



## Parallelrechnen

Johannes Hötzer | 19-02-2015 | KIT/HS/DHBW

Scaling the large way

News

News |

### ► Vorlesung 02

- <http://www.heise.de/newsticker/meldung/iX-Workshop-zur-parallelen-Programmierung-2552514.html>
- <http://www.heise.de/newsticker/meldung/Lenovo-baut-Supercomputer-mit-64-Bit-ARM-Technik-2552963.html>

Seite 2 | Johannes Hötzer | Parallelrechnen | 19-02-2015

Motivation      Prozess vs. Threads      Threads im Betriebssystem  
○○○○

### Teil I

## Threads and Processes

Seite 3 | Johannes Hötzer | Parallelrechnen | 19-02-2015

Motivation      Prozess vs. Threads      Threads im Betriebssystem  
○○○○

Inhaltsverzeichnis |

Motivation

Prozess vs. Threads

Threads im Betriebssystem

Keine Unterstützung durch Betriebssystem  
1:1 Abbildung  
m:n Abbildung

Abschluss

Seite 4 | Johannes Hötzer | Parallelrechnen | 19-02-2015

Notes

---

---

---

---

---

---

---

---

Notes

---

---

---

---

---

---

---

---

Notes

---

---

---

---

---

---

---

---

Notes

---

---

---

---

---

---

---

---

Motivation I

Was haben Staus mit paralleler Programmierung zu tun?



Motivation

Prozess vs. Threads

Threads im Betriebssystem

Abchluss

Motivation II

- ▶ CPU ist begrenzte Ressource
- ▶ Betriebssystem regelt Zugriff auf Ressourcen
- ▶ Scheduling
- ▶ Bei zu vielen Nutzern kommt es auch hier zu Staus
- ▶ Frage nach Art des Scheduling und der **Ebene** sowie der Kosten

Motivation

Prozess vs. Threads

Threads im Betriebssystem

Abchluss

Prozess vs. Threads

- Prozess:
- ▶ eigener Adressraum (privater Speicher)
  - ▶ Daten müssen selbst übermittelt werden
  - ▶ werden immer vom Betriebssystem verwaltet werden
- Thread:
- ▶ gemeinsamer Adressraum (gemeinsamer Speicher / Heap)
  - ▶ Verwaltung übernimmt Threadbibliothek
  - ▶ können aber vom Betriebssystem unterstützt / verwaltet werden

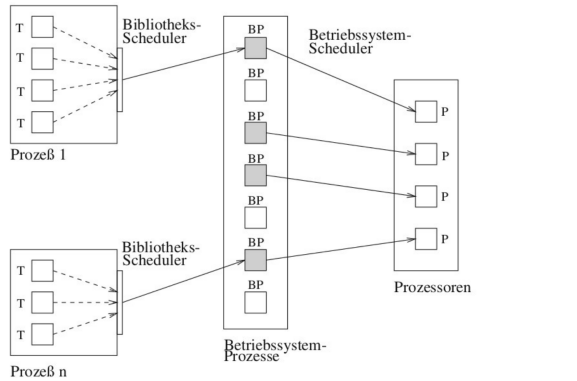
Motivation

Prozess vs. Threads

Threads im Betriebssystem

Abchluss

Keine Unterstützung durch Betriebssystem



Notes

Notes

Notes

Notes

MotivationProzess vs. ThreadsThreads im BetriebssystemAbschluss

1:1 Abbildung

The diagram illustrates a 1:1 mapping between user-space processes and kernel threads. On the left, two boxes represent 'Prozeß 1' and 'Prozeß n'. 'Prozeß 1' contains four threads (T), with two shaded. 'Prozeß n' contains three threads (T), with two shaded. In the center, a box labeled 'Betriebssystem-Threads' contains eight corresponding kernel threads (BT), with four shaded. On the right, a box labeled 'Prozessoren' contains four processors (P). Arrows show each process box connected to its set of kernel threads, and each kernel thread connected to a specific processor. A 'Betriebssystem-Scheduler' box is positioned above the kernel threads, with arrows pointing to them.

Seite 9Johannes Hötzer | Paralleltrechnen | 19-02-2015

Notes

---

---

---

---

---

---

---

---

MotivationProzess vs. ThreadsThreads im BetriebssystemAbschluss

m:n Abbildung

The diagram illustrates an m:n mapping. On the left, two boxes represent 'Prozeß 1' and 'Prozeß n'. 'Prozeß 1' contains four threads (T), with two shaded. 'Prozeß n' contains three threads (T), with two shaded. In the center, a box labeled 'Betriebssystem-Threads' contains six kernel threads (BT), with three shaded. On the right, a box labeled 'Prozessoren' contains four processors (P). Arrows show each process box connected to its set of threads, which then connect to a shared pool of kernel threads. A 'Bibliotheksscheduler' box is positioned between the user threads and the kernel threads, with arrows pointing to the threads. A 'Betriebssystem-Scheduler' box is positioned above the kernel threads, with arrows pointing to them.

Seite 10Johannes Hötzer | Paralleltrechnen | 19-02-2015

Notes

---

---

---

---

---

---

---

---

MotivationProzess vs. ThreadsThreads im BetriebssystemAbschluss

Beispiel: Solaris

The diagram illustrates the Solaris example of thread management. At the top, two boxes represent 'Benutzerprozeß 1' and 'Benutzerprozeß n'. 'Benutzerprozeß 1' contains four user threads (T), with two shaded. 'Benutzerprozeß n' contains four user threads (T), with two shaded. Below each user thread box is a box labeled 'L' (library thread). Arrows show each user thread connected to its corresponding library thread. In the center, a box labeled 'Betriebssystem-Scheduler' contains eight kernel threads (BT), with four shaded. On the right, a box labeled 'Prozessoren' contains four processors (P). Arrows show each library thread connected to its corresponding kernel thread, which then connects to a specific processor. A 'Bibliotheksscheduler' box is positioned above the library threads, with arrows pointing to them.

Seite 11Johannes Hötzer | Paralleltrechnen | 19-02-2015

Notes

---

---

---

---

---

---

---

---

MotivationProzess vs. ThreadsThreads im BetriebssystemAbschluss

Zusammenfassung

- Unterschied Thread und Prozess
- Verwaltung von Threads und Prozessen
- Ebenen der Verwaltung

Notes

---

---

---

---

---

---

---

---

## Teil II

# Fork und Join

### Inhaltsverzeichnis I

Motivation

Einführung

fork

getpid

waitpid

fork-Bomben

Abschluss

### Motivation

Wie erzeuge ich einen neuen Prozess?

### Motivation

Wie erzeuge ich einen neuen Prozess?

> ./myprogramm

- Bash Befehl der einen neue Prozess startet.
- Aber wie programmiert man so etwas?

