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

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

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

Wie kann eine Parallelisierung erfolgen?

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

Pipelining

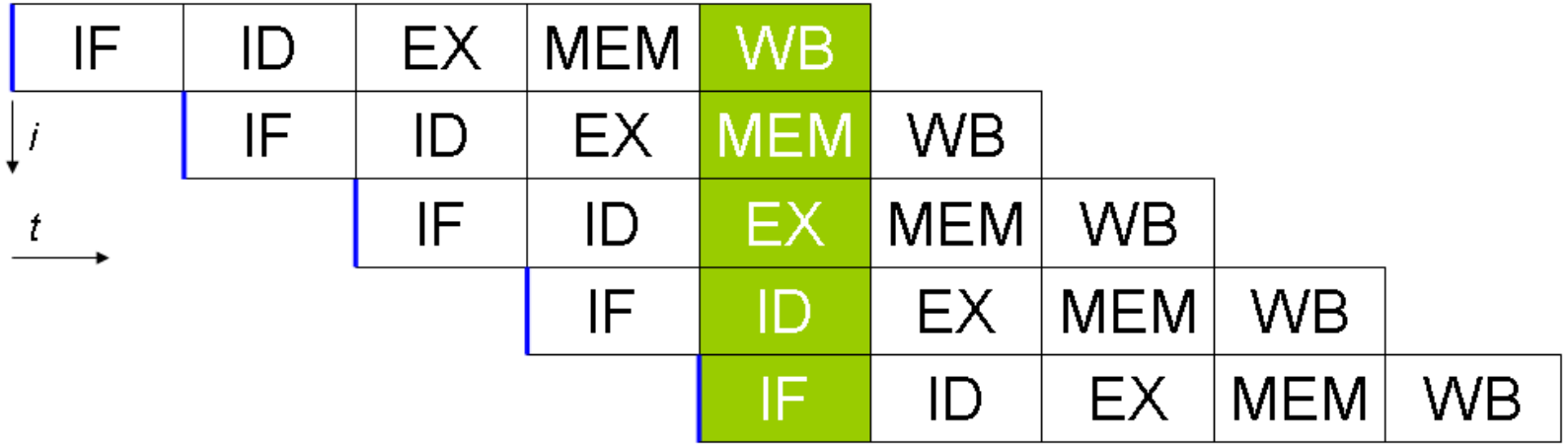
- Moderne Prozessoren arbeiten nach dem Fließbandprinzip: Über das Fließband kommen laufend neue Instruktionen hinzu und jede Instruktion wird nacheinander von verschiedenen Fließbandarbeitern bearbeitet.
- Dies parallelisiert die Ausführung, da unter günstigen Umständen alle Fließbandarbeiter gleichzeitig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

Typische Instruktionen auf Maschinenebene

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

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

Zerlegung in Arbeitsschritte



Eine einfache Aufteilung sieht folgende einzelne Schritte vor:

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

Das Diagramm wurde von Stephane Tsacas geklaut.

Branch Prediction

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

Parallelisierung von Speicherzugriffen

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

Flynns Aufteilung der Parallelrechner

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.

SIMD: Array-Prozessoren

- Die klassische Variante der SIMD sind die Array-Prozessoren.
- Eine Vielzahl von Prozessoren steht zur Verfügung mit zugehörigem Speicher, die diesen in einer Initialisierungsphase laden.
- Dann 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.

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

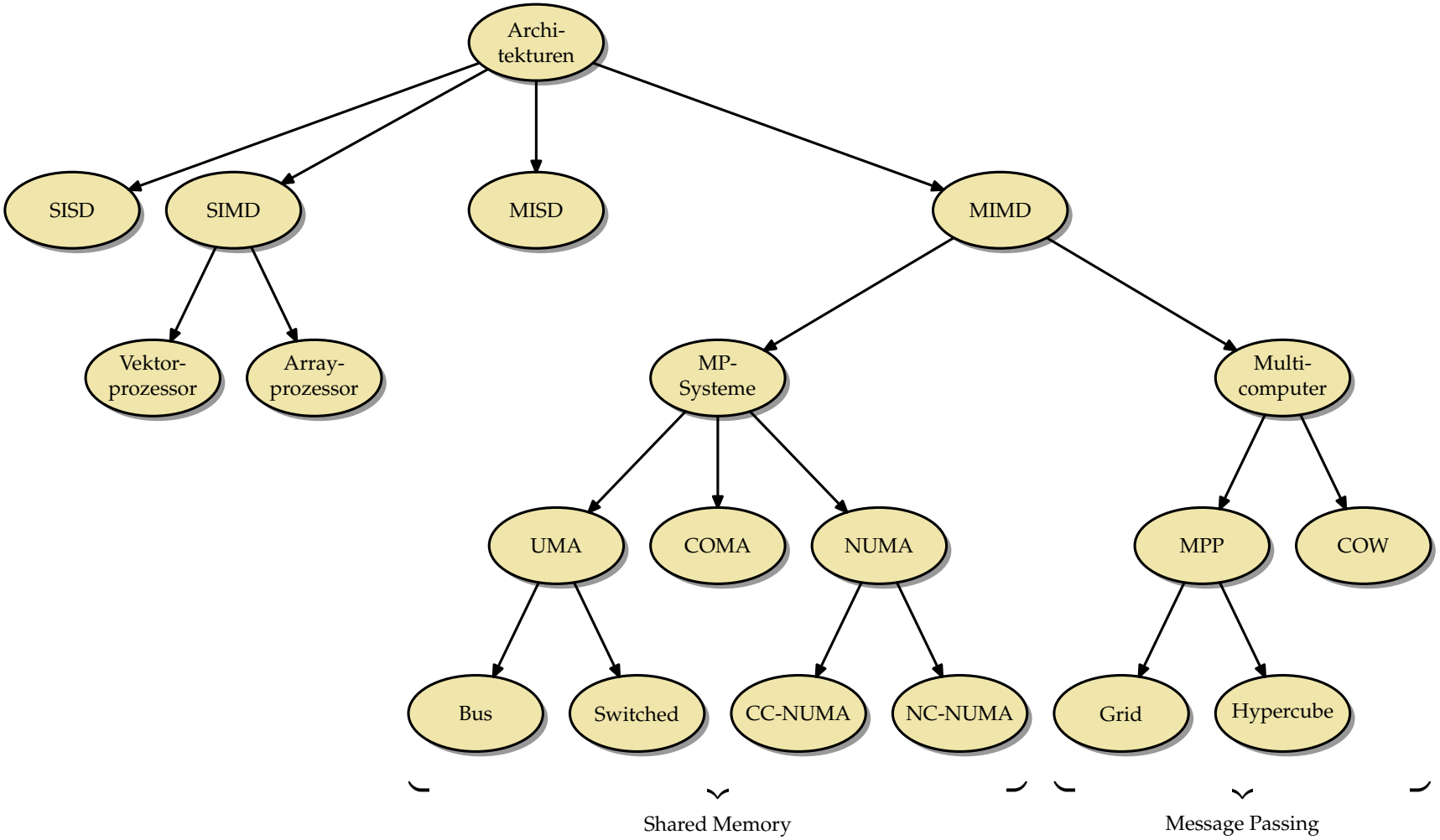
Die MMX- und SSE-Instruktionen des Pentium 4 setzen ebenfalls dieses Modell um.

