Architecture of Parallel Computer Systems
- Efficient HPC Systems -

WS 15/16

L.079.05722
Master Course for Informatik and Computer Engineering
V2/Ü1

www.uni-paderborn.de/pc2

Organizational

- Lecturer:
  Jens Simon (PC²)
- Lecture: Dienstags, **13:45 – 15:15, O1.258**
- Exercise: Dienstags, **15:15 – 16:00, O1.258**
- News about Courses:
  http://www.uni-paderborn.de/pc2/people/jens-simon/courses/

Course Assessment

- Attendance at
  - Lecture (2 SWS)
  - Exercise (1 SWS)
- Submission of solved exercises
  - Sample solutions optionally, if provided by participants
- Examination
  - Oral Exam

Lecture Procedure

- Slides are provided immediately after the lecture on the web site of the lecture
- A script of the lecture is not available
- Lecture targets on
  - Introduction in parallel programming and programming methods
  - Overview on architectures of parallel systems
  - Different aspects on efficiency
  - Technologies of from processors up to communication networks
  - Small servers as well as whole data centers
  - ...

Exercise Procedure

- Theoretical exercises associated to the lecture
- Practical exercises on HPC systems of the PC²
  - Send initially an email with name and IMT account to me

About Me

- Dr. rer. nat. Jens Simon
  - Career:
    - 1992 to 1998: Universität Paderborn / PC²
    - 1999 to 2001: Konrad-Zuse Institut Berlin (ZIB)
    - Since then: PC²
  - Lecturer:
    - Courses
    - Project groups
  - Research:
    - Computer systems, (specialized) processors
    - High Speed communication networks
    - Middleware for HPC-Systeme
  - Services:
    - Writing proposals
    - Project work
    - HPC consultant
    - ...

Literature

Computer architecture

- Culler, D. E.; Singh, J. P.; Gupta, A.: *Parallel computer architecture : A hardware/software approach*, 1999
- Rauber, T.; Rünger, G.: *Parallele Progammierung*, Springer, 2012
- Hwang, K.: *Advanced Computer Architecture, 1993*
- Hennessy, J.; Patterson D.: Computer Architecture – A Quantitative Approach, 1996
- Tanenbaum, A.; Computerarchitektur – Strukturen – Konzepte – Grundlagen, 2005
- Barros L.A., Hölzle U; Datacenter As A Computer, 2009
- Kaxiras S., Martonosi, M.; Computer Architecture Techniques for Power-Efficiency, 2008

Literature

Parallel programming

- Andrews, G. R.: *Multithreaded, Parallel, and Distributed Programming*, 2000
- Leopold, C.: Parallel and Distributed Computing – a survey of models, paradigms, and approaches, 2001
- Tanenbaum, A.: Moderne Betriebssysteme, 2009
- Groop, W.; Lusk, E.; Skjellum, A.: Using MPI – Portable Parallel Programming with the Message-Passing Interface, 1999
- Groop, W.; Lusk, E.; Thakur, R.: Using MPI-2 – Advanced Features of the Message-Passing Interface, 1999
- Chapman, B.; Jost, G.; van der Pas, R.: Using OpenMP – Portable Shared Memory Parallel Programming, 2007

Acknowledgement

Parts of the lecture are inspired by the book

"**Parallel Computer Architecture - A Hardware / Software Approach**" *von Culler, Singh und Gupta*

Parallel Computing: Users‘ View

Warum Paralleles Rechnen?

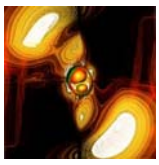
- Die Welt ist höchst parallel – ein bewährtes Prinzip
  - Physikalische Abläufe sind parallel
    - Kristallwachstum, Gravitationsfelder, Chemische Reaktionen, ...
  - Soziale und Arbeitsprozesse sind parallel
    - Ameisenkolonie, Straßenverkehr, Produktionsstätten, Finanzwirtschaft, Universitäten, Politik, ...
  - Menschliches Gehirn, Sensorik, Lernverhalten sind parallel
    - Denken, Sehen, Fühlen, Evolutionsprozesse, ...
- Eine Koordination der größtenteils unabhängigen Prozesse ist aber in den meisten Fällen notwendig

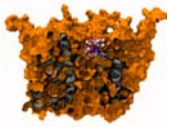
Das Prinzip der Parallelität bereits bei der Beschreibung der Abläufe nutzbar

Warum Parallelverarbeitung?

- Einzelprozessorsysteme bieten nicht genügend Ressourcen für viele Anwendungsbereiche
  - Trotz des rasanten Geschwindigkeitszuwachses bei PCs.
- Sogenannte „Grand-Challange“ Anwendungen benötigen bereits heute um einige Größenordnungen mehr Rechenleistung als die eines einzelnen PCs.
  - Klimaveränderung, Bioinformatik, Astrophysik, ...
- Kopplung verschiedener Methoden für das Auffinden einer besseren Lösung
  - Kooperative Systeme
- Höhere Anforderungen an Sicherheit und Verfügbarkeit
  - Redundante Systeme

Einsatzgebiete von Hochleistungsrechnern (HPC)