### Notes

### Notes

### Notes

### Notes

Notes

Notes

Notes

Notes

Motivation Einführung fork getpid waitpid fork-Bomben Abschluss

Beispiel

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 int main () {
6     int pid = fork();
7
8     if (pid == 0) {
9         printf ("Kindprozess: (PID: %d)\n", getpid());
10        sleep (1); //Fork
11        exit (0);
12    } else if (pid > 0) {
13        printf ("Elternprozess: (PID: %d)\n",  getpid());
14        sleep (1); //Fork
15    } else {
16        fprintf (stderr, "Error");
17        exit (1);
18    }
19    return 0;
20 }
21
```

Seite 21 | Johannes Hötzer | Paralleltrechnen | 19-02-2015

Notes

Motivation Einführung fork getpid waitpid fork-Bomben Abschluss

waitpid

Aufgabe

Vaterprozess wartet auf den Kindprozess mit der angegebenen PID

Signatur

```
1 int waitpid(pid_t pid, int *status, int options);
```

<http://linux.die.net/man/2/waitpid>

Seite 22 | Johannes Hötzer | Paralleltrechnen | 19-02-2015

Notes

Motivation Einführung fork getpid waitpid fork-Bomben Abschluss

fork-Bomben

Ezeugung endlos vieler Prozesse

C

```
1 int main(void) {
2     while(1)
3         fork();
4 }
```

Bash

```
1 :O{ :!:& };;
```

Wer kann das übersetzen?

Seite 23 | Johannes Hötzer | Paralleltrechnen | 19-02-2015

Notes

Motivation Einführung fork getpid waitpid fork-Bomben Abschluss

Abschluss

Seite 24 | Johannes Hötzer | Paralleltrechnen | 19-02-2015

Notes

- fork/Join API
- Low-Level API
- Findet auf Systemebene Anwendung

Teil III

OpenMP

Motivation

Einführung

Erzeugen von parallelen Abschnitten

#pragma omp parallel

#pragma omp for

#pragma omp sections

#pragma omp single

#pragma omp master

Steuern paralleler Abschnitte

#pragma omp barrier

#pragma omp critical

#pragma omp atomic

#pragma omp flush

#pragma omp cancel

omp\_set\_num\_threads

omp\_get\_num\_threads

omp\_in\_parallel

omp\_get\_max\_threads

omp\_get\_thread\_num

Locks

Fehlerquellen

Notes

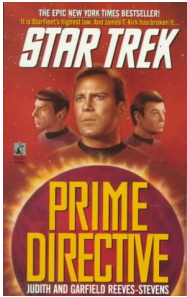
Notes

Notes

Notes

- ▶ <http://openmp.org>
- ▶ <https://computing.llnl.gov/tutorials/openMP/>

Was hat Star Trek mit Parallelisierung zu tun?



In OpenMP dreht sich alles um Directiven

Einführung

- ▶ Open Multi-Processing (OpenMP)
- ▶ Standard für Shared-Memory-Programmierung
- ▶ basierend auf Posix Threads
- ▶ C, C++, Fortran
- ▶ Compiler Direktiven (von nahezu allen unterstützt, GCC, Intel, clang,...)
- ▶ \*inx, Windows
- ▶ eine plattformunabhängige Programmierschnittstelle
- ▶ Parallelität basierend auf dem "Fork-Join-Modell"
- ▶ im HPC Bereich gekoppelt mit MPI
- ▶ mit OpenMP parallelisierte Programme laufen auch ohne Compiler Unterstützung mit Ausnahmen richtig
- ▶ oft zur Parallelisierung von Schleifen verwendet

Notes

Notes

Notes

Notes



Vorteile

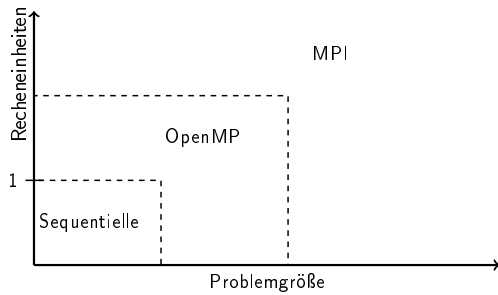
- Einfach zu verwenden
- Parallelisierung kann inkrementell erfolgen
- Keine explizite Kommunikation erforderlich
- Parallele und sequentielle Lösung in einem Programm
- Keine Pflege von zwei Programmversionen erforderlich
- Weit verbreitet
- Auf aktuellen Mehrkernplattformen verwendbar

Nachteile

- Fördert nicht das parallele Denken
- Beschränkt auf Systeme mit wenigen Prozessoren
- Shared Memory nur für wenige Prozessoren umsetzbar
- Skalierbarkeit und Effizienz der parallelen Programme ist beschränkt
- Gleichzeitiger Zugriff auf gemeinsame Variablen führt zu Flaschenhals
- Es bleiben sequentielle Programmteile erhalten (Amdahl's trap)

Warum sollte man OpenMP einsetzen?

- Guter Leistungszuwachs innerhalb kurzer Entwicklungszeit möglich
- Aber: Höherer Entwicklungsaufwand (MPI) führt meist zu besseren Ergebnissen



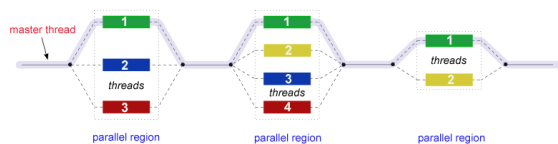
Notes

Notes

Notes

Notes

## Fork-Join Model



Quelle: <https://computing.llnl.gov/tutorials/openMP/>, besucht 09.01.2014

## OpenMP API Komponenten

Die OpenMP API besteht aus drei unterschiedlichen Teilen

- ▶ Compiler Directives
- ▶ Runtime Library Routines
- ▶ Environment Variables

Hallo Welt

```
1 #include <omp.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(int argc, char* argv[]) {
6     #pragma omp parallel
7     printf("Hallo Welt! %d \n", _OPENMP);
8
9     return 0;
10 }
```

## Arbeiten mit einem OpenMP-Programms

## Kompilieren

```
cc -Wall -std=c99 -D _BSD_SOURCE -fopenmp -o myprogramm myprogramm.c
```

## Ausführen

```
OMP_NUM_THREADS=4 ./myopenmpprogramm -a -b -c
```

**Note:** Gültig für Linux mit GCC

Directives

Compiler Erweiterung zur Parallelisierung

```
1 #pragma omp [directive] [clause[ [, ]clause] ...]
```

Environment Routines

Funktionen zur Steuerung und Monitoring der Threads, Prozessoren und der parallelen Umgebung

Notes

Notes

Erzeugen von parallelen Abschnitten

Aufgabe

Startet Team von parallel ausgeführten Threads

Signatur

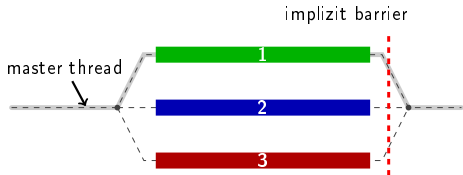
```
1 #pragma omp parallel if(scalar-expression)
2     num_threads(integer-expression)
3     default(shared|none)
4     private(list)
5     firstprivate(list)
6     shared(list)
7     reduction(reduction-identifier:list)
8 {
9     // Parallel: Block
10 }
```

Notes

Notes

Fäden

```
#pragma omp parallel
{
...
}
```



Beispiel

```
1 #include <omp.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(int argc, char *argv[]) {
6     int i = 10;
7     int sum;
8     #pragma omp parallel private(i) reduction(+:sum)
9     {
10         printf("thread %d: i = %d\n", omp_get_thread_num(), i);
11         i = 1000 + omp_get_thread_num();
12         sum = omp_get_thread_num();
13     }
14
15     printf("i = %d, sum = %d\n", i, sum);
16
17     return 0;
18 }
```

Wie sieht die Ausgabe aus?

Wie sieht die Ausgabe aus?

thread 0: i = 234  
thread 3: i = 32717  
thread 1: i = 32717  
thread 2: i = 1  
i = 10

Was steht in der Variable sum bei vier Threads?

6

private(list)

Liste aller Variablen die **private**, also nur für den Thread sichtbar sein sollen

**Note:** Private Variablen sind mit zufälligem Wert belegt

firstprivate(list)

Liste aller Variablen die **private** sein sollen, aber mit dem Wert vor dem Parallele-Block belegt sein sollen

if(scalar-expression)

Gibt an ob Block parallel (ture) ausgeführt wird

num\_threads(integer-expression)

Legt die Anzahl von Threads fest.

default(shared|none)

Gibt an ob Variablen per Default gemeinsam oder privat sind.

**Note:** Per Default sind Variablen gemeinsam

shared(list)

Liste aller Variablen die **gemeinsam** sein sollen

Notes

Notes

Notes

Notes

reduction(reduction-identfier:list)

Liste aller **Operation-Variablen-Paare** die entsprechend der Operation am Ende es Threads mit dem Wert der Variablen reduziert werden sollen

- + :Variable Summation aller Werte (Initialwert 0)
- :Variable Subtraktion aller Werte (Initialwert 0)
- \* :Variable Multiplikation aller Werte (Initialwert 1)
- && :Variable Boolsche UND Verknüpfung aller Werte (Initialwert 1)
- || :Variable Boolsche ODER Verknüpfung aller Werte (Initialwert 0)
- & :Variable Bitweise UND Verknüpfung aller Werte (Initialwert 0)
- | :Variable Bitweise ODER Verknüpfung aller Werte (Initialwert 0)
- ^ :Variable Bitweise XODER Verknüpfung aller Werte (Initialwert 0)

#pragma omp for

Aufgabe

Automatisches zerlegen einer for-Schleife in Teile die parallel ausgeführt werden

Signatur

