High Performance Computing: Teil 1
SS 2013

Andreas F. Borchert
Universität Ulm

16. April 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.

Aufnahme von Steve Jurvetson from Menlo Park, USA, CC-AT-2.0, Wikimedia Commons

- 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.

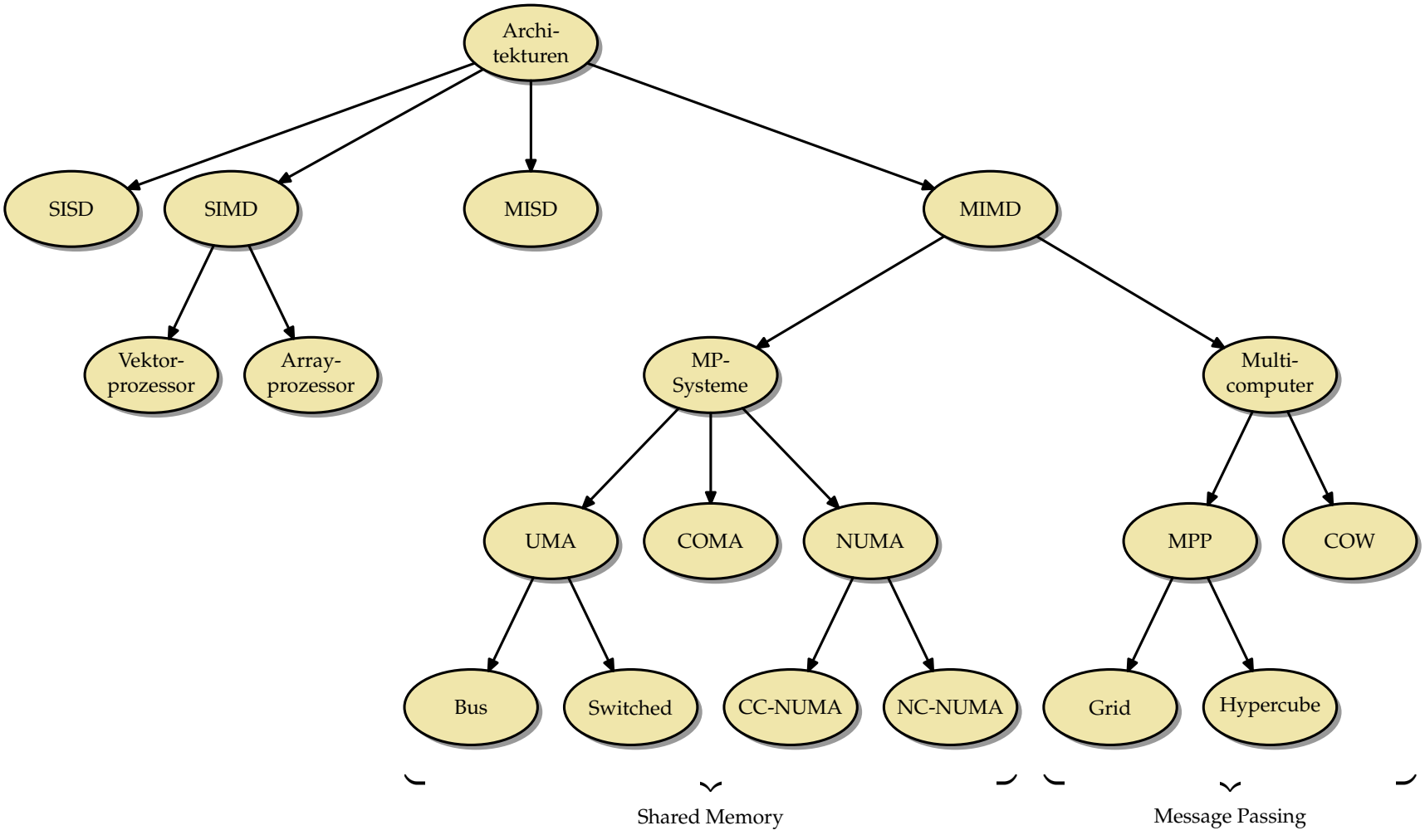
Hier gibt es zwei Varianten:

- 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 „-O3“ 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.