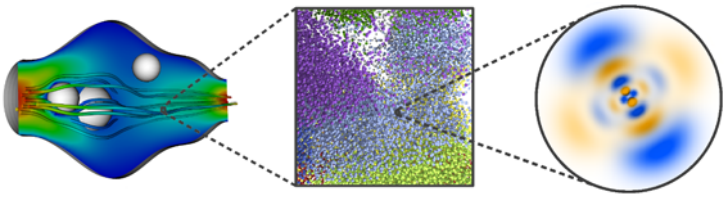




- Forschung
  - Klimawandel, Astrophysik, …
- Produktentwicklung
  - Mikrochip-Entwurf, Werkstofftests, Virtuelle Prototypen, …
- Informationssysteme
  - Datenbanken, Suchmaschinen, Wetterdienste, …
- Virtuelle Umgebungen
  - Fahrzeugsimulationen, Architekturplanungen, …
- Gesundheitswesen
  - Medikamentenentwicklung, Genomanalyse, …
- Finanzdienstleistungen
  - Fonds- u. Risikoanalysen, High-Frequency-Trading, …
- …

HPC in der Forschung

- Simulation
  - dritte Säule in der Forschung, neben Theorie und Experiment
  - sehr rechenintensiv und teilweise mit sehr hohem Speicherbedarf
  - zur Untersuchung von sehr langsam oder sehr schnell ablaufenden Prozessen
  - ohne störende äußere Einflüsse
  - zur Vermeidung gefährlicher Experimente
  - Nutzung unterschiedlicher Skalen
    - Z.B. Physik: Astrophysik bis hin zur Quantenmechanik



Kontinuumsmethode Molekulardynamik Quantenmechanik

Performance Metrics

- FLOPS, or FLOP/S: FLoating-point Operations Per Second
  - MFLOPS: MegaFLOPS, 10^6 flops
  - GFLOPS: GigaFLOPS, 10^9 flops, home PC
  - TFLOPS: TeraGLOPS, 10^12 flops, fast servers
  - PFLOPS: PetaFLOPS, 10^15 flops, present-day supercomputers (www.top500.org)
  - EFLOPS: ExaFLOPS, 10^18 flops, by 2020
- MIPS: Mega Instructions per Second
  - Equal to MegaHertz (if only 1IPS)
  - von Neumann computer had 0.00083 MIPS

Performance Metrics

- Theoretical peak performance R_theor
  - maximum FLOPS a machine can reach in theory.
    - Clock_rate*no_cores*no_FPU/CORE*INST/CLOCK
    - 3 GHz, 16 cores, 4 FPU/CORE, 2 INST/CLOCK → R_theor=3*10^9 * 16 * 4 * 2 = 384 GFLOPS
- Real performance R_real
  - FLOPS for specific operations, e.g. vector multiplication
- Sustained performance R_sustained
  - performance on an application, e.g. CFD

R_sustained << R_real << R_theor

Not uncommon
R_sustained < 10%R_theor

Most Powerful Supercomputer since 06/2013

- Name: Tianhe-2
- Place: Guangzhou, China
- Vendor: National University of Defense Technology, Inspur
- Purpose: Research and Education

- LINPACK: 33.86 Petaflops
- Floor Space: 720 m²
- Racks: 162 (125 compute, 13 Arch switches, 24 storage)
- Nodes: 16,000 (2 CPU sockets, 3 Xeon-Phi cards, 64GB)
- Sockets: 32,000 Intel Ivy Bridge Xeon,2.2GHz and 48,000 Xeon Phi
- Cores 3,120,000
- Memory: 1.4PB (1024TB CPU, 384TB Xeon PHI)
- Power consumption: 17.8MW (plus 24MW cooling)
- OS: Kylin Linux

Source: top500.org

Fastest European Supercomputer 11/2014

- "Piz Daint", Cray XC30
- CSCS, Swiss
- 6,27 PetaFLOP/s Linpack performance (7.8 PFlop/s peak)
- 5,272 nodes, each with one Intel E5-2670 + nVIDIA K20X
- Aries Dragonfly network topology (3 x 14 Gbit/s interconnect)
- total
  - 42,176 CPU-cores and 73,808 streaming processors
  - 169 TByte main memory
  - 2,3 MWatt power consumption
  - Floorspace ca. 60 m²
  - Investment: ca. €?? Mio.

Fastest German Supercomputer 6/2014

- "JuQueen", Blue Gene/Q, IBM
- Forschungszentrum Jülich(FZJ)
- 5.01 PetaFLOP/s Linpack performance (5.9 PFlop/s peak)
- 28,672 nodes, each with one IBM PowerPC A2 (16 cores)
- 5 D torus 40 Gbit/s interconnect
- total
  - 458,752 cores with 1.6 GHz
  - 448 TByte main memory
  - 2.3 MWatt power consumption
  - Floorspace ca. 80 m²
  - Investment: ca. €?? Mio.