```
1 #pragma omp [parallel] for
2
3     private(list)
4     firstprivate(list)
5     lastprivate(list)
6     collapse(n)
7     reduction(reduction-identfier:list)
8     schedule(kind[,chunk_size])
9
10 for (int i = 0; i < x; i++) {
11     ...
12 }
```

**Note:** Wichtig: die Iterationen müssen unabhängig voneinander durchführbar sein.

Beispiel

```
1 #include <omp.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #define N 100000
5 int main(int argc, char *argv[]) {
6     int i, a[N];
7
8     for (i = 0; i < N; i++) a[i] = rand();
9
10    #pragma omp parallel for
11    for (i = 0; i < N; i++) {
12        //calc(a, i)
13        printf("a[%d]=%d\n", i, a[i]);
14    }
15
16    return 0;
17 }
```

Notes

Notes

Notes

Notes



- Ein Team ist eine Gruppe von Threads die gleichzeitig ausgeführt werden
  - Am Anfang des Programms gibt es nur einen Thread (Master)
  - Eine **parallel** Directive teilt den aktuellen Thread auf in ein neues Team für den Bereich des Blocks
  - Am Ende verschmelzen die Threads wieder zu einem
- for** Zerlegt die Arbeit einer for-Schleife auf die Threads im Team. Es erzeugt keine neuen Threads, es zerlegt nur die Arbeit auf das Team
- parallel for** ist eine Abkürzung um beides in einer Directive zu tun. Threads erzeugen und Arbeit zerlegen.

Ohne **parallel** Directive läuft das Programm nur im Master-Thread ab.

Quelle: <http://biequit.iki.fi/story/howto/openmp/>, besucht 09.01.2014

For Directive

```
1 #pragma omp parallel for
2 for(int n=0; n<10; ++n) {
3     printf(" %d", n);
4 }
```

Selbstgebaute For Directive

```
1 #pragma omp parallel
2 {
3     int this_thread = omp_get_thread_num();
4     int num_threads = omp_get_num_threads();
5     int my_start = (this_thread ) * 10 / num_threads;
6     int my_end   = (this_thread+1) * 10 / num_threads;
7     for(int n=my_start; n<my_end; ++n)
8         printf(" %d", n);
9 }
```

Quelle: <http://biequit.iki.fi/story/howto/openmp/>, besucht 07.01.2014

#pragma omp sections

Notes

Notes

Notes

Notes

Aufgabe

Führt die in den "section"s definierten Code-Blöcke parallel in den Threads aus

Signatur

```
1 #pragma omp [parallel] sections private (list) firstprivate (list)
2     lastprivate (list) reduction (operator: list) nowait
3
4 {
5     // Task 0
6     #pragma omp section
7     {
8         // Task 1
9     }
10    #pragma omp section
11    {
12        // Task 2
13    }
14    // ...
15 }
```

Notes

---

---

---

---

---

---

---

---

Notes

---

---

---

---

---

---

---

---

Notes

---

---

---

---

---

---

---

---

Beispiel

Tafel

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main() {
5     #pragma omp parallel sections num_threads(4)
6     {
7         printf("Hello from thread %d\n", omp_get_thread_num());
8         #pragma omp section
9         printf("Hello from thread %d\n", omp_get_thread_num());
10        #pragma omp section
11        printf("Hello from thread %d\n", omp_get_thread_num());
12    }
13    return 0;
14 }
```

Wie sieht die Ausgabe aus?

Notes

---

---

---

---

---

---

---

---

#pragma omp single



Aufgabe

Führt die in single definierten Code-Blöcke nur in einem Thread aus

Signatur

```
1 #pragma omp single private (list)
2   firstprivate (list)
3   nowait
```

- für I/O wichtig
- implizite Barriere am Ende

Beispiel

Tafel

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main() {
5     #pragma omp parallel num_threads(4)
6     {
7         #pragma omp single
8         // Only a single thread can read the input .
9         printf("read input\n");
10
11        // Multiple threads in the team compute the results .
12        printf("compute results\n");
13
14        #pragma omp single
15        // Only a single thread can write the output .
16        printf("write output\n");
17    }
18 }
```

Wie sieht die Ausgabe aus?

#pragma omp master

Aufgabe

Block wird nur von Master-Thread ausgeführt

Signatur

```
1 #pragma omp master
```

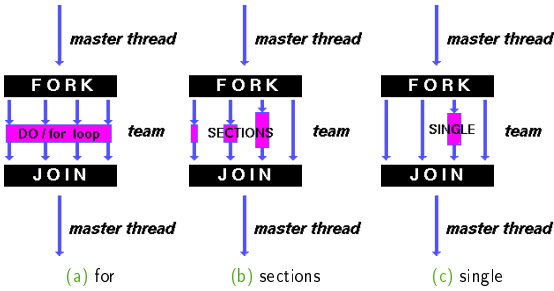
Note: Siehe auch #pragma omp single auf Folie 69

Notes

Notes

Notes

Notes



Quelle: <https://computing.llnl.gov/tutorials/openMP/>, besucht 09.01.2014

### Steuern paralleler Abschnitte

#### Aufgabe

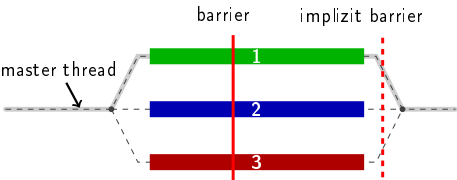
Synchronisiert alle Threads in einem parallelen Block. Die Threads warten an der Grenze, bis alle Threads die Barriere erreicht haben.

#### Signatur

```
1 #pragma omp barrier
```

#### Fäden

```
#pragma omp parallel
{
...
#pragma omp barrier
...
}
```



Notes

Notes

Notes

Notes

Aufgabe

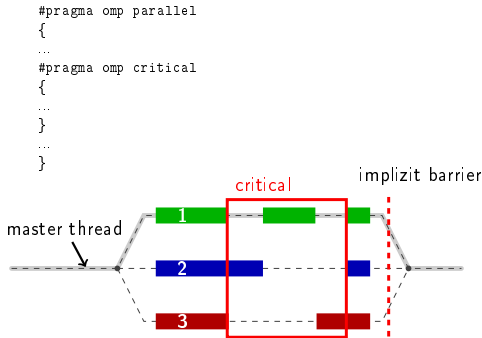
Immer nur ein Thread darf den Code-Block ausführen

Signatur

```
1 #pragma omp critical [(name)]
```

- Critical-Directive wirkt wie ein Lock mit dem Namen “name” das global verfügbar ist
- Critical-Bereiche mit unterschiedlichem Namen stellen unterschiedliche Locks dar, mit gleichem Namen gleiche Locks

Fäden



Aufgabe

Stellt sicher, dass auf variablen Atomar zugegriffen wird

Signatur

```
1 #pragma omp atomic
2 EXPRESSION
```

Note: Schneller als critical aber dafür nicht so flexibel

Beispiel

```
1 int a, b=0;
2 #pragma omp parallel for private(a) shared(b)
3 for(a=0; a<50; ++a) {
4     #pragma omp atomic
5     b += a;
6 }
```

Notes

Notes

Notes

Notes

Aufgabe

Stellt sicher, dass die temporäre Sicht auf die Daten aller Threads die gleiche ist

Signatur

```
1 #pragma omp flush([list])
```

Notes

```
1 #include <stdio.h>
2 #include <omp.h>
3 int main() {
4     int data, flag = 0;
5     #pragma omp parallel sections num_threads(2)
6     {
7         #pragma omp section
8         {
9             data = 1;
10            printf("Thread %d: set data to %d n", omp_get_thread_num(), data);
11            #pragma omp flush(data)
12            flag = 1;
13            #pragma omp flush(flag)
14        }
15        #pragma omp section
16        {
17            while (!flag) {
18                #pragma omp flush(flag)
19            }
20            #pragma omp flush(data)
21            printf("Thread %d: process %d \n", omp_get_thread_num());
22            data++;
23            printf("data = %d\n", data);
24        }
25    }
```

Notes

Wo ist das Problem?

