpirun: killing job...

--------------------------------------------------------------------------
mpirun noticed that process rank 0 with PID 28203 on node dairinis exited on signal 0 (UNKNOWN SIGNAL).
--------------------------------------------------------------------------
2 total processes killed (some possibly by mpirun during cleanup)
mpirun: clean termination accomplished

dairinis$
```

- 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.

- 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* 120

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*.

Rahmen der parallelisierten Fassung

mpi-jacobi.cpp

```
// 1D-partitioned task
double** run_jacobi_iteration(int rank, int nofprocesses, int N, double eps) {
   int n = N-2;
   assert(nofprocesses <= n);
   int nofrows = n / nofprocesses;
   int remainder = n % nofprocesses;
   int first_row = rank * nofrows + 1;
   if (rank < remainder) {
      ++nofrows;
      if (rank > 0) first_row += rank;
   } else {
      first_row += remainder;
   }
   int last_row = first_row + nofrows - 1;

   // ... initialization ...

   for(;;) {
      double maxdiff = single_jacobi_iteration(A, B, nofrows, n);
      double global_max;
      MPI_Reduce(&maxdiff, &global_max, 1, MPI_DOUBLE,
         MPI_MAX, 0, MPI_COMM_WORLD);
      MPI_Bcast(&global_max, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
      if (global_max < eps) break;
      // ... message exchange with neighbors ...
   }

   // ... collect results in process 0 ...
   return Result;
}
```

mpi-jacobi.cpp

```
double** A = new double*[nofrows+2];
for (int i = 0; i <= nofrows+1; ++i) {
   A[i] = new double[N];
   for (int j = 0; j < N; ++j) {
      initialize_A(A[i][j], i + first_row, j, N);
   }
}
double** B = new double*[nofrows+1];
for (int i = 1; i <= nofrows; ++i) {
   B[i] = new double[N-1];
}
```

- Speicher für A und B wird hier nur im benötigten Umfang der entsprechenden Teilmatrizen belegt.
- A enthält auch die Randzonen von den jeweiligen Nachbarn. Bei B entfällt dies. Aus Gründen der Einfachheit werden aber A und B auf gleiche Weise indiziert.

mpi-jacobi.cpp

```
double global_max;
MPI_Reduce(&maxdiff, &global_max, 1, MPI_DOUBLE,
   MPI_MAX, 0, MPI_COMM_WORLD);
MPI_Bcast(&global_max, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if (global_max < eps) break;
```

- Zuerst wird das globale Maximum festgestellt, indem aus den lokalen Maxima mit *MPI_Reduce* im Prozess 0 das globale Maximum festgestellt wird und dieses danach mit *MPI_Bcast* an alle Prozesse übermittelt wird.
- Sobald die maximale Abweichung klein genug ist, wird die Schleife abgebrochen und auf eine weitere Kommunikation zwischen den Nachbarn verzichtet.

```
MPI_Status status;
// send highest row to the process which is next in rank
if (rank == 0) {
   MPI_Send(A[nofrows] + 1, n, MPI_DOUBLE, rank+1, 0,
      MPI_COMM_WORLD);
} else if (rank == nofprocesses-1) {
   MPI_Recv(A[0] + 1, n, MPI_DOUBLE, rank-1, 0,
      MPI_COMM_WORLD, &status);
} else {
   MPI_Sendrecv(A[nofrows] + 1, n, MPI_DOUBLE, rank+1, 0,
      A[0] + 1, n, MPI_DOUBLE, rank-1, 0,
      MPI_COMM_WORLD, &status);
}
```

- Das ist die Umsetzung der ersten Welle, bei der jeweils eine Zeile zur nächsthöheren Prozessnummer übermittelt wird bzw. eine Zeile von der nächstniedrigeren Prozessnummer entgegenzunehmen ist.

```
// send lowest row to the process which is previous in rank
if (rank == 0) {
   MPI_Recv(A[nofrows+1] + 1, n, MPI_DOUBLE, rank+1, 0,
      MPI_COMM_WORLD, &status);
} else if (rank == nofprocesses-1) {
   MPI_Send(A[1] + 1, n, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD);
} else {
   MPI_Sendrecv(A[1] + 1, n, MPI_DOUBLE, rank-1, 0,
      A[nofrows+1] + 1, n, MPI_DOUBLE, rank+1, 0,
      MPI_COMM_WORLD, &status);
}
```

- Danach folgt die Rückwelle, bei der jeweils eine Zeile zur nächstniedrigeren Prozessnummer übermittelt wird bzw. eine Zeile von der nächsthöheren Prozessnummer entgegenzunehmen ist.

Vereinfachung mit Nullprozessen

mpi-jacobi.cpp

```
int previous = rank == 0? MPI_PROC_NULL: rank-1;
int next = rank == nofprocesses-1? MPI_PROC_NULL: rank+1;
for(;;) {
   double maxdiff = single_jacobi_iteration(A, B, nofrows, n);
   double global_max;
   MPI_Reduce(&maxdiff, &global_max, 1, MPI_DOUBLE,
      MPI_MAX, 0, MPI_COMM_WORLD);
   MPI_Bcast(&global_max, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   if (global_max < eps) break;
   MPI_Status status;
   // send highest row to the process which is next in rank
   MPI_Sendrecv(A[nofrows] + 1, n, MPI_DOUBLE, next, 0,
      A[0] + 1, n, MPI_DOUBLE, previous, 0, MPI_COMM_WORLD, &status);
   // send lowest row to the process which is previous in rank
   MPI_Sendrecv(A[1] + 1, n, MPI_DOUBLE, previous, 0,
      A[nofrows+1] + 1, n, MPI_DOUBLE, next, 0, MPI_COMM_WORLD, &status);
}
```

- Die Behandlung der Spezialfälle am Rand lässt sich durch den Einsatz von sogenannten Nullprozessen vermeiden.
- Eine Kommunikation mit dem Prozess *MPI_PROC_NULL* findet nicht statt.

Einsammeln der Ergebnisse

mpi-jacobi.cpp

```
// collect results in process 0
double** Result = 0;
if (rank == 0) {
   Result = new double*[N]; assert(Result);
   for (int i = 0; i < N; ++i) {
      Result[i] = new double[N]; assert(Result[i]);
      for (int j = 0; j < N; ++j) {
         initialize_A(Result[i][j], i, j, N);
      }
   }
   for (int i = 1; i <= last_row; ++i) {
      memcpy(Result[i] + 1, A[i] + 1, n * sizeof(double));
   }
   for (int i = last_row+1; i <= n; ++i) {
      MPI_Status status;
      MPI_Recv(Result[i] + 1, n, MPI_DOUBLE,
         MPI_ANY_SOURCE, i, MPI_COMM_WORLD, &status);
   }
} else {
   for (int i = 1; i <= nofrows; ++i) {
      MPI_Send(A[i] + 1, n, MPI_DOUBLE, 0, first_row + i - 1,
         MPI_COMM_WORLD);
   }
}
return Result;
```

- 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.

mpi-jacobi-nb.cpp