European Top500-Liste 6/2015

Rank	Site	System	Cores	RMAX (TFLOP/s)	RPEAK (TFLOP/s)	POWER [kW]
6	CSCS, Switzerland	Cray XC30	115,984	**6,2771**	7,788	2.32
9	FZJ, Germany	IBM BlueGene/Q	458,752	**5,008**	5,872	2,30
17	Exploration & Production - Eni S.p.A. Italy	IBM iDataPlex	72,000	**3,188**	4,605	1,23
20	LRZ, Germany	IBM iDataPlex	147,456	**2,897**	3,185	3,42
21	LRZ, Germany	IBM NeXtScale	86,016	**2,813**	3,578	1,48
23	HLRS, Germany	Cray XC40	94,608	**2,763**	3,784	1,51
29	Total Exploration, France	SGI ICE X	100,400	**2,098**	2,296	2,12
32	CINECA, Italy	IBM BlueGene/Q	163,840	**1,788**	2,097	822
35	EPSRC, UK	CrayXC30	118,080	**1,652**	2,550	
36	GENCI-CINES, France	Bull bullx DLC	50,544	**1,628**	2,102	935

Supercomputer wird zum Mobiltelefon

- Rechenleistung zur Lösung eines Gleichungssystems (Linpack-Benchmark)

1979: Cray 1 und Fortran
- 3.4 MFlop/s
- $5 - $9 Mio.
- 5.25 t
- 115 kW

32 Jahre

1 : 38 →
12.000 : 1 →
40.000 : 1 →
25.000 : 1 →

2012: Smartphone Samsung Galaxy S3 mit Android 4.1.2 und Java
- 130 MFlop/s
- $600
- 133 g
- ca. 5 Watt

Der Weg zum optimalen Supercomputer

Das Vorbild
- Rechenleistung des Gehirns angenommen als 100 TFlop/s (H. Moravec) oder auch 10 PFlop/s (R. Kurzweil)
- Speicherleistung der Synapsen ca. 2 Petabyte
- Gehirn: ca. 1,4 kg Gewicht, 20 Milliarden Nervenzellen, 15 – 20 Watt chemische Leistungsaufnahme

Trotz der rasanter Entwicklung von Technik und Wissenschaft (im Wesentlichen exponentielles Wachstum) noch keine derart effiziente Maschine in Aussicht

Beispiel: LHC Computing Grid

Beschleunigerring, Kollision unterschiedlicher Elementarteilchen für Higgs-Boson Nachweis

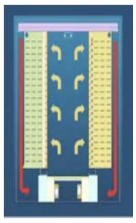
- Large Hadron Collider (LHC) im CERN liefert Messdaten („Events“)
- >140 Rechenzentren in 33 Ländern
- > 100.000 Prozessor-Cores für die Analyse der Messdaten im CERN
- > 140 PetaBytes Speicherkapazität im CERN ($140 \cdot 10^{15}$ Byte)
- Mehrebenenstruktur
  - Tier-0: CERN: Rohdaten
  - Tier-1: vorberechnete Daten
  - Tier-2: Datenverteilung und zentrale Rechenkapazitäten
  - Tier-3: lokale Rechenzentren
  - Tier-4: PCs der Wissenschaftler

Simon, CERN 2006

Beispiel: Google Data Center

- 45 Container, jeder mit
  - Ca. 1 000 Server
  - 250 kW Stromverbrauch
- ~ 4 bis 5 kW pro “Rack”

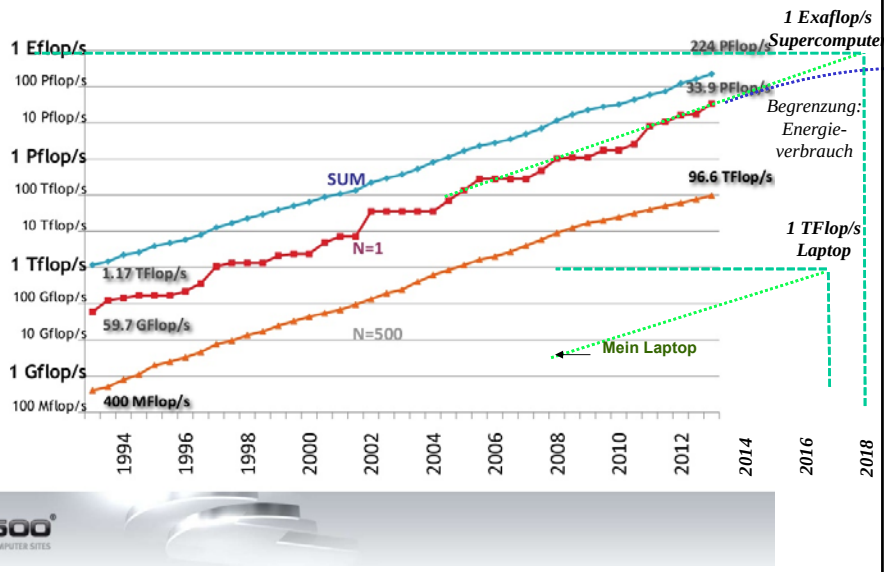
Quelle: Google™



Google Server-Knoten (Stand 2010)

- 2 Prozessoren
- Proprietäre double-wide DIMMs
- einzelner Lüfter pro Knoten (im Netzteil)
- nur 12 Volt Stromversorgung zum Knoten
- 12 Volt USV (Batterie)
- Verkabelung nur von der Frontseite

