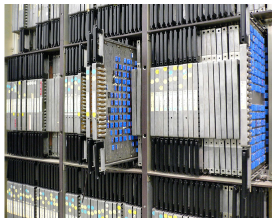


Rechenintensive parallele Anwendungen können nicht sinnvoll ohne Kenntnis der zugrundeliegenden Architektur erstellt werden.

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 sind.

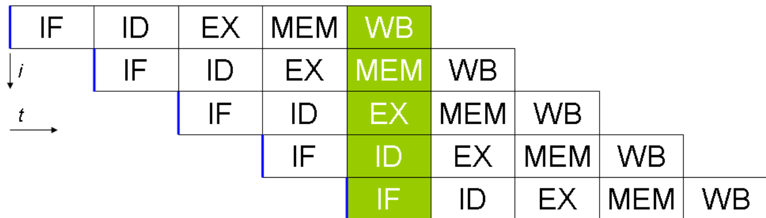
Kosten der Parallelisierung

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

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

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

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



Eine einfache Aufteilung sieht folgende einzelne Schritte vor:

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

Das Diagramm wurde von Stéphane Tsacas geklaut.

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

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

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

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

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

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

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

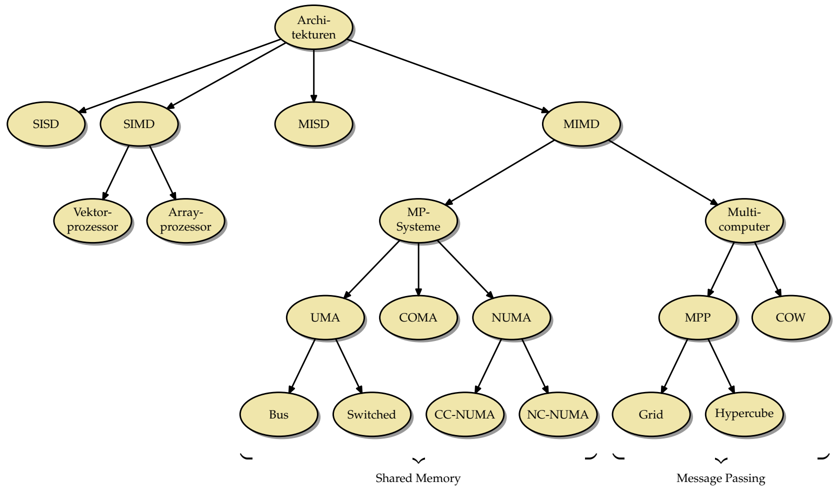
Die MMX- und SSE-Instruktionen des Pentium 4 setzen ebenfalls dieses Modell um.

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

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

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

Architekturen nach Tanenbaum



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

- Die 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.“

- 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.

- 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.

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

- Seit dem aktuellen C++-Standard ISO 14882-2012 (C++11) werden POSIX-Threads direkt unterstützt, wobei die POSIX-Schnittstelle in eine für C++ geeignete Weise verpackt ist.
- 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=c++11“ bzw. „-std=c++14“ verwendet werden.

- Die ausführende Komponente eines Threads wird in C++ immer durch ein sogenanntes Funktionsobjekt repräsentiert.
- In C++ sind alle Objekte Funktionsobjekte, die den parameterlosen Funktionsoperator unterstützen.
- Das könnte im einfachsten Falle eine ganz normale parameterlose Funktion sein:

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

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

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

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

fork-and-join.cpp

```
class Thread {
   public:
      Thread(int id) : id(id) {};
      Thread(const Thread& other) : id(other.id) {};
      void operator()() {
         std::cout << "thread " << id << " is operating" << std::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*();

fork-and-join.cpp

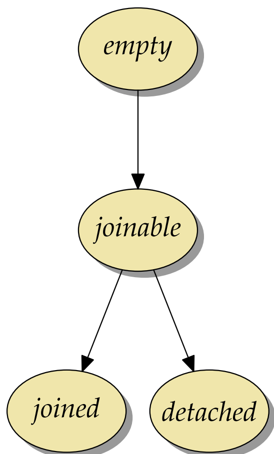
```
#include <iostream>
#include <thread>

// class Thread...

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

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

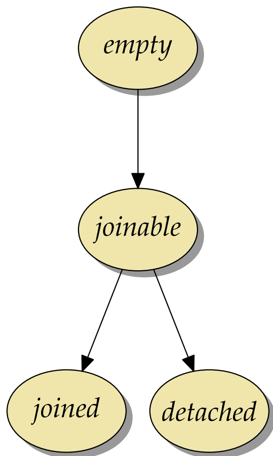
$$P_i = (\mathit{fork} \rightarrow \mathit{join} \rightarrow \mathit{SKIP})$$

