4. Parallelprogrammierung

AlDaBi Praktikum

Inhalt

- Einführung in Parallelität
- OpenMP

- Bemerkungen zur P-Aufgabe

Einführung in Parallelität

Folien z.T. aus VL „Programmierung von Hardwarebeschleunigern“
von Alexander Reinefeld und Thomas Steinke, WS09

David Weese
© 2010/11

Sascha Meiers
WS 2013/14

Voraussetzungen

- Es werden Mechanismen benötigt, um
  - Parallelität zu erzeugen
  - Prozesse voneinander unterscheiden zu können
  - zwischen Prozessen kommunizieren zu können

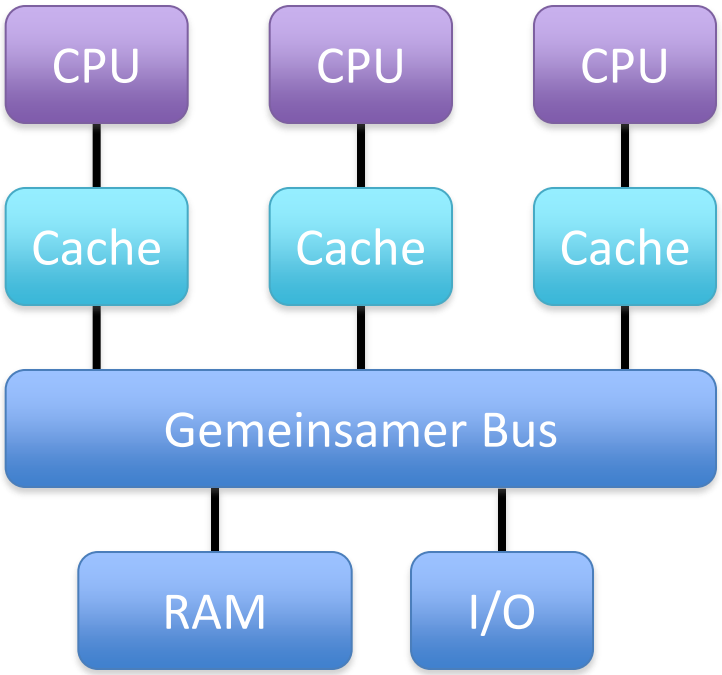
- Für die Formulierung paralleler Programme:
  - Datenparallele Programmiermodelle
    - HPF
  - Kontrollparallele Programmiermodelle
    - MPI, OpenMP, CUDA
  - Beides
    - VHDL, MitrionC

Parallelprogrammierung durch ...

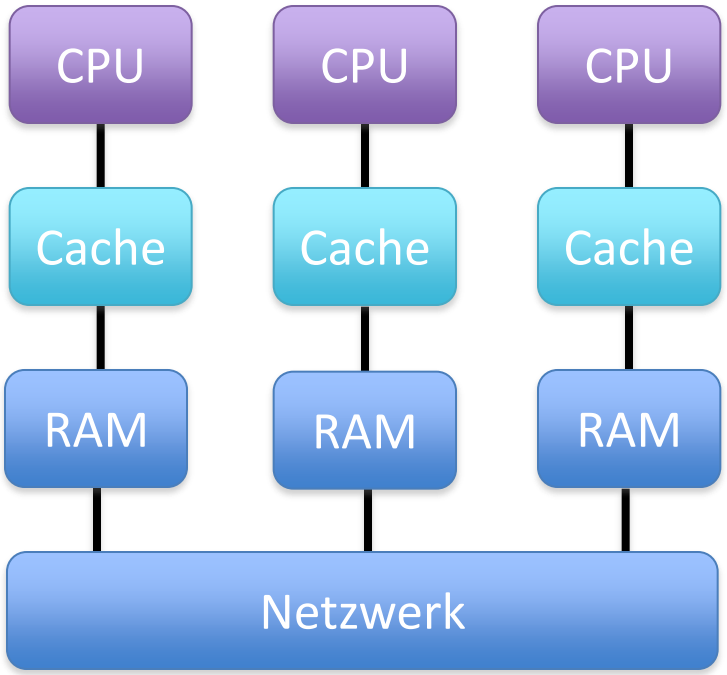
- Direkten Zugriff auf das Netzwerk eines Clusters
  - Effizient, aber nicht portabel
- Unterprogrammbibliotheken
  - Beispiele: PThreads, fork(), MPI, ...
- Erweiterung einer Programmiersprache durch Compiler-Direktiven
  - Beispiel: OpenMP
- Eigenständige parallele Programmiersprache
  - Großer Aufwand: Einarbeitung, neue Compiler, etc.
  - Beispiele: OCCAM, parallele Fortran-Varianten (HPF), MitrionC, ...