```
1 // Thread 0
2 b=1;
3 #pragma omp flush(b)
4 #pragma omp flush(a)
5 if (a == 0) {
6     // ...
```

```
1 // Thread 1
2 a=1;
3 #pragma omp flush(a)
4 #pragma omp flush(b)
5 if (b == 0) {
6     // ...
```

- ▶ Warum könnte hier der Compiler eine entscheidende Rolle spielen?
  - ▶ Warum findet sich dieser Fehler auch in Java und was hat der Hot-Spot Compiler damit zu tun und warum tritt er beim Debugging nicht auf?
  - ▶ Compiler kann in Thread 0 b verschieben oder sogar weg optimieren, ohne den Ablauf von Thread 0 zu ändern und umgekehrt Thread 1 mit a, da diese in den jeweiligen Threads nicht gelesen werden
  - ▶ Richtig wird es erst mit flush(a,b) da hier a bzw. b nicht unabhängig voneinander verschoben werden können
  - ▶ Sehr schwer zu findende Fehler
- Quelle: [Sch10, S. 51]

Notes

Aufgabe

Beenden des innersten parallelen Blocks des übergebenen Parameters

Signatur

```
1 #pragma omp cancel construct-type-clause
```

construct-type-clause

- ▶ parallel
- ▶ sections
- ▶ for
- ▶ taskgroup

Notes

Aufgabe

Setzt die Anzahl an Threads für die parallelen Bereiche, wenn nichts anderes definiert ist

Signatur

```
1 omp_set_num_threads(int n);
```

Aufgabe

Gibt Anzahl an Threads in der parallelen Umgebung zurück

Signatur

```
1 int omp_get_num_threads();
```

Aufgabe

Test ob Thread innerhalb (Wert ungleich 0) eines parallelen Bereichs aufgerufen wird

Signatur

```
1 int omp_in_parallel();
```

Aufgabe

Gibt maximal mögliche Anzahl an Threads für einen parallelen Block zurück, wenn keine num\_threads clause vorhanden ist

Signatur

```
1 int omp_get_max_threads();
```

Notes

Notes

Notes

Notes

Aufgabe

Gibt die ID des aufrufenden Threads zurück

Signatur

```
1 int omp_get_thread_num()
```

Locks

Aufgabe

Initialisieren eines Locks

Signatur

```
1 void omp_init_lock(omp_lock_t *lock);
```

Aufgabe

Setzen eines Locks. Blockiert den aufrufenden Thread, bis es wieder frei ist

Signatur

```
1 void omp_set_lock(omp_lock_t *lock);
```

Notes

Notes

Notes

Notes

Aufgabe

Testet, ob das Lock belegt (FALSE) ist und wenn nicht (TRUE) holt es sich das Lock. In beiden Fällen läuft das Programm weiter und der Programmierer muss den Code-Pfad bestimmen.

Signatur

```
1 int omp_test_lock(omp_lock_t *lock);
```

Note: Die Funktion lockt das Lock nur, wenn es frei ist! In beiden Fällen blockiert die Funktion nicht!

```
1 #include <stdio.h>
2 #include <omp.h>
3 omp_lock_t simple_lock;
4 int main() {
5     omp_init_lock(&simple_lock);
6
7     #pragma omp parallel num_threads(4)
8     {
9         int tid = omp_get_thread_num();
10
11         while (!omp_test_lock(&simple_lock))
12             printf_s("Thread %d - failed to acquire simple_lock\n",
13                     tid);
14
15         printf_s("Thread %d - acquired simple_lock\n", tid);
16
17         printf_s("Thread %d - released simple_lock\n", tid);
18         omp_unset_lock(&simple_lock);
19     }
20     omp_destroy_lock(&simple_lock);
21 }
```

Quelle: <http://msdn.microsoft.com/de-de/library/6e1yzt8.aspx>, besucht 09.03.2014

Aufgabe

Freigeben des Locks

Signatur

```
1 void omp_unset_lock(omp_lock_t *lock);
```

Aufgabe

Freigeben des Speichers von einem Lock

Signatur

```
1 void omp_destroy_lock(omp_lock_t *lock);
```

Notes

Notes

Notes

Notes

Motivation

Einführung

Erzeugen von parallelen Abschnitten

Steuern paralleler Abschnitte

Fehlerquellen

Locks

Locks - Beispiel

1

#include <stdio.h>

2

#include <omp.h>

3

4

omp\_lock\_t my\_lock;

5

6

int main() {

7

omp\_init\_lock(&my\_lock);

8

9

#pragma omp parallel num\_threads(4)

10

{

11

int tid = omp\_get\_thread\_num( );

12

int i;

13

14

for (i = 0; i < 5; ++i) {

15

omp\_set\_lock(&my\_lock);

16

printf("Thread %d - starting locked region\n", tid);

17

printf("Thread %d - ending locked region\n", tid);

18

omp\_unset\_lock(&my\_lock);

19

}

20

}

21

omp\_destroy\_lock(&my\_lock);

22

23

}

Quelle: <http://msdn.microsoft.com/en-us/library/8zybk13s.aspx>, besucht 11.01.2014

Seite 93

Johannes Hötzer

Parallelrechnen

19-02-2015

Motivation

Einführung

Erzeugen von parallelen Abschnitten

Steuern paralleler Abschnitte

Fehlerquellen

Fehlerquellen

Motivation

Einführung

Erzeugen von parallelen Abschnitten

Steuern paralleler Abschnitte

Fehlerquellen

Fehlerquellen bei OpenMP

Race Conditions:  
Programmergebnis hängt vom genauen zeitliche Verhalten der Threads ab. Die häufigste Ursache ist, dass Variablen unbeabsichtigt als gemeinsam deklariert wurden

Deadlocks (Verklemmungen):  
Threads warten auf gesperrte Daten, welche nicht freigegeben werden

Livelock:  
Threads arbeiten unendlich an verschiedenen Aufgaben

Motivation

Einführung

Erzeugen von parallelen Abschnitten

Steuern paralleler Abschnitte

Fehlerquellen

Race Condition

```
1 int a = 0;
2 // form a team of threads and distribute the loop repetitions among them
3 #pragma omp parallel for default(shared)
4 for (int i = 0; i < n; i++) {
5     a += 1;
6 }
7 // the value of a here may be anything between 1 and n
```

Notes

Notes

Notes

Notes





## Fehlerquellen

## Unnötige Parallelisierung

```
1 #pragma omp parallel num_threads(2)
2 {
3     // ... N code lines
4     #pragma omp parallel for
5     for (int i = 0; i < 10; i++) {
6         myFunc();
7     }
8 }
```

```
1 #pragma omp parallel num_threads
2 {
3     // ... F code lines
4     #pragma omp for
5     for (int i = 0; i < 10; i++) {
6         myFunc();
7     }
8 }
```

Quelle: <http://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers#IDAXKQDC>, besucht 11.01.2014

## Fehlerquellen

## Ändern der Anzahl an Threads in parallelem Block