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

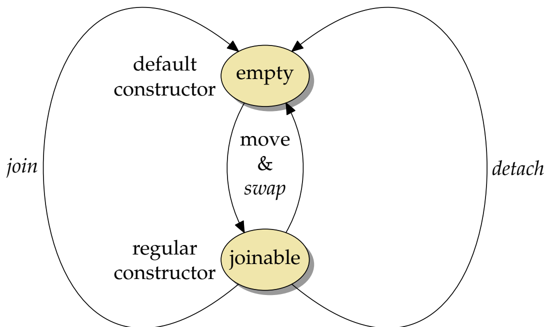
$$\begin{aligned}
P &= M \parallel P_1 \parallel \ldots \parallel P_n \\
M &= (\mathit{fork}_1 \rightarrow \cdots \rightarrow \mathit{fork}_n \rightarrow \mathit{join}_1 \rightarrow \cdots \rightarrow \mathit{join}_n \rightarrow \mathit{SKIP}) \\
P_i &= (\mathit{fork}_i \rightarrow \mathit{join}_i \rightarrow \mathit{SKIP})
\end{aligned}$$



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



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



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

destructing-joinable-thread.cpp

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

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

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

assigning-joinable-thread.cpp

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

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

std::thread im *detached state*

detached-thread.cpp

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

class Thread {
   public:
      Thread(int id) : id(id) {};
      Thread(const Thread& other) : id(other.id) {};
      void operator()() {
         std::cout << "thread " << id << " starts" << std::endl;
         std::this_thread::sleep_for(std::chrono::seconds(2));
         std::cout << "thread " << id << " ends" << std::endl;
      }
   private:
      const int id;
};

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

std::thread im *detached state*

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

fork-and-join2.cpp

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

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

fork-and-join2.cpp

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

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

simpson.cpp

```
double simpson(double (*f)(double), double a, double b, 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 $x_k = a + k \cdot h$.

Verpackung in ein Funktionsobjekt

simpson.cpp

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

   std::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.

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] = std::thread(SimpsonThread(f,
      xleft, x, intervals, results[i]));
}
```

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

simpson.cpp

```
double sum = 0;
for (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

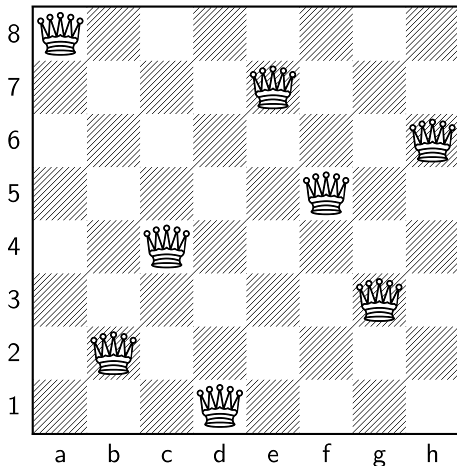
```
cmdname = *argv++; --argc;
if (argc > 0) {
   std::istringstream arg(*argv++); --argc;
   if (!(arg >> N) || N <= 0) usage();
}
if (argc > 0) {
   std::istringstream arg(*argv++); --argc;
   if (!(arg >> nofthreads) || nofthreads <= 0) usage();
}
if (argc > 0) usage();
```

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

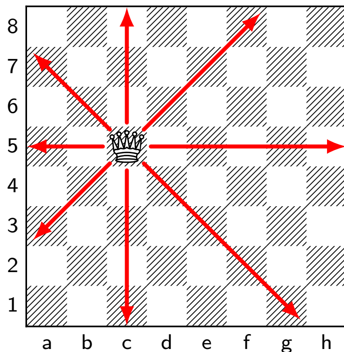
simpson.cpp

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

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



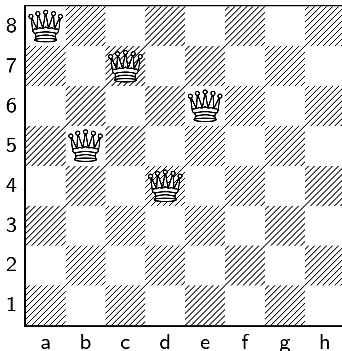
- Problem: Finde alle Stellungen für n Damen auf einem Schachbrett der Größe $n \times n$, so dass sie sich nicht gegenseitig bedrohen.



- Eine Dame im Schachspiel bedroht alle Felder
  - in der gleichen Zeile,
  - in der gleichen Spalte und
  - den beiden Diagonalen.

Lösungsansatz:

- Schritt für Schritt eine Lösung aufbauen, indem eine Dame nach der anderen auf das Brett gestellt wird.
- Wenn es keinen zulässigen Platz für die k-te Dame gibt, wird die $(k-1)$-te Dame vom Brett zurückgenommen und eine andere (noch nicht vorher probierte) Position verwendet.
- Wenn eine Lösung gefunden wird, wird sie notiert und die weiteren Möglichkeiten durchprobiert. (Wenn nur eine Lösung relevant ist, kann das Verfahren dann auch abgebrochen werden.)
- Der Verfahren endet, sobald alle Möglichkeiten durchprobiert worden sind.



- Verfahren, die schrittweise Möglichkeiten durchprobieren und im Falle von „Sackgassen“ zuvor gemachte Schritte wieder zurücknehmen, um neue Varianten zu probieren, nennen sich Backtracking-Verfahren.
- Eine Sackgasse beim 8-Damen-Problem zeigt das obige Diagramm, da es nicht möglich ist, eine weitere Dame unterzubringen.

- Prinzipiell sind alle Backtracking-Verfahren zur Parallelisierung geeignet.
- Im einfachsten Falle wird für jeden möglichen Schritt ein neuer Thread erzeugt, der dann den Schritt umsetzt und von dort aus nach weiteren Schritten sucht.
- Ein explizites Zurücknehmen von Zügen ist nicht notwendig, da jeder Thread mit einer privaten Kopie des Schachbretts arbeitet.
- Sobald eine Lösung gefunden wird, ist diese synchronisiert auszugeben bzw. in einer Datenstruktur zu vermerken.
- Jeder Thread ist dafür verantwortlich, die von ihm erzeugten Threads wieder mit *join* einzusammeln.

queens.cpp