Parallele Plattformen

Gemeinsamer Speicher
(shared memory)



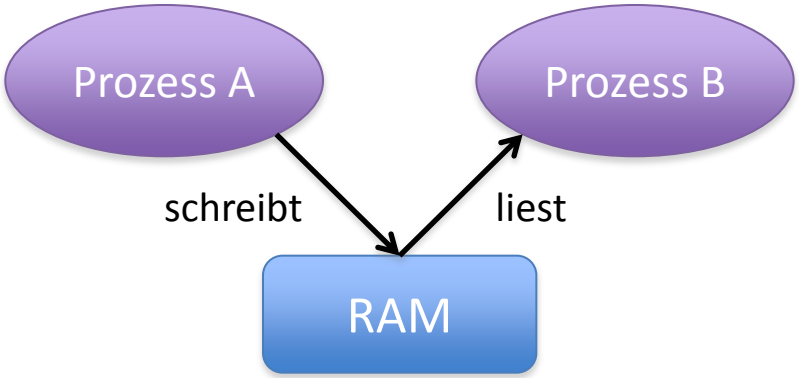
Verteilter Speicher
(distributed memory)



Prozesskommunikation …

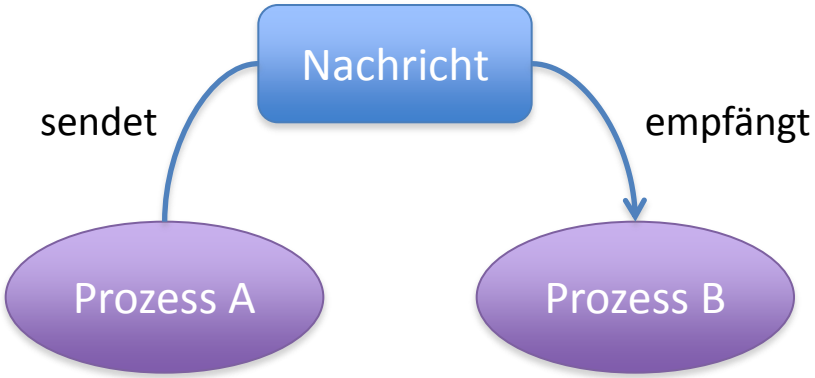
Gemeinsamer Speicher
(shared memory)

- … über gemeinsamen Speicherbereich
  - Schreiben/Lesen im RAM
  - Synchronisation über gegenseitigen Ausschluss



Verteilter Speicher
(distributed memory)

- … über Nachrichtenaustausch (message passing)
  - Senden/Empfangen von Nachrichten
  - Nur lokale (private) Variablen