```
// compute border zone
for (int j = 1; j <= n; ++j) {
   B[1][j] = 0.25 * (A[0][j] + A[1][j-1] + A[1][j+1] + A[2][j]);
   B[nofrows][j] = 0.25 * (A[nofrows-1][j] + A[nofrows][j-1] +
                           A[nofrows][j+1] + A[nofrows+1][j]);
}
// initiate non-blocking communication
MPI_Request req[4];
MPI_Irecv(A[0] + 1, n, MPI_DOUBLE, previous, 0, MPI_COMM_WORLD, &req[0]);
MPI_Irecv(A[nofrows+1] + 1, n, MPI_DOUBLE, next, 0, MPI_COMM_WORLD, &req[1]);
MPI_Isend(B[1] + 1, n, MPI_DOUBLE, previous, 0, MPI_COMM_WORLD, &req[2]);
MPI_Isend(B[nofrows] + 1, n, MPI_DOUBLE, next, 0, MPI_COMM_WORLD, &req[3]);
// computer inner zone
for (int i = 2; i < nofrows; ++i) {
   for (int j = 1; j <= n; ++j) {
      B[i][j] = 0.25 * (A[i-1][j] + A[i][j-1] + A[i][j+1] + A[i+1][j]);
   }
}
// prepare next iteration and compute maxdiff
double maxdiff = 0;
for (int i = 1; i <= nofrows; ++i) {
   for (int j = 1; j <= n; ++j) {
      double diff = fabs(A[i][j] - B[i][j]);
      if (diff > maxdiff) maxdiff = diff;
      A[i][j] = B[i][j];
   }
}
// block until initiated communication is finished
MPI_Status status; for (int i = 0; i < 4; ++i) MPI_Wait(&req[i], &status);
```

mpi-jacobi-nb.cpp

```
MPI_Request req[4];
MPI_Irecv(A[0] + 1, n, MPI_DOUBLE, previous, 0,
   MPI_COMM_WORLD, &req[0]);
MPI_Irecv(A[nofrows+1] + 1, n, MPI_DOUBLE, next, 0,
   MPI_COMM_WORLD, &req[1]);
MPI_Isend(B[1] + 1, n, MPI_DOUBLE, previous, 0,
   MPI_COMM_WORLD, &req[2]);
MPI_Isend(B[nofrows] + 1, n, MPI_DOUBLE, next, 0,
   MPI_COMM_WORLD, &req[3]);
```

- 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 131

mpi-jacobi-nb.cpp

```
// block until initiated communication is finished
for (int i = 0; i < 4; ++i) {
   MPI_Status status;
   MPI_Wait(&req[i], &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

```
Matrix*
run_jacobi_iteration(int rank, int nofprocesses, int N, double eps) {
   int n = N - 2; // without the surrounding border
   // create two-dimensional Cartesian grid
   int dims[2] = {0, 0}; int periods[2] = {false, false};
   MPI_Dims_create(nofprocesses, 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(MPI_COMM_WORLD, &rank); // update rank
   // ...
}
```

- 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};
MPI_Dims_create(nofprocesses, 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 periods[2] = {false, false};
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

```
// locate our own submatrix
int first_row, nof_rows, first_col, nof_cols;
get_submatrix(grid, dims, n, rank,
              first_row, first_col, nof_rows, nof_cols);

Matrix A(nof_rows + 2, nof_cols + 2, first_row, first_col);
Matrix B(nof_rows, nof_cols, first_row + 1, first_col + 1);
for (int i = A.firstRow(); i <= A.lastRow(); ++i) {
   for (int j = A.firstCol(); j <= A.lastCol(); ++j) {
      initialize_A(A(i, j), i, j, N);
   }
}
```

- *Matrix* ist eine Instantiierung der *GeMatrix*-Template-Klasse aus dem FLENS-Paket:
  typedef *flens*::*GeMatrix*<*flens*::*FullStorage*<**double**, *cxxblas*::*RowMajor*> > *Matrix*;
- *get_submatrix* ermittelt die Koordinaten der eigenen Teilmatrix, so dass dann A und B entsprechend deklariert werden können.

mpi-jacobi-2d.cpp

```
void get_submatrix(const MPI_Comm& grid, int* dims,
                   int n, int rank,
                   int& first_row, int& first_col,
                   int& nof_rows, int& nof_cols) {
   int coords[2];
   MPI_Cart_coords(grid, rank, 2, coords); // retrieve our position
   get_partition(n, dims[0], coords[0], first_row, nof_rows);
   get_partition(n, dims[1], coords[1], first_col, nof_cols);
}
```

- Die Funktion *MPI_Cart_coords* liefert die Gitterkoordinaten (in *coords*) für einen Prozess (*rank*).
- Der dritte Parameter gibt die Zahl der Dimensionen an, die niedriger als die Zahl der Dimensionen des Gitters sein kann.

mpi-jacobi-2d.cpp

```
void get_partition(int len, int nofprocesses, int rank,
                   int& start, int& locallen) {
   locallen = (len - rank - 1) / nofprocesses + 1;
   int share = len / nofprocesses;
   int remainder = len % nofprocesses;
   start = rank * share + (rank < remainder? rank: remainder);
}
```

- Diese Hilfsfunktion ermittelt in üblicher Weise das Teilintervall $[start, start + locallen - 1]$ aus dem Gesamtintervall $[0, len - 1]$ für den Prozess *rank*.