```
1 #pragma omp parallel
2 {
3     omp_set_num_threads(2);
4     #pragma omp for
5     for (int i = 0; i < 10; i++) {
6         myFunc();
7     }
8 }
```

```
1 omp_set_num_threads(2)
2 #pragma omp parallel
3 // odd
4 #pragma omp parallel num_threads(2)
5 {
6     #pragma omp for
7     for (int i = 0; i < 10; i++) {
8         myFunc();
9     }
10 }
```

Quelle: <http://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers#IDAXKQDC>, besucht 11.01.2014

## Fehlerquellen

## Benutzung von locks ohne Initialisierung

```
1 omp_lock_t myLock;
2 #pragma omp parallel num_threads(2)
3 {
4     ...
5     omp_set_lock(&myLock);
6     ...
7 }
```

```

1 omp_lock_t myLock;
2 omp_init_lock(&myLock);
3 #pragma omp parallel num_threads(2)
4 {
5     ...
6     omp_set_lock(&myLock);
7     ...
8 }

```

Quelle: <http://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers#IDAXKQDC>, besucht 11.01.2014

## Fehlerquellen

Lock wird von anderem Thread freigegeben

```

1  omp_lock_t myLock;
2  omp_init_lock(&myLock);
3  #pragma omp parallel sections
4  {
5      #pragma omp section
6      {
7          ...
8          omp_set_lock(&myLock);
9          ...
10     }
11     #pragma omp section
12     {
13         ...
14         omp_unset_lock(&myLock);
15     }
16 }
17

```

Quelle: <http://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers#IDAXKQDC>, besucht 11.01.2014

## Notes

---

---

---

---

---

---

## Notes

---

---

---

---

---

---

## Notes

[illegible]

## Notes

---

---

---

---

---

---

## Fehlerquellen

```

1 omp_lock_t myLock;
2 omp_init_lock(&myLock);
3 #pragma omp parallel sections
4 {
5     #pragma omp section
6     {
7         ...
8         omp_set_lock(&myLock);
9         ...
10        omp_unset_lock(&myLock);
11        ...
12    }
13    #pragma omp section
14    {
15        ...
16        omp_set_lock(&myLock);
17        ...
18        omp_unset_lock(&myLock);
19        ...
20    }
21 }

```

Quelle: <http://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers#IDAXKQDC>, besucht 11.01.2014

## Fehlerquellen

## Benutzung von locks als Barrieren

```

1 omp_lock_t myLock;
2 omp_init_lock(&myLock);
3 #pragma omp parallel sections
4 {
5     #pragma omp section
6     {
7         ...
8         omp_set_lock(&myLock);
9     }
10    #pragma omp section
11    {
12        ...
13        omp_set_lock(&myLock);
14        omp_unset_lock(&myLock);
15        ...
16    }
17 }
18

```

Quelle: <http://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers#IDAXKQDC>, besucht 11.01.2014

## Fehlerquellen

```

1  omp_lock_t myLock;
2  omp_init_lock(&myLock);
3  #pragma omp parallel sections
4  {
5      #pragma omp section
6      {
7          ...
8          omp_set_lock(&myLock);
9          ...
10         omp_unset_lock(&myLock);
11         ...
12     }
13     #pragma omp section
14     {
15         ...
16         omp_set_lock(&myLock);
17         ...
18         omp_unset_lock(&myLock);
19         ...
20     }
21 }

```

Quelle: <http://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers#IDAXKQDC>, besucht 11.01.2014

## Fehlerquellen

## Gleichzeitige Nutzung von geteilten Ressourcen

```
1 #pragma omp parallel num_threads(2)
2 {
3     printf("Hello World\n");
4 }
```

### Output

```
HellHell oo WorWlodrl
d
```

Quelle: <http://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers#IDAXKQDC>, besucht 11.01.2014

## Notes

[illegible]

## Notes

---

---

---

---

---

---

## Notes

[illegible]

## Notes

---

---

---

---

---

---

Fehlerquellen

Gleichzeitige Nutzung von geteilten Ressourcen

```
1 #pragma omp parallel num_threads(2)
2 {
3     #pragma omp critical
4     {
5         printf("Hello World\n");
6     }
7 }
```

Quelle: <http://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers#IDAXQ0C>, besucht 11.01.2014

Seite 109 | Johannes Hötzer | Paralleltrechnen | 19-02-2015

Abschluss

Zusammenfassung

- ▶ Allgemeiner Überblick über openMP
- ▶ Directiven zur Erzeugung paralleler Abschnitte
- ▶ Directiven und Funktionen zur Steuerung paralleler Abschnitte
- ▶ Nicht behandelt, Tasks, Nested Parallelisierung sowie openMP Variablen

Eight Simple Rules for Designing Multithreaded Applications I

1. Identify Truly Independent Computations
2. Implement Concurrency at the Highest Level Possible
  - ▶ Suche nach hotspot
  - ▶ bottom up (Code Analyzer)
  - ▶ top down
3. Plan Early for Scalability to Take Advantage of Increasing Numbers of Cores
4. Make Use of Thread-Safe Libraries Wherever Possible
5. Use the Right Threading Model
  - ▶ don't use explicit threads if an implicit threading model has all the functionality
  - ▶ e.g. OpenMP

Notes

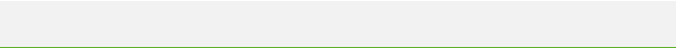
Notes

Notes

Notes

Eight Simple Rules for Designing Multithreaded Applications II

- ▶ explicit threads more complex, more mistake, harder to maintain, but absolute control
6. Never Assume a Particular Order of Execution
  7. Use Thread-Local Storage Whenever Possible or Associate Locks to Specific Data
  8. Dare/Try to Change the Algorithm for a Better Chance of Concurrency
  9. Erwarte nie eine feste Anzahl an Threads



Notes

Notes

Motivation

Wie kann man die Kreiszahl Pi  $\pi$  berechnen?

- ▶ Kettenbruchentwicklung nach Wallis  $\pi = \frac{4}{1 + \frac{1^2}{2 + \frac{3^2}{2 + \frac{5^2}{2 + \frac{7^2}{\ddots}}}}}$
- ▶ Gregory-Leibniz Series  $\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \mp \dots = \frac{\pi}{4}$
- ▶ Sphärische Geometrie
- ▶ BBP-Reihen
- ▶ Monte-Carlo-Algorithmus

Buffon's needle

- ▶ postuliert 1733 vor der Pariser Akademie der Wissenschaften von Georges-Louis Leclerc, Comte de Buffon
- ▶ fragt nach der Wahrscheinlichkeit, dass eine willkürlich geworfene Nadel ein Gitter paralleler Linien schneidet
- ▶ Kann für experimentelle Berechnung von  $\pi$  mit der Monte-Carlo Methode genutzt werden
- ▶ Berechnung ist langsam und ungenau

Notes

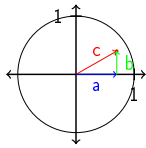
Notes

- Nach einigen Umformungen und Überlegungen

$$\pi = 4 \cdot P \quad (1)$$

- Die Zahl  $\pi$
- Faktor (viertel Kreis)
- Wahrscheinlichkeit ob Treffer im Kreis

## Wahrscheinlichkeit ob Treffer im Kreis



$$a^2 + b^2 = c^2 \quad (2)$$

$$\text{Treffer} = \begin{matrix} \text{true} & [1] & \text{if} & a^2 + b^2 \leq 1^2 \\ \text{false} & [0] & \text{if} & a^2 + b^2 > 1^2 \end{matrix} \quad (3)$$

$$P = \sum_{i=0}^N \frac{\text{Treffer}}{N} \quad (4)$$

## Berechnung von $\pi$

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int main(int argc, char **argv) {
6     int globalCount = 0, globalSamples=10000000;
7     double x, y;
8
9     for(int i = 0; i < globalSamples; ++i) {
10         x = (double)rand() / (double)RAND_MAX;
11         y = (double)rand() / (double)RAND_MAX;
12         if ((x*x + y*y) <= 1.0) globalCount++;
13     }
14     double pi = 4.0 * (double)globalCount / (double)(globalSamples);
15
16     printf("pi is %.9lf\n", pi);
17     return 0;
18 }
```