Begriffe

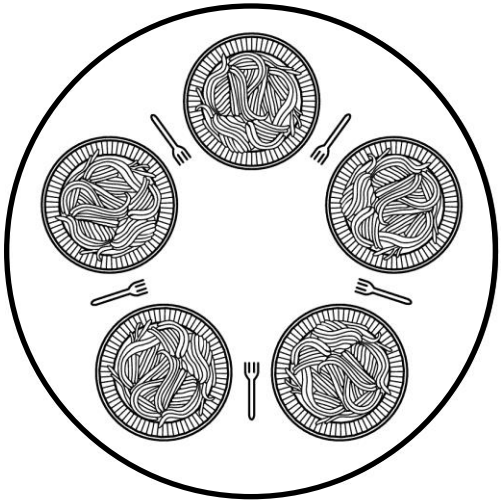
- Prozess
  - Besteht aus eigenem Adressraum
    - Für Stack, Heap, Programmcode, ...
    - Ist dadurch vor anderen Prozessen geschützt
  - Kann Resourcen reservieren
  - Hat einen oder mehrere Threads, die den Code ausführen
  - Beispiel:
    - Programm
- Thread
  - Gehören zu einem Prozess
  - Bestehen aus eigenem Stack und CPU-Registerzustand
  - Haben den Adressraum des zugehörigen Prozesses
    - Threads desselben Prozesses sind nicht voreinander geschützt
  - Beispiel:
    - Auf mehrere Threads verteilte for-Schleife eines Programms

Fallstricke

- Race Conditions
  - Situationen, in denen das „Wettrennen“ (race) der Prozesse beim Zugriff auf gemeinsame Ressourcen Auswirkungen auf das Ergebnis eines Programmlaufs hat
  - Beispiel: 2 Threads schreiben in dieselbe Speicherstelle
- Lösung
  - Synchronisation
    - **Mutex** (Mutual Exclusion)
      - kann von mehreren Threads verlangt werden (lock), aber nur einer besitzt sie bis er sie freigibt (unlock)
    - **Semaphore**
      - kann von mehreren Threads verlangt werden, ist aber immer nur in Besitz von höchstens k Threads
      - Mutex ist Semaphore mit k=1
    - **Barrier**
      - Eine Gruppe von Threads hält solange an einer Barriere an, bis alle angekommen sind, danach laufen alle weiter

Fallstricke (II)

- Bei unpassender Synchronisation entstehen:
  - Verklemmung (dead locks)
    - Threads warten gegenseitig auf die Resourcen der anderen
  - Aushungern (starvation)
    - Resource wechselt (unfairerweise) nur innerhalb einer Gruppe von Threads
    - Threads außerhalb der Gruppe erhalten die Resource nie



- Beispiel:
  - Philosophenproblem
    - Philosophen essen und denken
    - Zum Essen braucht jeder 2 Gabeln
    - Jeder kann gleichzeitig nur eine Gabel aufheben
  - Verklemmung
    - Jeder nimmt die linke Gabel auf und wartet auf die rechte
  - Lösung
    - Eine Mutex für den ganzen Tisch zum Prüfen und Aufnehmen zweier Gabeln

OPEN MULTI-PROCESSING

Folien z.T. aus VL „Parallele Programmierung“ von Nicolas Maillard und Marcus Ritt, TU Berlin

David Weese
© 2010/11

Sascha Meiers
WS 2013/14

Verfügbarkeit

Compiler	OpenMP Unterstützung	Compiler Schalter
g++	GCC 4.2: OpenMP 2.5 GCC 4.4: OpenMP 3.0 *(Pool-Recher: GCC 4.7.2)*	**`-fopenmp`**
Visual C++	VS 2005-2010: OpenMP 2.0	**`/openmp`** In der IDE*
Intel C++	V9: OpenMP 2.5 V11: OpenMP 3.0	Windows: **`/Qopenmp`** Linux: **`-openmp`**
Sun Studio	V12: OpenMP 2.5 V12 U1: OpenMP 3.0	**`-xopenmp`**

- OpenMP aktivieren in Visual Studio 2005 oder später:
  - Öffne Properties des Projektes und gehe zu:
    `Configuration Properties -> C/C++ -> Language`
  - Setze `OpenMP Support` auf `yes`
- Achtung:
  - Visual Studio 2005 Express Edition unterstüzt kein OpenMP, obwohl der Schalter existiert