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

```
MPI_Datatype vector_type(int len, int stride) {
   MPI_Datatype datatype;
   MPI_Type_vector(
      /* count = */ len,
      /* blocklength = */ 1,
      /* stride = */ stride,
      /* element type = */ MPI_DOUBLE,
      /* newly created type = */ &datatype);
   MPI_Type_commit(&datatype);
   return datatype;
}
```

- Da dem linken und rechten Nachbarn in einem zweidimensionalen Gitter jeweils Spaltenvektoren zu übermitteln sind, lässt sich dies nicht mehr mit dem vorgegebenen Datentyp *MPI_DOUBLE* und einer Anfangsadresse erreichen.
- Um solche Probleme zu lösen, können mit Hilfe einiger Typkonstruktoren eigene 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 $o_i \in \mathbb{N}_0$ 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

mpi-jacobi-2d.cpp

```
struct buffer {
   double* buf;
   MPI_Datatype type;
};
struct buffer in_vectors[] = {
   {&A(A.firstRow(), A.firstCol() + 1), vector_type(nof_cols, 1)},
   {&A(A.lastRow(), A.firstCol() + 1), vector_type(nof_cols, 1)},
   {&A(A.firstRow() + 1, A.firstCol()),
      vector_type(nof_rows, nof_cols + 2)},
   {&A(A.firstRow() + 1, A.lastCol()),
      vector_type(nof_rows, nof_cols + 2)}
};
struct buffer out_vectors[] = {
   {&B(B.lastRow(), B.firstCol()), vector_type(nof_cols, 1)},
   {&B(B.firstRow(), B.firstCol()), vector_type(nof_cols, 1)},
   {&B(B.firstRow(), B.lastCol()), vector_type(nof_rows, nof_cols)},
   {&B(B.firstRow(), B.firstCol()), vector_type(nof_rows, nof_cols)}
};
```

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

mpi-jacobi-2d.cpp

```
int in_neighbor[] = {upper, lower, left, right};
int out_neighbor[] = {lower, upper, right, left};
```

- Passend zu den Austauschvektoren werden die zugehörigen Prozessnummern der Nachbarn spezifiziert.
- Dabei ist zu beachten, dass die Austauschvektoren bzw. die zugehörigen Prozessnummern jeweils paarweise zusammenpassen. (Darauf kann nur verzichtet werden, wenn die gesamte Kommunikation nicht-blockierend ist.)

mpi-jacobi-2d.cpp

```
for(;;) {
   double maxdiff = single_jacobi_iteration(A, B);
   double global_max;
   MPI_Reduce(&maxdiff, &global_max, 1, MPI_DOUBLE,
      MPI_MAX, 0, MPI_COMM_WORLD);
   MPI_Bcast(&global_max, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   if (global_max < eps) break;

   // exchange borders with our neighbors
   for (int dir = 0; dir < 4; ++dir) {
      MPI_Status status;
      MPI_Sendrecv(
         out_vectors[dir].buf, 1, out_vectors[dir].type,
            out_neighbor[dir], 0,
         in_vectors[dir].buf, 1, in_vectors[dir].type,
            in_neighbor[dir], 0,
         MPI_COMM_WORLD, &status
      );
   }
}
```

mpi-jacobi-2d-nb.cpp

```
double global_max;
do {
   // compute border zones
   // ...
   // exchange borders with our neighbors
   // ...
   // computer inner region
   // ...
   // block until initiated communication is finished
   // ...
   // check remaining error
   // ...
} while (global_max > eps);
```

- 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-nb.cpp

```
// compute border zones
for (int j = B.firstCol(); j <= B.lastCol(); ++j) {
   int i = B.firstRow();
   B(i,j) = 0.25 * (A(i-1,j) + A(i,j-1) + A(i,j+1) + A(i+1,j));
   i = B.lastRow();
   B(i,j) = 0.25 * (A(i-1,j) + A(i,j-1) + A(i,j+1) + A(i+1,j));
}
for (int i = B.firstRow(); i <= B.lastRow(); ++i) {
   int j = B.firstCol();
   B(i,j) = 0.25 * (A(i-1,j) + A(i,j-1) + A(i,j+1) + A(i+1,j));
   j = B.lastCol();
   B(i,j) = 0.25 * (A(i-1,j) + A(i,j-1) + A(i,j+1) + A(i+1,j));
}
```

mpi-jacobi-2d-nb.cpp

```
// exchange borders with our neighbors
MPI_Request req[8];
for (int dir = 0; dir < 4; ++dir) {
   MPI_Isend(out_vectors[dir].buf, 1, out_vectors[dir].type,
         out_neighbor[dir], 0, MPI_COMM_WORLD, &req[dir*2]);
   MPI_Irecv(in_vectors[dir].buf, 1, in_vectors[dir].type,
         in_neighbor[dir], 0, MPI_COMM_WORLD, &req[dir*2+1]);
}
// compute inner region
for (int i = B.firstRow() + 1; i < B.lastRow(); ++i) {
   for (int j = B.firstCol() + 1; j < B.lastCol(); ++j) {
      B(i,j) = 0.25 * (A(i-1,j) + A(i,j-1) + A(i,j+1) + A(i+1,j));
   }
}
```

mpi-jacobi-2d.cpp

```
MPI_Datatype matrix_type(const Matrix::View& submatrix) {
   MPI_Datatype datatype; MPI_Type_vector(
      /* count = */ submatrix.numRows(),
      /* blocklength = */ submatrix.numCols(),
      /* stride = */ submatrix.engine().leadingDimension(),
      /* element type = */ MPI_DOUBLE, &datatype);
   MPI_Type_commit(&datatype); return datatype;
}
```

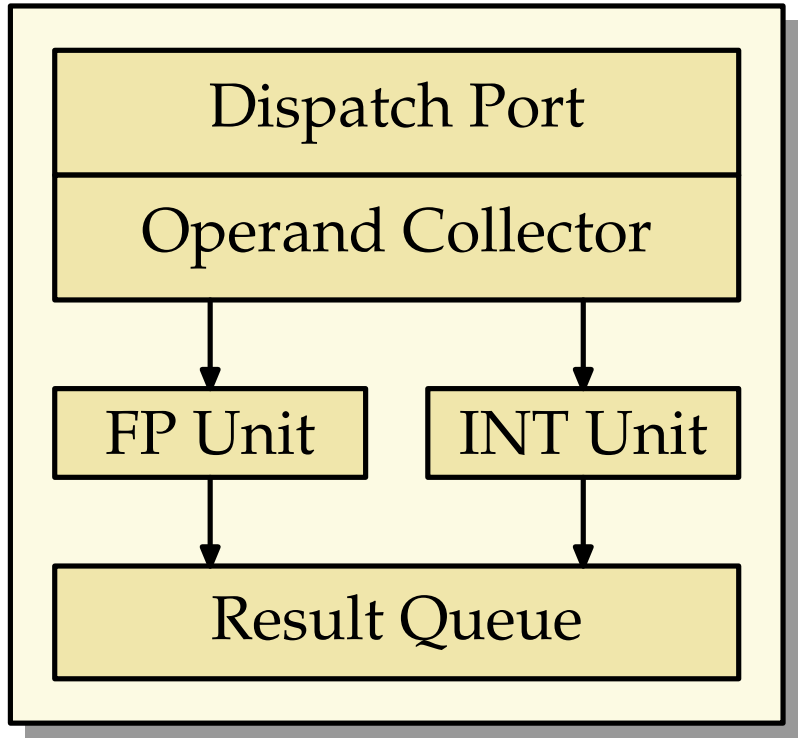
- Am Ende werden die einzelnen Teilmatrizen vom Prozess 0 eingesammelt.
- Hierzu werden vom Prozess 0 jeweils passende Sichten erzeugt (*Matrix*::*View*), für die *matrix_type* jeweils den passenden Datentyp generiert.
- Die Methode *leadingDimension* liefert hier den Abstand zwischen zwei aufeinanderfolgenden Zeilen. Da es sich hier um eine Teilmatrix handelt, kann dieser Abstand deutlich größer als *submatrix.numCols*() sein.