## Teil V

## Mutex

## Notes

## Notes

## Notes

## Notes

Motivation

Welcher Betrag ist am Ende auf dem Konto?

```
Account account = new Account(“Johannes Hötzer”, 0, Euro);
```

Thread 1	Thread 2
...	...
account.add(5,Euro);	account.add(9,Euro);
...	...
account.print();	account.print();

- ▶ 14 Euro
- ▶ 5 Euro
- ▶ 9 Euro
- ▶ ?? Euro

Inhaltsverzeichnis I

- Allgemein
  - Dekker-Algorithmus
  - Aktives und passives Warten
  - Sprachen mit Mutex

Mutex Umsetzungen

- Erweitertes Mutual Exclusion
  - Locks
  - Semaphores
  - Weitere Mutual Exclusions

Seiteneffekte von Mutexen

Abschluss

Mutex

- ▶ engl. **mutual exclusion**
- ▶ benötigt in Systemen mit gemeinsamem Speicher
- ▶ Eingeführt von Edsger W. Dijkstra 1965 in “Solution of a problem in concurrent programming control” (Dekker’s Algorithmus)[Dij65]
- ▶ Mutex-Verfahren koordinieren den zeitlichen Ablauf nebenläufiger Prozesse/ Threads
- ▶ Verfahren zum Sicherstellen des gegenseitigen Ausschlusses
- ▶ immer nur ein Prozess/Thread kann sich in kritischem Abschnitt befinden
- ▶ bezeichnet eine Gruppe von Verfahren die ineinander überführbar sind

**Note:** Paper [Dij65] sehr lesenswert, nur eine Seite

Dekker-Algorithmus

Dekker-Algorithmus

- ▶ Annahme: 2 Threads
- ▶ gemeinsamer Speicher
- ▶ Nur ein Core
- ▶ 3 Variablen:
  - ▶ 2 Flags (Signalisiert jeweils dem anderen Thread, dass dieser möglicherweise in kritischem Bereich ist)
  - ▶ turn (Token)
- ▶ Busy waiting
- ▶ Scheduling hat keinen Einfluss auf Korrektheit

Notes

Notes

Notes

Notes

```
1 // globale Variablendeklaration
2 boolean flag0 = false;
3 boolean flag1 = false;
4 int turn = 0; // alternativ : turn = 1

1 // Prozess 0
2 p0: flag0 = true;
3 while (flag1) {
4     if (turn != 0) {
5         flag0 = false;
6         while (turn != 0) {
7             // Leeranweisung ( Aktives Warten )
8         }
9         flag0 = true;
10    }
11 }
12 // kritischer Abschnitt
13 // .....
14 turn = 1;
15 flag0 = false;

1 // Prozess 1
2 p1: flag1 = true;
3 while (flag0) {
4     if (turn != 1) {
5         flag1 = false;
6         while (turn != 1) {
7             // Leeranweisung ( Aktives Warten )
8         }
9         flag1 = true;
10    }
11 }
12 // kritischer Abschnitt
13 // .....
14 turn = 0;
15 flag1 = false;
```

Möglicher Ablauf:

```
1 > turn=0
2 > Thread #0: flag0 = true;
3 > Thread #1: flag1 = true;
4 > Thread #0: Eintritt in die Schleife
5 > Thread #1: Eintritt in die Schleife
6 > Thread #0: Die Bedingung turn != 0 wird nicht erfüllt
7 > Thread #1: Die Bedingung turn != 1 wird erfüllt
8 > Thread #0: Erneuter Eintritt in die Schleife, da flag1 gesetzt
9 > Thread #1: flag1 = false;
10 > Thread #0: Die Bedingung turn != 0 wird nicht erfüllt
11 > Thread #1: Die Bedingung turn != 1 wird erfüllt
12 > Thread #0: Eintritt in den kritischen Abschnitt, da flag1 nicht
    gesetzt
13 > Thread #0: turn = 1;
14 > Thread #1: flag1 = true;
```

Möglichkeiten wenn Mutex belegt:

- ▶ Warten auf Freigabe des Mutex in dem Thread Kontrolle an Scheduler abgibt (passives warten)
- ▶ Andere Aufgabe durchführen und auf Mutex Freigabe warten (polling)
- ▶ Nur auf Mutex Freigabe warten (aktives polling/busy waiting)
- ▶ Zugriff verwerfen, wenn z.B. Messwert schon von anderem Thread bearbeitet wird

- ▶ Nahezu alle parallelen Programmiersprachen (Java, C, C++ C#...) unterstützen/bringen Mutexe mit
- ▶ oft Teil der API/Laufzeitumgebung
- ▶ Nur effizient wenn Betriebssystem Mutex unterstützt
- ▶ Es reicht eine Art von Mutex, Semaphore, Monitor, Lock oder Critical Section anzubieten (gegenseitig abbildbar)

Notes

---

---

---

---

---

---

---

---

Notes

---

---

---

---

---

---

---

---

Notes

---

---

---

---

---

---

---

---

Notes

---

---

---

---

---

---

---

---



## Mutex Umsetzungen

- ▶ Hardware Lösungen
- ▶ Software Lösungen

- ▶ Ausschalten der Interrupts während kritischem Abschnitt bei Single Core Architekturen  
Probleme
  - ▶ Timer Interrupt → Keine Zeitmessung
  - ▶ langer kritischer Bereich
- ▶ Busy-waiting bei Multi Core Architekturen
  - ▶ Atomare Test und Set Funktionen
  - ▶ Unterbrechungen durch System möglich
- ▶ Weiter atomare Funktionen
  - ▶ Compare and Swap (CAS)
  - ▶ Unterbrechungen durch System möglich

Quelle: [http://en.wikipedia.org/wiki/Mutual\\_exclusion](http://en.wikipedia.org/wiki/Mutual_exclusion), besucht 22.01.2014

### Software Lösungen mit aktivem Warten

- ▶ Dekker's Algorithmus [Dij65]
- ▶ Peterson's Algorithmus
- ▶ Lamport's bakery Algorithmus
- ▶ Szymanski's Algorithmus
- ▶ Taubenfeld's black-white bakery Algorithmus

Quelle: [http://en.wikipedia.org/wiki/Mutual\\_exclusion](http://en.wikipedia.org/wiki/Mutual_exclusion), besucht 22.01.2014

### Notes

### Notes

### Notes

### Notes

# Erweitertes Mutual Exclusion

## Lock

- ▶ Stellt sicher, dass sich immer nur ein Thread in kritischem Abschnitt befindet
- ▶ Bei Rekursion Deadlock

## Reentrant Mutexes (Rekursive Locks)

- ▶ Thread darf bereits gelocktes Lock mehrfach locken
- ▶ Wichtig für rekursive Algorithmen
- ▶ Muss Lock genauso oft frei geben wie es gelockt wurde
- ▶ Java “synchronized” Keyword ist Reentrant Mutex

## Readers-Writer Lock

- ▶ auch Shared-Exclusive Lock
- ▶ viele Leser
- ▶ ein Schreiber
- ▶ POSIX standard pthread\_rwlock\_t
- ▶ Java ReadWriteLock, ReentrantReadWriteLock

- ▶ Kontrolliert Zugriff auf begrenzte Resource durch mehrerer Prozesse
- ▶ wenn nur eine Resource, auch Binäre Semaphore/Lock genannt
- ▶ Wird mit Anzahl an freien Ressourcen initialisiert
- ▶ Wenn Thread/Prozess Ressourcen möchte wird Variable heruntergesetzt bis sie 0 ist