```
class Thread {
   public:
      // ...
   private:
      unsigned int row; /* current row */
      /* a queen on (row, col) threatens a row, a column,
         and 2 diagonals;
         rows and columns are characterized by their number (0..n-1),
         the diagonals by row-col+n-1 and row+col,
         (n is a shorthand for the square size of the board)
      */
      unsigned int rows, cols; // bitmaps of [0..n-1]
      unsigned int diags1; // bitmap of [0..2*(n-1)] for row-col+n-1
      unsigned int diags2; // bitmap of [0..2*(n-1)] for row+col
      std::list<std::thread> threads; // list of forked-off threads
      std::list<unsigned int> positions; // columns of the queens
};
```

queens.cpp

```
constexpr bool in_set(unsigned int member, unsigned int set) {
   return (1<<member) & set;
}
constexpr unsigned int include(unsigned int set, unsigned int member) {
   return set | (1<<member);
}
constexpr unsigned int N = 8; /* side length of chess board */

class Thread {
   public:
      // ...
   private:
      // ...
      unsigned int rows, cols; // bitmaps of [0..n-1]
      unsigned int diags1; // bitmap of [0..2*(n-1)] for row-col+n-1
      unsigned int diags2; // bitmap of [0..2*(n-1)] for row+col
      // ...
};
```

- Die von bereits besetzten Damen bedrohten Zeilen, Spalten und Diagonalen lassen sich am einfachsten durch Bitsets repräsentieren.

Konstruktoren eines Threads

queens.cpp

```
Thread() : row(0), rows(0), cols(0), diags1(0), diags2(0) {
}
Thread(const Thread& other, unsigned int r, unsigned int c) :
      row(r+1), rows(other.rows), cols(other.cols),
      diags1(other.diags1), diags2(other.diags2),
      positions(other.positions) {
   positions.push_back(c);
   rows = include(rows, r); cols = include(cols, c);
   diags1 = include(diags1, r - c + N - 1);
   diags2 = include(diags2, r + c);
}
```

- Der erste Konstruktor dient der Initialisierung des ersten Threads, der von einem leeren Brett ausgeht.
- Der zweite Konstruktor übernimmt den Zustand eines vorhandenen Bretts und fügt noch eine Dame auf die übergebene Position hinzu, die sich dort konfliktfrei hinsetzen lässt.

queens.cpp

```
std::list<std::thread> threads;
```

- Objekte des Typs *std*::*thread* können in Container aufgenommen werden.
- Dabei ist aber die Verlagerungssemantik zu beachten, d.h. der Thread wandert in den Container und wenn er dort bleiben soll, sollte danach mit Referenzen gearbeitet werden.
- Diese Liste dient dazu, alle in einem Rekursionsschritt erzeugten Threads aufzunehmen, damit sie anschließend allesamt wieder mit *join* eingesammelt werden können.

Rekursionsschritt als Funktionsoperator

queens.cpp

```
void operator()() {
   if (row == N) { print_board();
   } else {
      for (unsigned int col = 0; col < N; ++col) {
         if (in_set(row, rows)) continue;
         if (in_set(col, cols)) continue;
         if (in_set(row - col + N - 1, diags1)) continue;
         if (in_set(row + col, diags2)) continue;
         // create new thread with a queen set at (row,col)
         threads.push_back(std::thread(Thread(*this, row, col)));
      }
      for (auto& t: threads) t.join();
   }
}
```

- Wenn alle acht Zeilen besetzt sind, wurde eine Lösung gefunden, die dann nur noch ausgegeben werden muss.
- Ansonsten werden in der aktuellen Zeile sämtliche Spalten durchprobiert, ob sich dort konfliktfrei eine Dame setzen lässt. Falls ja, wird ein neuer Thread erzeugt.

queens.cpp

```
    for (auto& t: threads) t.join();
```

- Beim Zugriff auf Threads im Container sollte das versehentliche Kopieren vermieden werden, das zu einer Verlagerung führen würde.
- In solchen Fällen ist es ggf. sinnvoll, mit Referenzen zu arbeiten.
- Bei **auto** übernimmt der C++-Übersetzer die Bestimmung des korrekten Typs. Die Angabe von „&" ist hier aber noch notwendig, um die Verlagerung der Threads aus dem Container zu vermeiden.