Einführung

- OpenMP stellt eine Programmierschnittstelle (API) für C/C++ (und Fortran) zur Verfügung
  - Erweiterung der Programmiersprache durch #pragma-Direktiven
    - Eine Direktive besteht aus einem Namen und einer Klauselliste:
      #pragma omp directive [clause list]
  - Bibliotheksfunktionen
    - #include <omp.h>

- Nur für Plattformen mit gemeinsamen Speicher (shared memory)

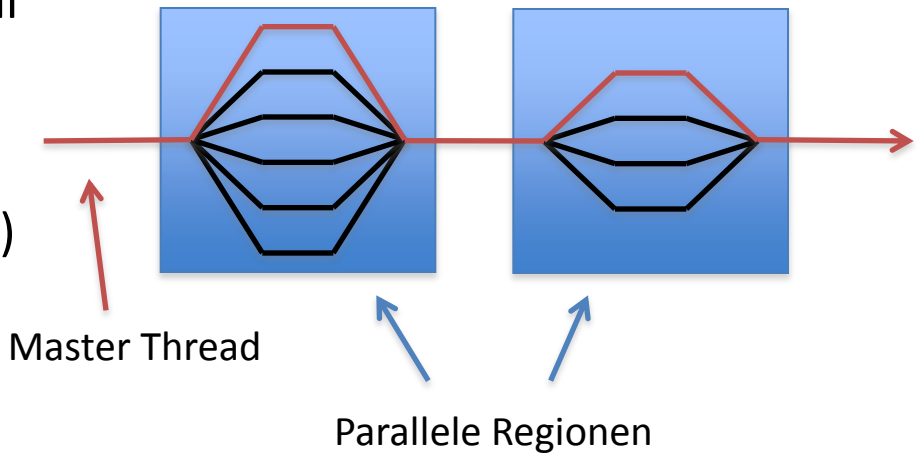
- Nützliche Links
  - OpenMP Homepage - http://www.openmp.org/
  - OpenMP Tutorial - http://computing.llnl.gov/tutorials/openMP/
  - Kurzübersicht - http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf

Programmiermodell

- Parallelität lässt sich schrittweise erzeugen
  - Ein serielles Programm wird zu einem parallelen
  - Serielle Version weiterhin nutzbar

- Direktive parallel startet ein Team
  - Aktueller Thread ist Master
  - Alle Threads bearbeiten den folgenden Abschnitt
- Direktive for teilt Iterationen auf Team-Mitglieder auf
  - Jeder Thread bearbeitet einen Teil

- Ende des parallelen Blocks
  - Implizite Synchronisation (barrier)
  - Nur Master läuft weiter



OpenMP Beispiel

- Einfaches Beispielprogramm in OpenMP:

```
#include <cstdio>
#include <omp.h>

int main(int, char*[])
{
    #pragma omp parallel
    printf("Hello, world.\n");

    return 0;
}
```

- Ausgabe auf Intel Core 2 Duo:

```
Hello, world.
Hello, world.
```

OpenMP Beispiel (II)

- **Unterscheidung** der Threads:

```
#include <cstdio>
#include <omp.h>

int main(int, char*[])
{
    #pragma omp parallel
    printf("I'm thread %i of %i.\n",
        omp_get_thread_num(),
        omp_get_num_threads());

    return 0;
}
```

- Ausgabe:

```
I'm thread 0 of 2.
I'm thread 1 of 2.
```

Von seriell zu parallel

- Serielles Programm:

```
double A[10000];
for (int i = 0; i < 10000; ++i)
    A[i] = langwierige_berechnung(i);
```

- Parallelisiert mit OpenMP:

```
double A[10000];
#pragma omp parallel for
for (int i = 0; i < 10000; ++i)
    A[i] = langwierige_berechnung(i);
```