## Notes

## Notes

## Notes

## Notes

- Monitors
- Message passing
- Barrieren

## Seiteneffekte von Mutexen

- Gefahr von Deadlocks
- Vergessen Lock frei zu geben
- Ausbremsen von Threads (lock contention), weil sie auf Lock warten müssen
- Verhungern von Prozessen weil sie nie das Lock bekommen
- Prioritätsinvertierung, in dem ein Prozess mit niederer Priorität den Prozess mit höher Priorität blockiert
- Zerstören Parallelität
- Lock overhead

## Abschluss

### Notes

### Notes

### Notes

### Notes

- ▶ Allgemeine Funktion/Nutzen von Mutexen
- ▶ Umsetzungen Anhand von Dekker-Algorithmus
- ▶ Realisierungen von Mutexen
- ▶ Erweitertes Mutual Exclusion

Motivation	Einführung	Thread erzeugen und beenden ○○○○	Joining and Detaching Threads	Mutex ○○○
------------	------------	-------------------------------------	-------------------------------	--------------

Teil VI

Posix Threads

Motivation	Einführung	Thread erzeugen und beenden ○○○○	Joining and Detaching Threads	Mutex ○○○
------------	------------	-------------------------------------	-------------------------------	--------------

Inhaltsverzeichnis I

Motivation

Einführung

Thread erzeugen und beenden

- Thread erzeugen
- Thread beenden
- Thread Attribute
- Miscellaneous Routines

Joining and Detaching Threads

Mutex

- Creating and Destroying of Locks
- Locking and Unlocking
- Creating and Destroying of Read-Write-Locks

Motivation	Einführung	Thread erzeugen und beenden ○○○○	Joining and Detaching Threads	Mutex ○○○
------------	------------	-------------------------------------	-------------------------------	--------------

Literatur

- ▶ <http://pubs.opengroup.org/onlinepubs/007904975/basedefs/pthread.h.html>
- ▶ <https://computing.llnl.gov/tutorials/pthreads/>

Notes

Notes

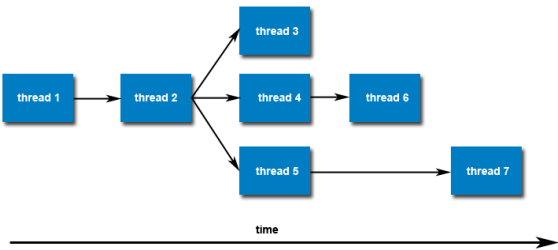
Notes

Notes

Welche Sturkturen/Funktionen benötigt ein paralleler Algorithmus

- ▶ Elementaroperationen (wie sequentiell)
- ▶ Erzeugen parallel-ausführbarer Abschnitte
- ▶ Privaten/Gemeinsamen Speicher
- ▶ Koordination: Kommunikation, Synchronisation, Prozessverwaltung/Threadverwaltung

- ▶ Erweiterung von POSIX
- ▶ POSIX Threads (Pthreads)
- ▶ C/C++ API in POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995) geregelt
- ▶ Implementierungen für \*nix und Windows
- ▶ Inhalt
  - ▶ Thread management
  - ▶ Mutexes
  - ▶ Condition Variables
  - ▶ Synchronization Funktionen



Abbildungsquelle: <https://computing.llnl.gov/tutorials/pthreads/images/peerThreads.gif>

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #define NUM_THREADS 5
5 void *PrintHello(void *threadid){
6     long tid = (long)threadid;
7     printf("Hello World! It's me, thread %ld!\n", tid);
8     pthread_exit(NULL);
9 }
10
11 int main(int argc, char *argv[]){
12     pthread_t threads[NUM_THREADS];
13     for(long t=0; t<NUM_THREADS; t++){
14         printf("In main: creating thread %ld\n", t);
15         int rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
16         if (rc){
17             printf("ERROR: return code from pthread_create() is %d\n", rc);
18             exit(-1);
19         }
20     }
21     pthread_exit(NULL);
22     return 0;
23 }
```

Notes

Notes

Notes

Notes

Kompilieren

```
cc -Wall -std=c99 -D _BSD_SOURCE -pthread -o myprogramm myprogramm.c
```

Ausführen

```
./mympprogramm -a -b -c
```

Note: Gültig für Linux mit GCC

Notes

Notes

Thread erzeugen und beenden

Aufgabe

Erzeugen eines neuen Threads mit dem Code als Function-Pointer

Signatur

```
1 int pthread_create(pthread_t* restrict thread,           //Thread Objekt
2                     const pthread_attr_t* restrict attr, //Erzeugerte Thread
3                     void *(*start_routine)(void*),       //Function-Pointer zur
4                     //Funktion die parallel ausgef. & hnt werden soll
4                     void* restrict arg);                 //Parameter fSr Thread
```

- Gibt ThreadID zurück
- Default Werte wenn pthread\_attr\_t\* NULL

```
1 int rc;
2 long t;
3
4 for(t=0; t<NUM_THREADS; t++)
5 {
6     printf("Creating thread %ld\n", t);
7     rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
8     ...
9 }
```

Notes

Notes

Notes

Beenden eines Threads

1 void pthread\_exit(void \*value\_ptr);

value\_ptr wird möglicherweise wartendem Thread zur Verfügung gestellt

Abbrechen eines Threads

1 int pthread\_cancel(pthread\_t thread);

Notes

- ▶ Thread endet normal nach bearbeiten seiner Start-Routine
- ▶ Thread endet nach Aufruf der Subroutine pthread\_exit, unabhängig ob er fertig mit seiner Arbeit ist, oder nicht
- ▶ Thread wird abgebrochen nach Aufruf der Subroutine pthread\_cancel
- ▶ Thread wird abgebrochen weil Vater-Prozess beendet wird
- ▶ Thread wird abgebrochen weil main endet ohne pthread\_exit aufzurufen

Notes

Erzeugen eines Attribut-Objekts

Erzeugen eines Attribut-Objekts zum setzen spezieller Thread-Attribute (z.B. Joinable)

1 int pthread\_attr\_init(pthread\_attr\_t \*attr);

Löschen des Attribut Objekts

1 int pthread\_attr\_destroy(pthread\_attr\_t \*attr);

Eigene Thread ID

```
1 pthread_t pthread_self(void);
```

Vergleich zweier Thread IDs

Vergleicht zwei Thread IDs ob diese gleich sind. Der "==" Operator reicht nicht zum Vergleich aus!

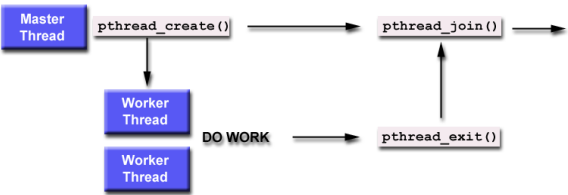
```
1 int pthread_equal(pthread_t t1, pthread_t t2);
```

Notes

Notes

Joining and Detaching Threads

Notes



Abbildungsquelle: <https://computing.llnwd.net/tutorials/threads/>

Notes

Warten auf einen anderen Thread

```
1 int pthread_join(pthread_t thread, void **value_ptr);
```

Loslösen eines Threads

d.h. auf diesen Thread kann nicht mehr gewartet werden

```
1 int pthread_detach(pthread_t thread);
```



Joining and Detaching Threads

Attribut setzen

Setzen des Attributs, das auf Thread gewartet (PTHREAD\_CREATE\_JOINABLE) werden kann oder nicht ( PTHREAD\_CREATE\_DETACHED)

```
1 int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

Abfragen des Attributs

```
1 int pthread_attr_getdetachstate(const pthread_attr_t *attr, int * detachstate);
```

Notes

Joining and Detaching Threads - Beispiel

```
1 pthread_t thread[NUM_THREADS];
2 pthread_attr_t attr;
3
4 pthread_attr_init(&attr);
5 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
6
7 for(t=0; t<NUM_THREADS; t++) {
8     rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
9 }
10
11 pthread_attr_destroy(&attr);
12 for(t=0; t<NUM_THREADS; t++) {
13     rc = pthread_join(thread[t], &status);
14 }
15
16 pthread_exit(NULL);
```