Einsammeln der Ergebnisse

mpi-jacobi-2d.cpp

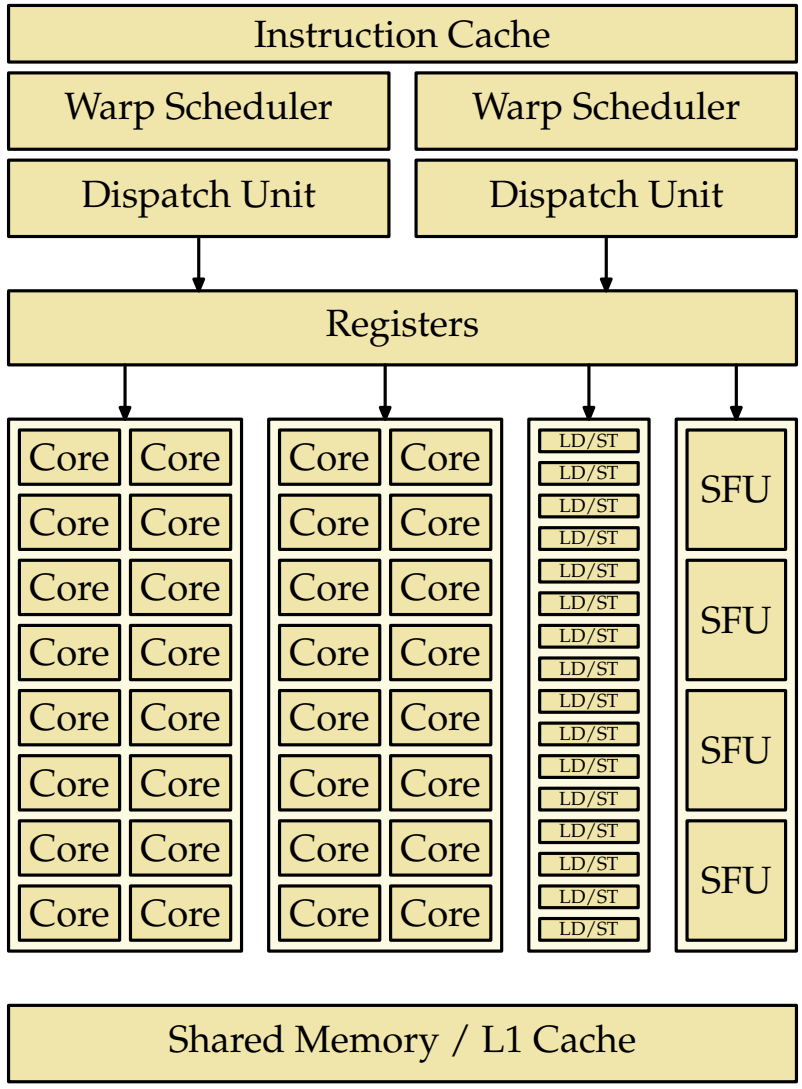
```
Matrix* Result = 0;
if (rank == 0) {
   Result = new Matrix(N, N, 0, 0); assert(Result);
   for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
         initialize_A((*Result)(i,j), i, j, N);
      }
   }
   for (int p = 0; p < nofprocesses; ++p) {
      int first_row, first_col, nof_rows, nof_cols;
      get_submatrix(grid, dims, n, p,
         first_row, first_col, nof_rows, nof_cols);
      ++first_row; ++first_col;
      Matrix::View submatrix(Result->engine().view(first_row, first_col,
                             first_row + nof_rows - 1,
                             first_col + nof_cols - 1,
                             first_row, first_col));
      if (p == 0) {
         submatrix = B;
      } else {
         MPI_Status status;
         MPI_Recv(&submatrix(submatrix.firstRow(), submatrix.firstCol()),
            1, matrix_type(submatrix), p, 0,
            MPI_COMM_WORLD, &status);
      }
   }
} else {
   MPI_Send(&B(B.firstRow(), B.firstCol()),
      B.numRows() * B.numCols(), MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
return Result;
```

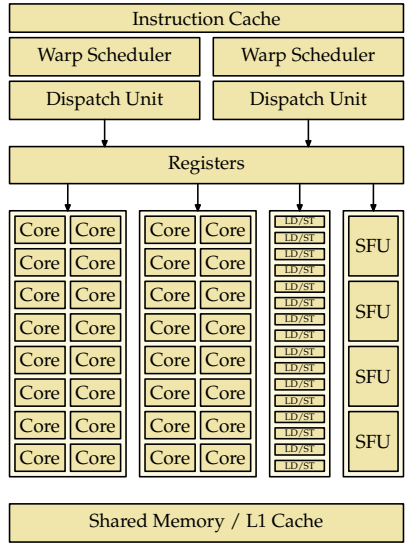
- Schon sehr früh gab es diverse Grafik-Beschleuniger, die der normalen CPU Arbeit abnahmen.
- Die im März 2001 von Nvidia eingeführte GeForce 3 Series führte programmierbares Shading ein.
- Im August 2002 folgte die Radeon R300 von ATI, die die Fähigkeiten der GeForce 3 deutlich erweiterte um mathematische 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) und CUDA (Compute Unified Device Architecture, von Nvidia). Wir beschäftigen uns hier mit CUDA, da es zur Zeit die größte Popularität genießt und bei uns auch zur Verfügung steht.