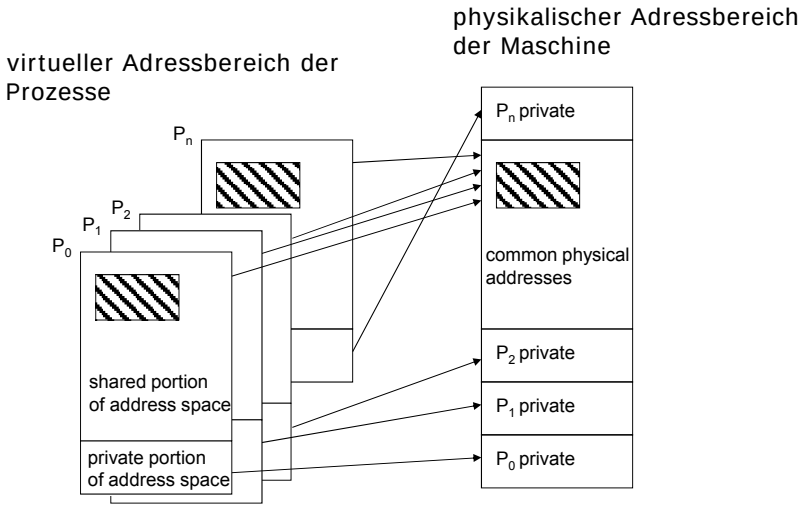
Wird dieses immer so weitergehen?



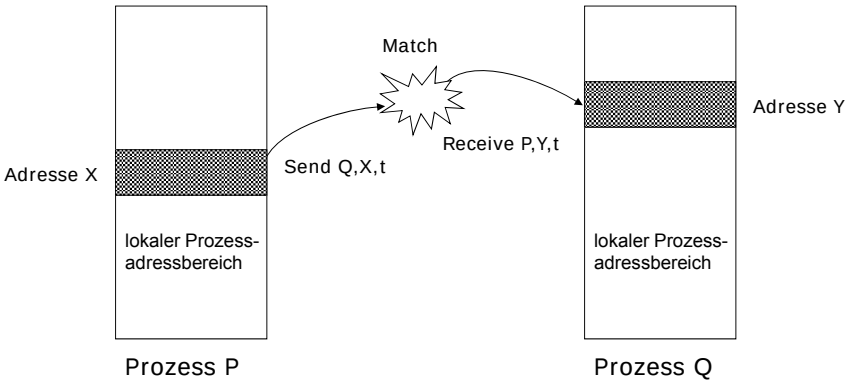
Programmierung von Parallelrechnern

- Speichermodelle
  - Gemeinsamer Speicher
  - Verteilter Speicher
- Programmiermethoden
  - Speichergekoppelt
  - Nachrichtengekoppelt
- Programmierkonzepte
  - Annotierung sequentieller Programme (OpenMP)
  - Mehrere Ausführungsfäden - Threads (PThreads)
  - Expliziter Nachrichtenaustausch - Message-Passing (MPI)

Speichermodell für „Shared-Memory“ Programme



Speichermodell für „Message-Passing“ Programme



Programmiermethoden

- Nachrichtengekoppelte Multiprozessorsysteme
  - Prozesse kommunizieren mittels expliziten Nachrichtenaustausch
    Prozess$_1$: *send(x);* Prozess$_2$: *receive(x);*
  - Standards: MPI-1 und MPI-2

- Speichergekoppelte Multiprozessorsysteme
  - Zugriff auf gemeinsame Variablen *y = x*; oder *shmem_get/shmem_put;*
  - Synchronisation notwendig *lock(mutex_x); ... unlock(mutex_x);*
  - Nutzung von Multithreading: leichtgewichtige Prozesse
  - POSIX-Threads als Standard (PThreads)
  - durchgängiges Konzept für Workstations, Multiprozessor-Workstations (SMPs) und für die großen verteilten Multiprozessorsysteme aller Arten

Programmierkonzepte

- POSIX Threads
  - Gemeinsamer Speicher und Threads

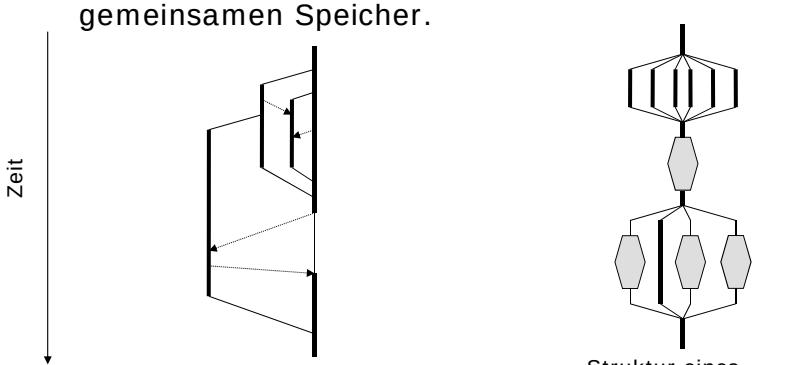
- OpenMP
  - Erweiterung der Programmiersprachen C/C++ und Fortran

- Message Passing Interface (MPI)
  - Standardisierter Nachrichtenaustausch

- PGAS
  - Partitioned Global Address Space

Ausführungsfäden

- Im Shared-Memory-Modell laufen mehrere Tasks (Threads), oftmals mit unterschiedlichen Funktionalitäten parallel ab. Die Tasks kommunizieren untereinander durch Schreiben und Lesen auf einem gemeinsamen Speicher.