- Serielle Version weiterhin nutzbar
  - `#pragma omp` wird ignoriert von Compilern ohne OpenMP-Unterstützung

Direktiven

- Parallele Regionen
  - `omp parallel`
- Klauseln zur Definition gemeinsamer und privater Daten
  - `omp shared, private, …`
- Aufteilung von Schleifen
  - `omp for`
- Synchronisation
  - `omp atomic, critical, barrier, …`
- Bibliotheksfunktionen und Umgebungsvariablen
  - `omp_set_num_threads(), omp_set_lock(), …`
  - `OMP_SCHEDULE, OMP_NUM_THREADS, …`

parallel

- Parallele Ausführung mit **parallel**
  - Folgender Block wird von allen Threads parallel ausgeführt
  - Programmierer verteilt die Arbeit
    - Entweder manuell
    - Oder mit weiteren Direktiven, bspw. for, sections, ...

```
double A[10000];
int cnt = omp_get_num_threads();
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int i_start = 10000 *  id    / cnt;
    int i_end   = 10000 * (id+1) / cnt;
    langwierige_berechnung(A, i_start, i_end);
}
```

for

- Aufteilung von Schleifeniteration mit **for**
  - OpenMP teilt Iterationen den einzelnen Threads zu
  - Nur forSchleifen mit bestimmter Syntax

```
double A[elemente];
#pragma omp parallel
#pragma omp for
for (int i = 0; i < 10000; ++i)
    A[i] = langwierige_berechnung(i);
```

- Seit OpenMP 3.0 sind auch Iterator-Schleifen parallelisierbar:

```
vector<int> v(10000);
typedef vector<int>::iterator iter;
#pragma omp parallel
#pragma omp for
for (iter i = v.begin(); i < v.end(); ++i)
    langwierige_berechnung(*i);
```

for (II)

- Einschränkungen
  - Nur ganzzahlige Schleifenvariablen (mit oder ohne Vorzeichen)
  - Test nur mit <, <=, >, >=
  - Schleifenvariable verändern: nur einfache Ausdrücke
    - Operatoren ++, , +, +=, =
  - Obere und untere Grenze unabhängig von Schleifendurchlauf
    - Ausdrücke sind möglich

- Automatische Synchronisation nach **for** mit Barriere
  - kann mit **nowait** unterdrückt werden

    - Kurzform:
  - `#pragma omp parallel for`

for (III)

- Verschiedene Arten, den Indexraum auf Threads zu verteilen
- Auswahl erfolgt mit **schedule**
  - schedule(static[,k])
    - Indexraum wird in Blöcke der Größe k zerlegt und den Threads **reihum** zugewiesen
    - k=1: 012012012.....0
    - k=5: 00000111112222200000111112222 2...00000
  - schedule(dynamic[,k])
    - Indexraum wird in Blöcke der Größe k zerlegt und den Threads **nach Bedarf** zugewiesen
  - schedule(guided[,k])
    - Indexraum wird in Blöcke **proportional zur Restarbeit** auf Threads aufgeteilt und nach Bedarf zugewiesen; k = minimale Iterationszahl
  - schedule(auto)
    - Implementierungspezifisch (Standard, wenn keine Angabe)
  - schedule(runtime)
    - Entscheidung zur Laufzeit (omp_set_schedule, omp_get_schedule, OMP_SCHEDULE)

Klauseln

- Festlegen der Anzahl der Threads:

```
double A[10000];
#pragma omp parallel for num_threads(4)
for (int i = 0; i < 10000; ++i)
    A[i] = langwierige_berechnung(i);
```

  - Alternativ ginge auch `omp_set_num_threads(4);`
  - Oder über Umgebungsvariable `export OMP_NUM_THREADS=4`

- Bedingte Parallelisierung:

```
double A[elemente];
#pragma omp parallel for if (elemente > 100)
for (int i = 0; i < elemente; ++i)
    A[i] = langwierige_berechnung(i);
```