- 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

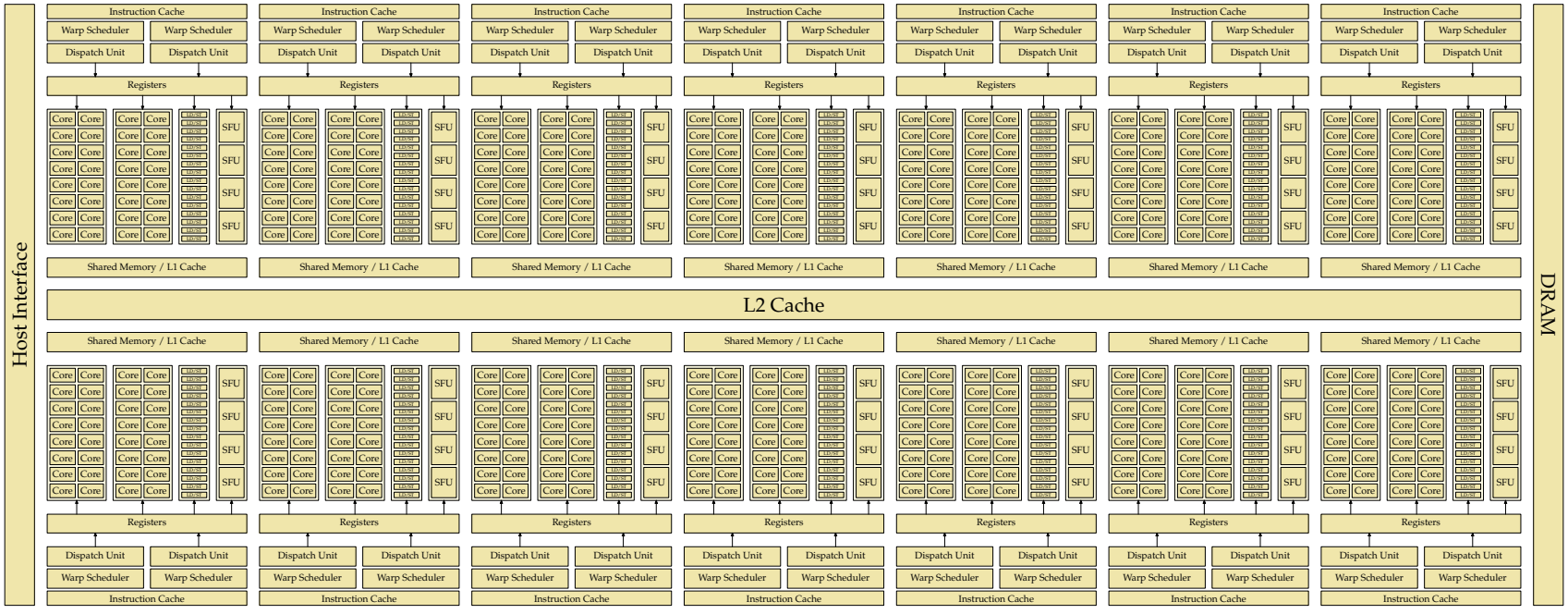




- Je nach Ausführung der GPU werden zahlreiche Kerne und weitere Komponenten zu einem Multiprozessor zusammengefasst.
- Auf Olympia hat ein Multiprozessor 32, auf Hochwanner 48 Kerne.
- Hinzu kommen Einheiten zum parallelisierten Laden und Speichern (*LD/ST*) und Einheiten für spezielle Operationen wie beispielsweise *sin* oder *sqrt* (*SFU*).

- 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.

GPU mit mehreren Multiprozessoren



- Je nach Ausführung der GPU werden mehrere Multiprozessoren zusammengefasst.
- Olympia bietet 14 Multiprozessoren mit insgesamt 448 Kernen an. Bei Hochwanner sind es nur 2 Multiprozessoren mit insgesamt 96 Kernen.

- 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 und hat Zugriff auf gemeinsamen Speicher.
- Threads eines Blockes können sich untereinander synchronisieren und über den gemeinsamen Speicher kommunizieren.
- Ein Block kann (bei uns auf Olympia und Hochwanner) bis zu 32 Warps bzw. 1024 Threads umfassen.

- Drei Speicherbereiche stehen zur Verfügung:
  - Lokaler Speicher: Steht nur einem Thread zur Verfügung.
  - Gemeinsamer Speicher: Steht den Threads eines Blocks gemeinsam zur Verfügung. (48 KiB pro Block auf Olympia und Hochwanner.)
  - Globaler Speicher: Steht allen Threads einer GPU zur Verfügung. (1 GiB auf Hochwanner, ca. 1,2 GiB auf Olympia.)
- Ab CUDA-Level 2.0 liegen alle drei Speicherbereiche im gleichen Adressraum.
- Neuere Implementierungen erlauben es, Teile des Hauptspeichers in den GPU-Adressraum abzubilden. Das hat sowohl Vor- als auch Nachteile. Die Entwicklung geht in Richtung einer besseren Integration von CPUs und GPUs mit dem Ziel, den Speicher gemeinsam effizienter und kooperativer zu nutzen.

- Wenn ein Warp auf den globalen Speicher zugreift, dann wird die höchste Effizienz erreicht, wenn die einzelnen Speicherzugriffe konsekutiv erfolgen.
- Wenn die Zugriffe nicht konsekutiv sind, erhöht sich die Zahl der zu ladenden Cache-Lines und die Bandbreite wird entsprechend reduziert, wodurch die Ausführung sich verzögert.
- Bei dem gemeinsamen und lokalen Speicher spielt das keine Rolle.
- Wenn bei einer Matrix die zu ladenden Vektoren ungünstig liegen (etwa Spaltenvektoren bei einer *row major*-Ordnung), dann kann es sich lohnen, den zu bearbeitenden Teil konsekutiv zu laden und im gemeinsamen Speicher abzulegen. (Siehe später folgendes Beispiel *mmu.cu*).

- Jeder Warp hat entweder den Zustand *active* oder *waiting*.
- Wenn ein Warp auf eine Barrier-Instruktion stößt, wechselt der Zustand von *active* auf *waiting*.
- Warps, die das Ausführungsende erreichen, verbleiben im Zustand *inactive*.
- Wenn ein Block keine aktiven Warps mehr hat und einige davon wegen einer Barrier-Instruktion warten, dann wechseln diese Warps von *waiting* auf *active*.

- Der Instruktionssatz ist proprietär und bis heute wurde von Nvidia kein öffentliches Handbuch dazu herausgegeben.
- Dank Wladimir J. van der Laan ist dieser jedoch weitgehend dekodiert und es gibt sogar einen an der Université de Perpignan entwickelten Simulator.
- 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.