Notes

Mutex

Notes

Creating and Destroying of Locks

Creating and Destroying Mutexes

Erzeugen eines Mutex

Dynamisch: (attr kann NULL sein)

```
1 int pthread_mutex_init(pthread_mutex_t *restrict mutex,
2                        const pthread_mutexattr_t *restrict attr);
```

Statisch:

```
1 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Löschen eines Mutex

```
1 int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Notes

## Locking and Unlocking

## Locking and Unlocking Mutexes

## Locken eines Mutexes

```
1 int pthread_mutex_lock(pthread_mutex_t *mutex);
```

### Testen ob Mutex belegt ist

```
1 int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

## Freigeben eines Mutex

```
1 int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

## Locking and Unlocking

## Locking - Beispiel

```

1 pthread_t callThd[5];
2 pthread_mutex_t mutexsum;
3 //...
4 void *dotprod(void *arg) {
5     pthread_mutex_lock(&mutexsum);
6     sum += mysum;
7     pthread_mutex_unlock(&mutexsum);
8
9     pthread_exit((void*) 0);
10 }
11 //...
12 pthread_mutex_init(&mutexsum, NULL);
13 for(i=0; i<5; i++) {
14     pthread_create(&callThd[i], NULL, dotprod, (void *)i);
15 }
16 pthread_mutex_destroy(&mutexsum);

```

## Creating and Destroying of Read-Write-Locks

## Creating and Destroying Read-Write-Locks

## Erzeugen eines Read-Write-Locks

[illegible]

## Löschen eines Read-Write-Locks

```
1 int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

## Creating and Destroying of Read-Write-Locks

## Locking the Read-Write-Locks

Lock zum schreiben erhalten

```
1 int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

Lock zum lesen erhalten

```
1 int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

## Notes

---

---

---

---

---

---

## Notes

---

---

---

---

---

---


## Notes

[illegible]


## Notes

[illegible]





 **GEORGE ALMÁSI, RALPH BELLOFATTO, JOSÉ BRUNHEROTO, CĂLIN CAȘCAVAL, JOSÉ G. CASTAÑOS, PAUL CRUMLEY, C. CHRISTOPHER ERWAY, DEREK LIEBER, XAVIER MARTORELL, JOSÉ E. MOREIRA, RAMENDRA SAHOO, ALDA SANOMIYA, LUIS CEZE, and KARIN STRAUSS.**

An overview of the bluegene/l system software organization.  
*Parallel Processing Letters*, 13(04):561–574, 2003.


 **Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat.**  
A scalable, commodity data center network architecture.  
*SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008.

Notes

 **Gene M. Amdahl.**  
Validity of the single processor approach to achieving large scale computing capabilities.  
In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67* (Spring), pages 483–485, New York, NY, USA, 1967. ACM.


 **Clay Breshears.**  
*The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications.*  
O'Reilly Media, 0 edition, 5 2009.

Notes

 **David Culler, J.P. Singh, and Anoop Gupta.**  
*Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design).*  
Morgan Kaufmann, 1 edition, 8 1998.

 **E. W. Dijkstra.**  
Solution of a problem in concurrent programming control.  
*Commun. ACM*, 8(9):569–, September 1965.


Notes


 **William James Dally and Brian Patrick Towles.**  
*Principles and Practices of Interconnection Networks (The Morgan Kaufmann Series in Computer Architecture and Design).*  
Morgan Kaufmann, 1 edition, 1 2004.


 **Victor Eijkhout.**  
*Introduction to High Performance Scientific Computing.*  
lulu.com, 10 2012.

 **S.H. Fuller and L.I. Millett.**  
Computing performance: Game over or next level?  
*Computer*, 44(1):31–38, 2011.

Notes

 **Message Passing Interface Forum.**  
*MPI: A Message-Passing Interface Standard, Version 3.0.*  
High Performance Computing Center Stuttgart, 2012.


 **Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta.**  
*Introduction to Parallel Computing (2nd Edition).*  
Addison-Wesley, 2 edition, 1 2003.

 **William Gropp, Ewing L. Lusk, and Anthony Skjellum.**  
*Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation).*  
The MIT Press, second edition edition, 11 1999.


Notes


 **William Gropp, Ewing L. Lusk, and Rajeev Thakur.**  
*Using MPI-2: Advanced Features of the Message Passing Interface (Scientific and Engineering Computation).*  
The MIT Press, 1st edition, 11 1999.


 **Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea.**  
*Java Concurrency in Practice.*  
Addison-Wesley Professional, 1 edition, 5 2006.

 **Dawn Griffiths David Griffiths.**  
*C von Kopf bis Fuß.*  
O'Reilly Vlg. GmbH und Co., 9 2012.


Notes


 **John L. Gustafson.**  
Reevaluating amdahl's law.  
*Commun. ACM*, 31(5):532–533, May 1988.


 **Georg Hager.**  
Introduction to high performance computing for scientists and engineers (chapman & hall/crc computational science), 9 2012.

 **David R. Hanson.**  
*C Interfaces and Implementations: Techniques for Creating Reusable Software.*  
Addison-Wesley Professional, 1 edition, 8 1996.


Notes


 **J. Kim, W.J. Dally, S. Scott, and D. Abts.**  
Technology-driven, highly-scalable dragonfly topology.  
*In Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 77–88, June 2008.


 **John Kim.**  
*High-radix interconnection networks.*  
PhD thesis, Stanford University, 2008.

 **Leslie Lamport.**  
Time, clocks, and the ordering of events in a distributed system.  
*Commun. ACM*, 21(7):558–565, July 1978.

Notes

 **Chuck Lam.**  
*Hadoop in Action.*  
Manning Publications, 1 edition, 12 2010.

 **Britta Nestler, Harald Garcke, and Björn Stinner.**  
Multicomponent alloy solidification: Phase-field modeling and simulations.  
*Phys. Rev. E*, 71:041609, Apr 2005.

 **David Padua, editor.**  
*Encyclopedia of Parallel Computing (Springer Reference).*  
Springer, 2011 edition, 9 2011.

---

---

---

---

---

---

---

 **Thomas Rauber.**  
*Parallele Programmierung (eXamen.press) (German Edition).*  
Springer, 3. aufl. 2012 edition, 9 2012.

 **Thomas Rauber and Gudula Rünger.**  
Parallele programmierung (examen.press) (german edition), 9 2012.

 **Josef Schüle.**  
*Paralleles Rechnen.*  
Oldenbourg Wissensch.Vlg, 9 2010.

 **Anthony Skjellum.**  
*MPI - Eine Einführung.*  
Oldenbourg Wissensch.Vlg, 6 2007.

---

---


---


---


---

---

---

 **Peter van der Linden.**  
*Expert C Programming: Deep C Secrets.*  
Prentice Hall, 1st edition, 6 1994.

 **Eric F. Van de Velde.**  
*Concurrent scientific computing.*  
Texts in applied mathematics ; 16. Springer, New York, 1994.

 **Alexander Vondrous, Michael Selzer, Johannes Hötzer, and Britta Nestler.**  
Parallel computing for phase-field models.  
*International Journal of High Performance Computing Applications*, 2013.

---

---


---

---

---

---

---

 **G. Zumbusch.**  
Tuning a finite difference computation for parallel vector processors.  
In *Parallel and Distributed Computing (ISPDG), 2012 11th International Symposium on*, pages 63–70, June 2012.

---

---

---

---

---

---

---

Autor: Johannes Hötzer  
E-mail: johannes.hoetzer+hpc@gmail.com  
Build Date: 19. Februar 2015  
Git hash: 4cde039ad2fa31493565d1fc0f8009583e511f8a

Notes

---

---

---

---

---

---

---

Notes

---

---

---

---

---

---

---

Notes

---

---

---

---

---

---

---

Notes

---

---

---

---

---

---

---