Mehrdimensionale Schleifen

- Zusammenfassung von Schleifen
  - forDirektive wirkt nur auf nächste Schleife
  - **collapse(k)** kombiniert Indexraum der folgenden k Schleifen

- Beispiel: Matrixmultiplikation

```
#pragma omp parallel for collapse(3)
for (int i = 0; i < dim1; ++i)
    for (int j = 0; j < dim2; ++j)
        for (int k = 0; k < dim3; ++k)
            C[i][j] = A[i][k]*B[k][j];
```

sections

- Parallelisierung ohne Schleifen mit Abschnitten
  - Jedem Abschnitt wird ein Thread zugewiesen
  - Nur statische Parallelität möglich: Abschnitt gilt für ganzes Team

```
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
    arbeit1();
    #pragma omp section
    arbeit2();
    #pragma omp section
    arbeit3();
}
```

sections (II)

- Rekursionen sind mit **sections** sehr einfach parallelisierbar
  - Beispiel: Quicksort mit verschachtelten Teams:

```
void quicksort(int a[], int l, int r)
{
    if (l < r)
    {
        int i = partition(a, l, r);
        #pragma omp parallel sections
        {
            #pragma omp section
            quicksort(a, l, i - 1);
            #pragma omp section
            quicksort(a, i + 1, r);
        }
    }
}
```

task

- Parallelisierung ohne Schleifen mit Tasks (OpenMP 3.0)
  - Jeder Task wird (reihum) einem Thread zugewiesen
  - Unterschied zu sections
    - Definition eines Tasks an beliebiger Stelle von beliebigem Thread im Team
    - Ein Thread im Team führt den Task aus

```
void quicksort(int a[], int l, int r)
{
  if (l < r)
  {
    int i = partition(a, l, r);
    #pragma omp task
    quicksort(a, l, i - 1);
    quicksort(a, i + 1, r);
  }
}
```

```
int main()
{
  #pragma omp parallel
  {
    #pragma omp single
    quicksort(a, 1, 99);
  }
}
```

Synchronisation

- Block der nur von einem / dem Master Thread im Team ausgeführt werden soll
  - #pragma omp **single** / #pragma omp **master**
  - Beispiel: Datei einlesen, Textausgabe auf der Konsole

- Block der von jedem Thread einzeln (seriell) ausgeführt werden soll
  - #pragma omp **critical**
  - Analog zur Mutex, entspricht dem Bereich zwischen lock und unlock
  - Beispiel: Zugriff auf gemeinsame Datenstruktur / Resource

- Atomares Schreiben (“kleiner kritischer Abschnitt”)
  - #pragma omp **atomic**
  - Beispiel: Inkrementieren eines globalen Zählers

- Synchronisation aller Threads mit Barriere
  - #pragma omp **barrier**

Speicherklauseln für Variablen

- **shared**
  - Daten sind für alle Threads sichtbar/änderbar. Standard in Schleifen.
- **private**
  - Jeder Thread hat eigene Kopie. Daten werden nicht initialisiert. Sind außerhalb des parallelen Abschnitts nicht bekannt.
- **firstprivate**
  - private Daten. Initialisierung mit letztem Wert vor parallelem Abschnitt
- **lastprivate**
  - private Daten. Der Thread, der die letzte Iteration ausführt, übergibt den Wert aus dem parallelen Abschnitt an das Hauptprogramm.
- **threadprivate**
  - globale Daten, die im parallelen Programmabschnitt jedoch als privat behandelt werden. Der globale Wert wird über den parallelen Abschnitt hinweg bewahrt.
- **reduction**
  - private Daten, werden am Ende auf einen globalen Wert zusammengefasst.