- Zunächst wurden nur ganzzahlige Datentypen (32 Bit) und Gleitkommazahlen (**float**) unterstützt.
- Erst ab Level 1.3 kam die Unterstützung von **double** hinzu. Die GeForce GTX 470 auf Olympia unterstützt Level 2.0, die Quadro 600 auf Hochwanner unterstützt Level 2.1. (Beim Übersetzen mit *nvcc* sollte immer die Option „–gpu-architecture compute_20“ angegeben werden.)
- Ferner werden Zeiger unterstützt.
- Zugriffe sind (auch per Zeiger ab Level 2.0) möglich auf den gemeinsamen Speicher der GPU (*global memory*), auf den gemeinsamen Speicher eines Blocks (*shared memory*) und auf lokalen Speicher.

- Anwendungen, die GPUs ausnutzen möchten, werden zunächst auf der CPU gestartet.
- Eine Anwendung fällt somit in einen Teil, der auf der regulären CPU läuft und einen Teil, der von der GPU verarbeitet wird.
- Beide Teile haben völlig verschiedene Architekturen und Instruktionssätze.
- Die GPU ist der Anwendung nicht direkt zugänglich, sondern nur über einen speziellen Geräte-Treiber (unter Linux */dev/nvidia0* und */dev/nvidiactl*), der das Laden von Programmen, die Konfiguration eines Programmlaufs und den Austausch von Daten ermöglicht.

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 bzw. C++ (CUDA C bzw. 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/bin*), der CUDA C bzw. CUDA C++ unterstützt,
- eine zugehörige Laufzeitbibliothek (*libcudart.so* in */usr/local/cuda/lib*) und
- darauf aufbauende Bibliotheken (einschließlich BLAS und FFT).

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

vecadd.cu

```
__global__ void VecAdd(float* a, float* b, float* c) {
   int i = threadIdx.x;
   c[i] = a[i] + b[i];
}
```

- Der Übersetzer *nvcc* muss nach der statischen und semantischen Analyse vor der Code-Generierung eine Aufteilung entsprechend der Zielarchitektur durchführen, je nachdem ob der Programmtext für die GPU oder die reguläre CPU bestimmt ist.
- *VecAdd* ist ein Beispiel für eine Funktion, die für die GPU bestimmt ist. In CUDA wird eine Funktion, die von der CPU aufrufbar ist, jedoch auf der GPU ausgeführt wird, mit dem Schlüsselwort **__global__** gekennzeichnet. (Solche Funktionen werden *kernel* genannt.)
- *threadIdx.x* liefert hier die Thread-Nummer (mehr dazu später).

vecadd.ptx

```
        cvt.s32.u16     %r1, %tid.x;
        cvt.u64.s32     %rd1, %r1;
        mul.lo.u64      %rd2, %rd1, 4;
        ld.param.u64    %rd3, [__cudaparm__Z6VecAddPfS_S__a];
        add.u64         %rd4, %rd3, %rd2;
        ld.global.f32   %f1, [%rd4+0];
        ld.param.u64    %rd5, [__cudaparm__Z6VecAddPfS_S__b];
        add.u64         %rd6, %rd5, %rd2;
        ld.global.f32   %f2, [%rd6+0];
        add.f32         %f3, %f1, %f2;
        ld.param.u64    %rd7, [__cudaparm__Z6VecAddPfS_S__c];
        add.u64         %rd8, %rd7, %rd2;
        st.global.f32   [%rd8+0], %f3;
```

- 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.
- Die PTX-Instruktionssatz ist öffentlich:

  `http://docs.nvidia.com/cuda/pdf/ptx_isa_3.1.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.

vecadd.cubin.dis

```
000000: a0000001 04000780 cvt.rn.u32.u16 $r0, $r0.lo
000008: 30020009 c4100780 shl.u32 $r2, $r0, 0x00000002
000010: 2102e800          add.half.b32 $r0, s[0x0010], $r2
000014: 2102ec0c          add.half.b32 $r3, s[0x0018], $r2
000018: d00e0005 80c00780 mov.u32 $r1, g[$r0]
000020: d00e0601 80c00780 mov.u32 $r0, g[$r3]
000028: b0000204          add.half.rn.f32 $r1, $r1, $r0
00002c: 2102f000          add.half.b32 $r0, s[0x0020], $r2
000030: d00e0005 a0c00781 mov.end.u32 g[$r0], $r1
```

- Mit *ptxas* (wird normalerweise von *nvcc* implizit aufgerufen) lässt sich PTX-Assembler für eine vorgegebene GPU übersetzen.
- Der Instruktionssatz der GPU-Architektur ist bei Nvidia proprietär und wurde bislang nicht veröffentlicht.
- Obiger Text wurde mit Hilfe des von Wladimir J. van der Laan entwickelten Disassemblers erzeugt:
  `http://wiki.github.com/laanwj/decuda/`
- *$r0* ist ein Register, *s*[0*x0010*] adressiert den gemeinsamen Speicher des Blocks (*shared memory*) und *g*[*$ro*] eine mit einem Register indizierte Zelle im globalen Speicher der GPU (*global memory*).

Im Normalfall wird von *nvcc* ein C- bzw. C++-Programmtext erzeugt, der

- aus dem regulären für die CPU bestimmten Programmtext besteht,
- die für die GPU erzeugten CUBIN-Code als Daten-Arrays enthält und
- mit zahlreichen Aufrufen der Laufzeitbibliothek ergänzt wurde, die den Datenaustauch vornehmen, den CUBIN-Code in die GPU laden und zur Ausführung bringen (dies erfolgt mit Hilfe des entsprechenden Gerätetreibers).

Dieser C- bzw. C++-Code kann dann ganz regulär weiter übersetzt werden. Somit liefert *nvcc* im Normalfall am Ende ein fertiges, alleinstehendes ausführbares Programm, das alle benötigten Teile enthält.

vecadd.cu

```
float* cuda_a; cudaMalloc((void**)&cuda_a, N * sizeof(float));
float* cuda_b; cudaMalloc((void**)&cuda_b, N * sizeof(float));
float* cuda_c; cudaMalloc((void**)&cuda_c, N * sizeof(float));
cudaMemcpy(cuda_a, a, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(cuda_b, b, N * sizeof(float), cudaMemcpyHostToDevice);
VecAdd<<<1, N>>>(cuda_a, cuda_b, cuda_c);
cudaMemcpy(c, cuda_c, N * sizeof(float), cudaMemcpyDeviceToHost);
cudaFree(cuda_a); cudaFree(cuda_b); cudaFree(cuda_c);
```