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

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

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

Architekturen nach Tanenbaum



Unsere Übungsmaschinen

- 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).
- 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.

Unsere Übungsmaschinen

- 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.“

Die Ursprünge der Threads

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

Die Ursprünge der Threads

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

C++ und Threads

- Der ISO-Standard für C++ enthält keinerlei Sprachkonstrukte oder Bibliotheken zur Unterstützung von Threads.
- Gehani et al entwickelte bei den Bell Laboratories die Programmiersprache Concurrent C++ Anfang der 90er-Jahre. Sie gewann aber niemals eine größere Popularität. Siehe: Narain H. Gehani, "Capsules: A Shared Memory Access Mechanism for Concurrent C/C++," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 7, pp. 795-811, July 1993.
- Seit der Publikation des POSIX-Threads-Standards wurde dies auch zur Standardschnittstelle für C++.
- Der kommende ISO-Standard für C++ wird jedoch Threads berücksichtigen. Diese Schnittstelle orientiert sich an der, die von der *Boost Library* angeboten wird.

- Die *Boost Library* ist die wichtigste C++-Bibliothek jenseits der durch den ISO-Standard gegebenen Bibliotheken.
- Im Rahmen der *Boost Library* werden Bibliotheken entwickelt, die später in den nächsten Standard vorgesehen werden.
- Von der momentanen *Boost Library* sind 10 Komponenten für den kommenden ISO-Standard für C++ vorgesehen, darunter auch die Bibliothek für Threads.
- Unter Debian: Paket *libboost-dev* installieren. Unter MacOS X: Pakete *gcc45* und *boost* über MacPorts (`http://www.macports.org/`) installieren. Der Sun Studio Compiler kommt leider mit der Thread-Bibliothek von Boost nicht zurecht.
- `http://www.boost.org/`

Übersetzen mit der Boost-Thread-Library

- Unter Linux:
  - *LDLIBS := -lboost_thread*
- Unter MacOS:
  - */opt/local/bin* muss in den *PATH* aufgenommen werden
  - *CXX := g++-mp-4.5*
  - *LDFLAGS := -L/opt/local/lib*
  - *LDLIBS := -lboost_thread-mt*

fork-and-join.cpp

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

   private:
      const int id;
};
```

- Threads werden durch beliebige Funktionsobjekte repräsentiert, die ohne Parameter aufrufbar sind, d.h. sie müssen den ()-Operator zur Verfügung stellen mit **void** als Rückgabetyp.
- Parameter können durch private Variablen repräsentiert werden (im Beispiel *id*), die durch einen Konstruktor initialisiert werden.
- 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 <boost/thread.hpp>

using namespace std;

// class Thread...

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

- Objekte des Typs *boost*::*thread* können mit einem Funktionsobjekt initialisiert werden. Die Threads werden sofort aktiv.
- Mit der *join*-Methode wird auf die Beendigung des jeweiligen Threads gewartet.

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

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

$$\begin{aligned}
P &= M \parallel P_1 \parallel \cdots \parallel P_n \\
M &= (fork_1 \rightarrow \cdots \rightarrow fork_n \rightarrow join_1 \rightarrow \cdots \rightarrow join_n \rightarrow STOP) \\
P_i &= (fork_i \rightarrow join_i \rightarrow STOP)
\end{aligned}$$

fork-and-join2.cpp

```
// fork off some threads
boost::thread threads[10];
for (int i = 0; i < 10; ++i) {
   threads[i] = boost::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 Threads aus der Boost-Bibliothek einander zugewiesen werden, dann wird ein Thread nicht dupliziert, sondern die Referenz auf einen Thread wandern 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.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;

   boost::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] = boost::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.