Struktur eines Multi-Threading-Programms

Struktur eines OpenMP-Programms

PThreads

- PThreads ist ein POSIX-Standard für Threads.
- Bietet Funktionen zur Verwaltung von Threads.
- Schnittstelle zu vielen sequentiellen Programmiersprachen.
- Besteht aus einer Bibliothek und Header-Dateien
  - *Header-Datei*
    #include <pthread.h>
  - Bibliothek
    libpthread.a
- Compiler-Aufruf

```
gcc –pthread file.c        # GNU Compiler, Linux
icl /pthreads file.c       # Intel Compiler, Windows
```

PThread-Bibliothek

- Erzeugen eines POSIX Threads
  int ***pthread_create*** *(pthread_t *thread_id,*
  *const pthread_attr_t *attributes,*
  *void *(*thread_function)(void *),*
  *void *arguments);*
- Terminieren eines POSIX Threads
  – nach dem Rückkehr aus der Funktion
  – durch Aufruf der Rückkehrfunktion
  int ***pthread_exit*** *(void *status);*
- Warten auf Terminierung eines Threads
  int ***pthread_join*** *(pthread_t thread, void **status_ptr);*
- Rückgabe der Identität des Threads
  pthread_t ***pthread_self*** *();*
- Vergleich von Thread-Ids
  int ***pthread*** *(pthread_t t1, pthread_t t2);*

PThread-Bibliothek

- Mutex Variablen
  int ***pthread_mutex_init*** *(mutex, attr);*
  int ***pthread_mutex_destroy*** *(mutex);*
  int ***pthread_mutexattr_init*** *(attr);*
  int ***pthread_mutexattr_destroy*** *(attr);*
  int ***pthread_mutex_lock*** *(mutex);*
  int ***pthread_mutex_trylock*** *(mutex);*
  int ***pthread_mutex_unlock*** *(mutex);*
- Condition Variablen
  int ***pthread_cond_init*** *(cond, attr);*
  int ***pthread_cond_destroy*** *(cond);*
  int ***pthread_condattr_init*** *(attr);*
  int ***pthread_condattr_destry*** *(attr);*
  int ***pthread_cond_wait*** *(cond, mutex);*
  int ***pthread_cond_signal*** *(cond);*
  int ***pthread_cond_broadcast*** *(cond);*

Beispiel: POSIX Threads

```
/* Zuerst wird die Pthread-Bibliothek eingebunden */
#include <pthread.h>
...
/* Definiton der Thread-Routine */
void *routine(void *arg) {
   printf("Hello World!\n");
   pthread_exit ((void *) 0);
}
...
int main() {
   /* Definition der benötigten Variablen */
   pthread_t          thread;
   pthread_attr_t     attr = NULL;
   ...
   /* Endlich der Aufruf */
   err = pthread_create (&thread, attr, routine, (void *)arg);
   if (err) {
      printf ("FEHLER! pthread_create() : %d\n", err);
      exit(-1);
   }
}
```

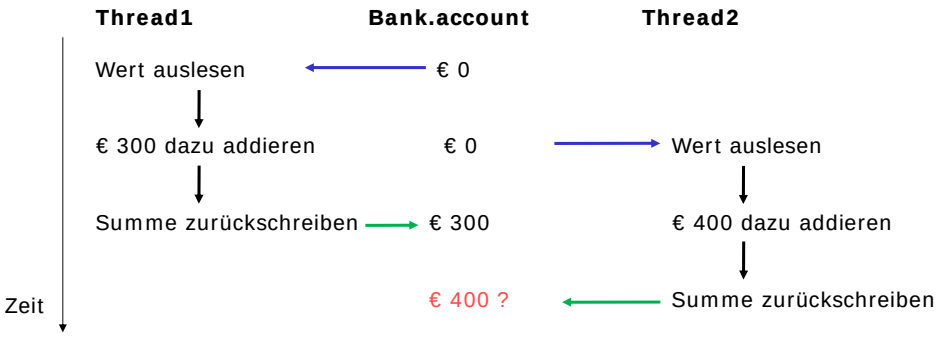

PThreads: Join

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #define NUM_THREADS 3
4  ...
5  /* Thread-Routine */
6  void *routine (void *arg) {
7    sleep(1);
8    printf("Thread Nummer: %d.\n", arg);
9    pthread_exit ((void *) 0);
10 }
11 ...
12 int main() {
13   /* define Mutex-variables */
14   pthread_t       thread[NUM_THREADS];
15   pthread_attr_t  attr;
16   int             i, status, err;
17
18    /* set detach status to JOINABLE */
19    pthread_attr_init (&attr);
20    pthread_attr_setdetachstate (&attr,
21            PTHREAD_CREATE_JOINABLE);
22 /* Start all threads */
23   for(i=0; i < NUM_THREADS; i++) {
24    err = pthread_create (&thread[i], &attr,
25                          routine, (void *) (i+1));
26    if (err) printf("(main) ERROR - Creation failed...\n");
27   }
28
29   ...
30   /* Join with previously startet threads */
31   for(i=0; i < NUM_THREADS; i++) {
32    err = pthread_join (thread[i], (void **)&status);
33    if (err) printf("(main) ERROR - No join...\n");
34   }
35   printf( "All threads are successfully executed!\n");
36 }
```

Problem: Synchronisation

- Anforderung: Geregelter und sicherer Zugriff auf Daten und Methoden, die von mehreren Threads genutzt werden.

Beispiel:



Mutex unter POSIX Threads

Mutex hat zwei grundlegende Funktionen.

- Lock:
  - ist Mutex frei, dann belegt der Thread den Mutex
  - ist Mutex belegt, dann blockiert der Thread bis der Mutex frei wird
- Unlock:
  - gibt Mutex frei

- Initialisierung

  int ***pthread_mutex_init*** *(pthread_mutex_t *mut, const pthread_mutexattr_t *attr);*

- Destruktion

  int ***pthread_mutex_destroy*** *(pthread_mutex_t *mut);*

Mutex

- *Lock*
  int ***pthread_mutex_lock*** *(pthread_mutex_t *mut);*
  – falls Mutex frei, belege den Mutex
  – falls Mutex belegt, warte bis Mutex frei wird

- *Unlock*
  int ***pthread_mutex_unlock*** *(pthread_mutex_t *mut);*
  – Mutex freigeben (falls Thread auf Lock wartet, belege den Mutex mit dem Lock eines wartenden Threads)

- Mutex Test
  int ***pthread_mutex_trylock*** *(pthread_mutex_t *mut);*
  – falls Mutex frei, belege den Mutex, andernfalls gebe Wert *EBUSY* zurück

PThreads: Mutex