- Die GPU und die CPU haben getrennten Speicher.
- Mit der Funktion *cudaMalloc* kann auf der GPU-Speicher belegt werden (*global memory*). Dieser wird nicht initialisiert.
- Der globale GPU-Speicher bleibt über den gesamten Programmlauf hinweg persistent, d.h. auch über mehrere Kernel-Aufrufe hinweg.
- Mit *cudaMemcpy* können Daten von der CPU in die GPU oder zurück kopiert werden.
- Mit *cudaFree* kann der GPU-Speicher wieder freigegeben werden.

vecadd.cu

```
VecAdd<<<1, N>>>(cuda_a, cuda_b, cuda_c);
```

- Jeder Aufruf eines Kernels ist mit einer Konfiguration der GPU verbunden, die in <<< ... >>> gefasst wird.
- Im einfachsten Falle enthält die Konfiguration zwei Parameter: Die Zahl der Blöcke (hier 1) und die Zahl der Threads pro Block (hier N).
- Die Zahl der Threads pro Block darf das von der jeweiligen GPU gesetzte Limit nicht überschreiten. Dies ist bei uns 1024.
- Alle Kernel-Parameter sollten entweder elementar sein (also etwa **int** oder **double**), Zeiger auf den globalen Speicher der GPU oder kleine Strukturen, die daraus bestehen. (Neuere CUDA-Versionen akzeptieren auch C++-Klassen.)

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 zweidimensional). (Bei neueren Versionen auch 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).
- *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 Beschänkungen sind bei der Dimensionierung zu berücksichtigen. Sonst kann der Kernel nicht gestartet werden.

simpson.cu

```
// numerical integration according to the Simpson rule
__global__ void simpson(Real a, Real b, Real* sums) {
   const int N = get_nofthreads();
   const int i = get_id();
   Real xleft = a + (b - a) / N * i;
   Real xright = xleft + (b - a) / N;
   Real xmid = (xleft + xright) / 2;
   sums[i] = (xright - xleft) / 6 * (f(xleft) + 4 * f(xmid) + f(xright));
}
```

- Bei der Vielzahl möglicher Threads kann häufig auf den Einsatz von Schleifen verzichtet werden.
- Aggregierende Funktionen existieren nicht. Deswegen ist es sinnvoll, die Einzelresultate im globalen Speicher abzulegen.
- *Real* wurde hier per **typedef** definiert (normalerweise **double**, bei älteren Karten muss ggf. **float** verwendet werden).

simpson.cu

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

- Nur mit **___device___** gekennzeichnete Funktionen dürfen auf der Seite der GPU aufgerufen werden.
- Es stehen auch diverse mathematischen Funktionen zur Verfügung.

In den auf der GPU laufenden Funktionen stehen spezielle Variablen zur Verfügung, die die Identifizierung bzw. Einordnung des eigenen Threads ermöglichen im bis zu drei-dimensional strukturierten Block und dem maximal zweidimensionalen Gitter von Blocks:

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
gridDim.x	Dimensionierung des Gitters für x
gridDim.y	Dimensionierung des Gitters für y

Identifizierung des eigenen Threads

simpson.cu

```
/* return unique id within a block */
__device__ int get_threadid() {
   return threadIdx.z * blockDim.x * blockDim.y +
  threadIdx.y * blockDim.x +
  threadIdx.x;
}

/* return block id a thread is associated to */
__device__ int get_blockid() {
   return blockIdx.x + blockIdx.y * gridDim.x;
}

/* return number of threads per block */
__device__ int get_nofthreads_per_block() {
   return blockDim.x * blockDim.y * blockDim.z;
}

/* return number of blocks */
__device__ int get_nofblocks() {
   return gridDim.x * gridDim.y;
}

/* return total number of threads */
__device__ int get_nofthreads() {
   return get_nofthreads_per_block() * get_nofblocks();
}

/* return id which is unique throughout all threads */
__device__ int get_id() {
   return get_blockid() * blockDim.x * blockDim.y * blockDim.z +
          get_threadid();
}
```

simpson.cu

```
int blocksize = max_threads_per_block();
if (blocksize > N) {
   blocksize = N;
} else {
   if (N % blocksize != 0) {
      cerr << cmdname << ": please select a multiple of "
         << blocksize << endl;
      exit(1);
   }
}
int nof_blocks = N / blocksize;
dim3 blockdim(blocksize, 1, 1);
dim3 griddim(nof_blocks, 1);
// ...
simpson<<<griddim, blockdim>>>(a, b, cuda_sums);
```

- Bei dem Simpson-Verfahren ist es sinnvoll, sowohl das Gitter als auch jeden Block eindimensional zu strukturieren.

simpson.cu

```
Real sums[N];
Real* cuda_sums; cudaMalloc((void**)&cuda_sums, N * sizeof(Real));
simpson<<<griddim, blockdim>>>(a, b, cuda_sums);
cudaMemcpy(sums, cuda_sums, N * sizeof(Real), cudaMemcpyDeviceToHost);
cudaFree(cuda_sums);
double sum = 0;
for (int i = 0; i < N; ++i) {
   sum += sums[i];
}
```

- Hier wird das Feld mit Summen zu Beginn im globalen Speicher der GPU belegt, dann mit der *simpson*-Funktion gefüllt, danach zum Speicher der CPU kopiert und schließlich aufsummiert.

jacobi.cu

```
typedef Real Matrix[BLOCK_SIZE+2][BLOCK_SIZE+2];

__global__ void jacobi(Matrix A, int nofiterations) {
   int i = threadIdx.x + 1;
   int j = threadIdx.y + 1;
   for (int it = 0; it < nofiterations; ++it) {
      Real Aij = 0.25 * (A[i-1][j] + A[i][j-1] + A[i][j+1] + A[i+1][j]);
      __syncthreads();
      A[i][j] = Aij;
      __syncthreads();
   }
}
```

- Beim Jacobi-Verfahren bietet sich eine zweidimensionale Organisation eines Blocks an.
- Es ist hierbei darauf zu achten, dass das Quadrat von *BLOCK_SIZE* noch kleiner als 512 ist. Hierfür bieten sich 16 (Zweier-Potenz) und 22 (maximaler Wert) an.

jacobi.cu

```
for (int it = 0; it < nofiterations; ++it) {
   Real Aij = 0.25 * (A[i-1][j] + A[i][j-1] + A[i][j+1] + A[i+1][j]);
   __syncthreads();
   A[i][j] = Aij;
   __syncthreads();
}
```

- Beim Jacobi-Verfahren werden bekanntlich die letzten Werte der Nachbarn eingeholt.
- Dies muss synchronisiert erfolgen. Solange alles in einen Warp passen würde, wäre das kein Problem, aber die Warps können unterschiedlich schnell vorankommen.
- Mit einem Aufruf von *__syncthreads()* wird eine Synchronisierung aller Threads eines Blocks erzwungen. D.h. erst wenn alle den Aufruf erreicht haben, geht es für alle weiter.