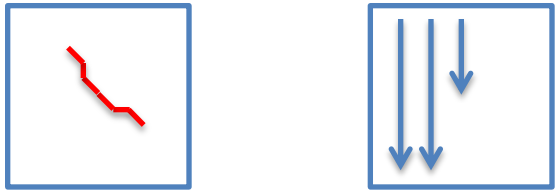
BEMERKUNGEN ZUR P-AUFGABE

David Weese
© 2010/11

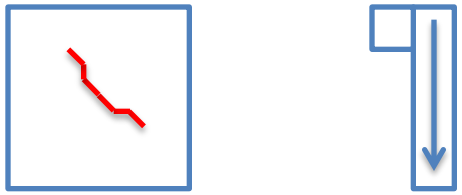
Sascha Meiers
WS 2013/14

Bemerkungen zu Aufgabe 3

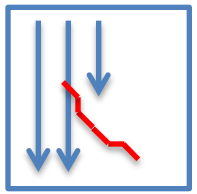
- Veschiedene richtige Ansätze:
  - Tracebackmatrix + Scorematrix: *14 Gruppen*



  - Tracebackmatrix + Score-Spalte: *0 Gruppen*



  - Nur die Scorematrix: *3 Gruppen*



- Falsche Ansätze:
  - Mehr als 2 Matrizen
  - vector<vector<vector<int> > >

Bemerkungen zu Aufgabe 3 (II)

- Dynamische Speicherallokation:
  - Nicht erlaubt (nach ISO-Standard):

```
string t1 = argv[1];
string t2 = argv[2];
unsigned m = t1.length() +1;
unsigned n = t2.length() +1;
int matrix [m][n];
```

g++ warning: ISO C++ forbids variable length array ‘matrix’

  - Besser: vector

```
string t1 = argv[1];
string t2 = argv[2];
unsigned m = t1.length() +1;
unsigned n = t2.length() +1;
vector<vector<int> > matrix(m,vector<int>(n));
```

(oder durch resize vergrößern)

Bemerkungen zu Aufgabe 4

- Suche im Suffixarray
  - Suche erweitern auf beide Grenzen R_P und L_P
  - Diesmal kommt es auf Effizienz an
    - mlr-Heuristik
    - substr() bzw. Copy-Operationen vermeiden
- Einlesen von Textdateien
  - Datenbank einlesen und Zeilen konkatenieren (ohne \n)
  - Queries einlesen und jede Zeile als neuen String interpretieren
  - File handling in C++: http://www.cplusplus.com/doc/tutorial/files/
- Einlesen des Suffixarrays
  - Binärdatei:
    - Kann ohne Wissen über den Inhalt nicht interpretiert werden
    - In unserem Fall: Folge von 4 byte unsiged ints
    - Einlesefunktion vorgegeben, so wird sie benutzt:

```
vector<unsigned> sa;
if(! readBinarySA(sa, argv[1]))
    return 1;
```

Bemerkungen zu Aufgabe 4 (II)

- Test-Daten
  - Modifiziertes Chromosom 1
    - Länge: 225 Mbp (Mega base pairs)
  - Suffixarray dazu hat ca. 850 MB
  - 300 Wörter die im Text vorkommen

- Aufgabenstellung
  - Suche von **exakten** Matches
  - Erwartete* Laufzeit pro Suchwort: $O(\log n + m)$
    - Online Algorithmen: $O(n + m)$
  - Suche von mehreren Wörtern parallel
    - Beschleunigung um bis zu `num_threads`

- In Aufgabe 5: Suche von approximativen Matches

* Worst-case Laufzeit $O(m \log n)$

Bemerkungen zu Aufgabe 5

- Aufgabe 5 ist die Weihnachtsaufgabe
  - Ausgabe: 16.12.2013
  - Abgabe: 20.01.2014

  } 5 Wochen

- Thema: QUASAR
  - Deutlich mehr Aufwand als bisherige Aufgaben

- Kein Tutorium dazu in diesem Jahr
  - Besuch der P-VL am 19.12. zu empfehlen
  - Bei Bedarf Zusatztutorium am 9.01.
    - Bitte Mail an mich falls gewünscht