```
1  #include <pthread.h>
2
3  /* define Mutex-variable */
4  pthread_mutex_t      mutex_sum =
5              PTHREAD_MUTEX_INITIALIZER;
6  int                  sum = 100;
7
8  void *minus (void *arg) {
9    int i = 0;
10
11   for(i=0; i < 1000000; i++) {
12     pthread_mutex_lock (&mutex_sum);
13     random();
14     sum = sum - 1;
15     pthread_mutex_unlock (&mutex_sum);
16   }
17 }
18
19 void *plus (void *arg) {
20   int i = 0;
21
22   for(i=0; i < 1000000; i++) {
23     pthread_mutex_lock (&mutex_sum);
24     random();
25     sum = sum + 1;
26     pthread_mutex_unlock (&mutex_sum);
27   }
28 }
```

```
29 int main() {
30    // Definition der Variablen
31    pthread_t        thread[2];
32    pthread_attr_t   attr;
33    int              i, status, err;
34
35    pthread_attr_init (&attr);
36    pthread_attr_setdetachstate (&attr,
37          PTHREAD_CREATE_JOINABLE);
38
39    /* start threads */
40    pthread_create (&thread[0], &attr, minus,
41                                  (void *) NULL);
42    pthread_create (&thread[1], &attr, plus,
43                                  (void *) NULL);
44    /* Warte auf beide Threads */
45    for(i=0; i < 2; i++) {
46      err = pthread_join (thread[i], (void **)&status);
47      if (err) printf("No join...\n");
48    }
49    printf("Summe : %d\n",sum);
50 }
```

Conditions

Mutex synchronisiert den Zugriff von Threads auf Daten, aber wie können Threads basierend auf den aktuellen Werten der Daten synchronisiert werden?

- Beispiel: Status-Variable (an/aus, voll/leer)

Einfache Lösung:

- kontinuierliche Abfrage einer Variable (Polling)
- Polling verschwendet aber Prozessorressourcen!

Besserer Ansatz:

- über eine Kondition gesteuerter Zugriff auf einen Mutex
- Thread blockiert bis Kondition erfüllt ist

Lösung: Condition-Variable

Condition Variable (1)

- Initialisierung
  int ***pthread_cond_init*** *(pthread_cond_t *cond, pthread_condattr_t *attr);*

- Destruktion
  int ***pthread_cond_destroy*** *(pthread_cond_t *cond);*

- Warten
  int ***pthread_cond_wait*** *(pthread_cond_t *cond, pthread_mutex_t *mut);*
  - blockierendes Warten mit Freigabe des Mutex

- Signalisierung
  int ***pthread_cond_signal*** *(pthread_cond_t *cond);*
  - mindestens einen Thread aufwecken und nachfolgende Mutex-Aquirierung

Condition Variable (2)

- Broadcast Signalisierung
  int ***pthread_cond_broadcast*** *(pthread_cond_t *cond);*
  – alle Threads aufwecken mit nachfolgender Mutex-Aquirierung

- Warten mit Time-Out
  int ***pthread_cond_timedwait*** *(pthread_cond_t *cond,*
  *pthread_mutex_t *mut,*
  *const struct timespec *abstime);*
  – identisch zu *pthread_cond_wait()*, mit Zeitschranke
  – Rückgabe von *ETIMEDOUT* falls Zeitschranke überschritten
  struct timespec to { time_t tv_sec; long tv_nsec; };

- Erzeugen / Entfernen
  int ***pthread_cond_init*** *(pthread_cond_t *cond, pthread_condattr_t *attr);*
  – oder statisch, *phtread_cond_t cond =* ***PTHREAD_COND_INITIALIZER****;*
  int ***pthread_cond_destroy*** *(pthread_cond_t *cond);*

Condition: Anmerkungen

- eine Condition-Variable ist immer in Verbindung mit einem Mutex zu nutzen
- entsprechender Mutex muss vor Aufruf von *pthread_cond_wait()* belegt sein
- Konditionen werden signalisiert
- Warten mehrere Threads auf eine Kondition, dann muss *pthread_condition_broadcast()* anstatt p*thread_condition_signal()* genutzt werden
- Konditionen werden nicht „gespeichert“, d.h. *pthread_cond_wait()* **muss vor** dem entsprechenden p*thread_condition_signal()* aufgerufen werden

Beispiel: Condition Variable (1)