- Matrix-Matrix-Multiplikationen sind hochgradig parallelisierbar.
- Bei der Berechung von $C = A * B$ 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.
- O.B.d.A. betrachten wir nur quadratische $N \times N$ Matrizen mit $16 \mid N$.

mmm.cu

```
int main(int argc, char** argv) {
   cmdname = *argv++; --argc;
   if (argc != 2) usage();
   Matrix A; if (!read_matrix(*argv++, A)) usage(); --argc;
   Matrix B; if (!read_matrix(*argv++, B)) usage(); --argc;
   cout << "A = " << endl << A << endl;
   cout << "B = " << endl << B << endl;
   if (A.N != B.N) {
      cerr << cmdname << ": sizes of the matrices do not match" << endl;
      exit(1);
   }
   if (A.N % BLOCK_SIZE) {
      cerr << cmdname << ": size of matrices is not a multiply of "
         << BLOCK_SIZE << endl;
      exit(1);
   }

   A.copy_to_gpu();
   B.copy_to_gpu();
   Matrix C; C.resize(A.N); C.allocate_cuda_data();
   dim3 block(BLOCK_SIZE, BLOCK_SIZE);
   dim3 grid(A.N / BLOCK_SIZE, A.N / BLOCK_SIZE);

   mmm<<<grid, block>>>(A.cuda_data, B.cuda_data, C.cuda_data);

   C.copy_from_gpu();
   cout << "C = " << endl << setprecision(14) << C << endl;
}
```

- Es ist sinnvoll, eine Klasse für Matrizen zu verwenden, die die Daten sowohl auf der CPU als auch auf der GPU je nach Bedarf hält.
- Diese Klasse kann dann auch das Kopieren der Daten unterstützen.
- Generell ist es sinnvoll, Kopieraktionen soweit wie möglich zu vermeiden, indem etwa Zwischenresultate nicht unnötig von der GPU zur CPU kopiert werden.
- Eine Klasse hat auch den Vorteil, dass das die Freigabe der Datenflächen automatisiert wird.

mmm.cu

```
struct Matrix {
   unsigned int N;
   Real* data;
   bool cuda_allocated;
   Real* cuda_data;

   Matrix() :
      N(0), data(0), cuda_allocated(false), cuda_data(0) {
   }
   ~Matrix() {
      if (data) delete data;
      if (cuda_allocated) release_cuda_data();
   }

   // ...
};
```

- *N* ist die Größe der Matrix, *data* der Zeiger in den Adressraum der CPU, *cuda_data* der Zeiger in den Adressraum der GPU.

mmm.cu

```
bool copy_to_gpu() {
   if (!cuda_allocated) {
      if (!allocate_cuda_data()) return false;
   }
   return cudaMemcpy(cuda_data, data, N * N * sizeof(Real),
            cudaMemcpyHostToDevice) == cudaSuccess;
}

bool copy_from_gpu() {
   assert(cuda_allocated);
   return cudaMemcpy(data, cuda_data, N * N * sizeof(Real),
            cudaMemcpyDeviceToHost) == cudaSuccess;
}
```

- *copy_to_gpu* und *copy_from_gpu* kopieren die Matrix zur GPU und zurück.

mmm.cu

```
bool allocate_cuda_data() {
   if (cuda_allocated) return true;
   Real* cudap;
   if (cudaMalloc((void**)&cudap, N * N * sizeof(Real)) !=
         cudaSuccess) {
      return false;
   }
   cuda_data = cudap;
   cuda_allocated = true;
   return true;
}

void release_cuda_data() {
   if (cuda_data) {
      cudaFree(cuda_data);
      cuda_data = 0;
   }
}
```

- Mit *allocate_cuda_data* wird die Matrix im Adressraum der GPU belegt, mit *release_cuda_data* wieder freigegeben.

mmm.cu

```
bool resize(unsigned int N_) {
   if (N == N_) return true;
   Real* rp = new Real[N_ * N_];
   if (!rp) return false;
   if (data) delete data;
   release_cuda_data();
   data = rp; N = N_;
   return true;
}

Real& operator()(unsigned int i, unsigned int j) {
   return data[i*N + j];
}

const Real& operator()(unsigned int i, unsigned int j) const {
   return data[i*N + j];
}
```

- Mit *resize* wird die Größe festgelegt bzw. verändert. Die beiden ()-Operatoren dienen dem indizierten Zugriff (auf der Seite der CPU).

mmm-ab.cu

```
#define ELEMENT(m,i,j) ((m)[(i) * stride + (j)])

__global__ void mmm(Real* a, Real* b, Real* c) {
   unsigned int stride = gridDim.y * BLOCK_SIZE;
   unsigned int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
   unsigned int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;

   Real sum = 0;
   for (int k = 0; k < BLOCK_SIZE * gridDim.y; ++k) {
      sum += ELEMENT(a, row, k) * ELEMENT(b, k, col);
   }
   ELEMENT(c, row, col) = sum;
}
```

- Dies ist die triviale Implementierung, bei der jeder Thread $c_{row,col}$ direkt berechnet.
- Der Zugriff auf *a* ist hier ineffizient, da ein Warp hier nicht auf konsekutiv im Speicher liegende Werte zugreift.

mmm.cu

```
__shared__ Real ablock[BLOCK_SIZE][BLOCK_SIZE];
```

- 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.

Blockweise Multiplikation

mmm.cu

```
#define ELEMENT(m,i,j) ((m)[(i) * stride + (j)])

__global__ void mmm(Real* a, Real* b, Real* c) {
   __shared__ Real ablock[BLOCK_SIZE][BLOCK_SIZE];
   unsigned int stride = gridDim.y * BLOCK_SIZE;
   unsigned int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
   unsigned int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;

   Real sum = 0;
   for (int round = 0; round < gridDim.y; ++round) {
      ablock[threadIdx.y][threadIdx.x] =
         ELEMENT(a, row, round*BLOCK_SIZE + threadIdx.x);
      __syncthreads();

      #pragma unroll
      for (int k = 0; k < BLOCK_SIZE; ++k) {
         sum += ablock[threadIdx.y][k] *
            ELEMENT(b, round*BLOCK_SIZE + k, col);
      }
      __syncthreads();
   }
   ELEMENT(c, row, col) = sum;
}
```