```
1  #include <pthread.h>
2  #define NUM_THREADS 3
3
4  pthread_mutex_t mutex;
5  pthread_cond_t  cond;
6
7  /* Eine Kondition für jeden Thread */
8  /*     0 : starten                  */
9  /*     1 : stoppen                  */
10 int flag[NUM_THREADS];
11
12 void *work (void *arg) {
13   int id = (int) arg;
14   int sum, i;
15
16   /* Mutex Variable sperren */
17   pthread_mutex_lock (&mutex);
18
19 /* In einer Schleife wird jetzt geprüft, */
20 /* ob der Thread laufen darf             */
21   while (flag[id]) {
22     printf("(thread %d) Muss noch warten...\n", id);
23     pthread_cond_wait (&cond, &mutex);
24   }
25
26   pthread_mutex_unlock (&mutex);
27
28   printf("(thread %d) Jetzt geht's los!\n", id);
29
30 /* Hier würde im Normalfall die eigentliche       */
31 /* Threadfunktion stehen; eine Schleife tut's auch. */
32   for(i=0; i < 10000; i++) {
33     sum = sum + (int) (100*random());
34   }
35
36   pthread_exit (NULL);
}
```

mutex koordiniert den Zugriff auf Variable *flag*

main Funktion auf der folgenden Folie

Beispiel: Condition Variable(2)

```
37 int main() {
38   /* Definition der Variablen */
39   pthread_t        thread[NUM_THREADS];
40   pthread_attr_t attr;
41
42   int i, status, err;
43
44    for(i=0; i < NUM_THREADS; i++)
45      flag[i] = 1;          /* Status „Thread-Stop“ */
46
47    pthread_attr_init (&attr);
48    pthread_attr_setdetachstate (&attr,
49          PTHREAD_CREATE_JOINABLE);
50
51   pthread_mutex_init (&mutex, NULL);
52   pthread_cond_init (&cond, NULL);
53
54   for(i=0; i < NUM_THREADS; i++) {
55         pthread_create (&thread[i], &attr,
56             work, (void *) i);
57   }
58 /* Alle Threads sind hochgefahren und warten */
59   printf("(main) Sende das Signal\n");
60 /* Sende das Signal, dass die Threads starten können */
61   pthread_mutex_lock (&mutex);
62   for(i=0; i < NUM_THREADS; i++)
63         flag[i] = 0;
64
65 /* Wecke sie auf */
66   pthread_cond_broadcast (&cond);
67   pthread_mutex_unlock (&mutex);
68
69   /* Warte auf alle Threads */
70   for(i=0; i < NUM_THREADS; i++) {
71     err = pthread_join (thread[i], (void **)&status);
72       if (err) printf("(main) ERROR - No join...\n");
73   }
74
75 /* Aufräumen */
76   pthread_attr_destroy (&attr);
77   pthread_mutex_destroy (&mutex);
78   pthread_cond_destroy (&cond);
79
80   pthread_exit (NULL);
81 }
```

Condition Variable

Achtung

- Signale werden nicht gespeichert
  - Falls kein Thread im *Condition-Wait*, dann gilt ein zu dem Zeitpunkt gesendetes Signal als verloren
  - Threads können somit verhungern, falls auf ein Signal gewartet wird das nicht mehr gesendet wird
- Lock und Unlock müssen richtig gesetzt werden
  - ansonsten eine Verklemmung im Programm möglich

Wechselseitiger Ausschluss und Synchronisation

- Mutex
  - Das Mutex wird verwendet für den wechselseitigen Ausschluss.
  - Der Konstruktor von *mutex* initialisiert das binäre Semaphor automatisch auf *unlocked*.
  - Mit *lock* versucht man in den kritischen Abschnitt einzutreten, mit *unlock* verlässt man diesen.
- Condition Variable
  - Eine Condition Variable ist immer mit mindestens einem Mutex verbunden.
  - Die Wait Operation gibt das Mutex frei und blockiert die Aktivität. Die Wait Operation kann nicht unterbrochen werden.
  - Die Signal Operation hat nur eine Auswirkung, falls mindestens eine Aktivität blockiert ist: in diesem Falle wird mindestens eine Aktivität aufgeweckt.
  - Die Broadcast Operation funktioniert wie Signal außer, dass alle blockierten Aktivitäten aufgeweckt werden.
  - Sobald eine auf Wait blockierte Aktivität aufgeweckt wird, so wird von dieser erneut die Lock Operation auf dem Mutex ausgeführt und fährt anschließend mit seiner Arbeit fort.

Was ist OpenMP?

OpenMP (*Open Multi Processing*) ist

- ein Standard für die Realisierung des Shared-Memory Programmiermodells
- compilerbasiert
- eine Erweiterung bestehender Programmiersprachen durch
  - Direktiven für den Compiler
  - einige wenige Bibliotheksroutinen
  - Umgebungsvariablen (z.B. Anzahl Prozesse, Umgebungsvariablen)

Möglichkeiten von OpenMP

OpenMP bietet

- Portabilität auf Shared-Memory Architekturen
- Skalierbarkeit
- inkrementelle Parallelisierung
- Unterstützt der Daten-Parallelisierung
- Erweiterungen für FORTRAN und C/C++

Geschichte von OpenMP

OpenMP

- entwickelte sich aus Erweiterungen bestehender Programmiersprachen durch einzelne Hersteller
- wurde mit dem Ziel der Portabilität entwickelt
- ist ein Standard sein 1997
- wurde von Cray, SGI, Digital Equipment Corporation, IBM, Intel usw. entwickelt und wird von vielen (allen) Hard- und Softwarefirmen unterstützt.

Quellen zu OpenMP

- R. Chandra et al.: Parallel Programming in OpenMP, Academic Press 2001
- Chapman, B.; Jost, G.; van der Pas, R.: Using OpenMP – Portable Shared Memory Parallel Programming, 2007
- http://www.openmp.org
- Intel Compiler (Linux non commercial use)
  - http://www.intel.com/software/products/global/eval.htm
  - icc –openmp file.c # Linux x86
  - ecc –openmp file.c # Linux x86-64
  - icl /Qopenmp file.c # Windows
- GCC Compiler ab Version 4.2
  - gcc –fopenmp file.c

OpenMP Programming Modell

- OpenMP basiert auf dem Shared-Memory Modell
- Die Arbeitslast wird zwischen Threads verteilt
  - Variablen können
    - gemeinsam (shared) für alle Threads sein
    - für jeden Thread dupliziert werden (private Nutzung)
  - Threads kommunizieren durch gemeinsame Variablen
- Unbeabsichtigt gemeinsam genutzte Variablen können zu sogenannten *race conditions* führen:
  - *race condition*: Das Ergebnis des Programms ändert sich, wenn sich das Laufzeitverhalten (scheduling) der Threads ändert
- *Kontrolle von „race conditions“* durch
  - Synchronisierung, um Datenkonflikte zu vermeiden
- Nachlässige Verwendung von Synchronisierungsanweisungen kann zu *deadlocks* führen

Umgebungsvariablen

- *OMP_NUM_THREADS*
  - gibt die Anzahl der Threads während des Programmlaufs an
  - ist die dynamische Anpassung der Anzahl der Threads aktiviert, so gibt sie die Zahl maximal verfügbarer Threads an
  - *setenv OMP_NUM_THREADS 4*

- *OMP_SCHEDULE*
  - betrifft nur die for-Schleife und parallelen *for* Direktiven die den Scheduling Type *RUNTIME* haben
  - setzt den Scheduling Typ und die chunk size aller Schleifen
  - *setenv OMP_SCHEDULE “GUIDED, 4“*

OpenMP Direktiven

- *#pragma* Direktiven
- Format:
  #pragma omp directive_name [clause [[,] [clause], ..]
- Unterscheidung von Gross- und Kleinbuchstaben
- bedingte Compilation
  #ifdef ***_OPENMP***
  block
  #endif
- Include-File für Bibliotheksroutinen:
  #include <***omp.h***>

OpenMP Laufzeitbibliothek

- Benutzerfunktionen der Laufzeitbibliothek
  - Anzahl der Threads
    numthreads = ***omp_get_num_threads****();*
  - Identifikation eines Threads
    myid = ***omp_get_thread_num****();*
- Systemfunktionen der Laufzeitbibliothek
  - Thread-Verwaltung
    - starten, halten, stoppen, ...
  - Thread-Steuerung
    - Scheduling, ...
  - ...

Parallele Region

- Block des Programms, der durch mehrere Threads parallel ausgeführt werden soll. Jeder Thread führt denselben Code aus
- C/C++:
  #pragma **omp parallel** [clause [clause] ..] *new line*
  structured block
- *Clause kann eine der folgenden sein:*
  - *private (list)*
  - *shared (list)*
  - *reduction (operator,list)* - *Reduktion auf den* Variablen
  - *schedule (type [, chunk])* - Aufteilung der Iterationen auf Threads
  - *nowait* - Thread nicht am Schleifenende synch

Fork-and-Join-Modell (1)

- *Fork-Join* Modell dient der parallelen Ausführung
- Begin der Ausführung mit einem einzigen Prozess (*master process*)
- paralleles Konstrukt geht aus Master Prozess hervor
  - Start:
    Master Thread erzeugt Team von *Threads*
  - Abschluss:
    Die Threads des Teams werden synchronisiert (implizite Barriere)
- Nur der Master Thread fährt mit der Ausführung fort

Beispiel: OpenMP „hello“

```
#include <stdio.h>

# ifdef _OPENMP
#  include <omp.h>
# endif

int main(int argc, char** argv)
{
  int myid, numthreads;

   myid = -1; numthreads = 0;
#pragma omp parallel private(myid)
  {
# ifdef _OPENMP

    myid = omp_get_thread_num();
    numthreads = omp_get_num_threads();
# endif
    printf("hello, world says thread %d of %d threads.\n", myid, numthreads);

    /* end omp parallel */
  }
  return 0;
}
```


Data Scope Clauses

- *private* (list)

  erklärt die Variablen in der Liste list zu privaten Variablen der einzelnen Threads des Teams

- *shared* (list)

  erklärt die Variablen in der Liste list zu gemeinsamen Variablen aller Threads des Teams

- ohne explizite Vereinbarung: Default shared aber
  - Stack- (lokale) Variablen in einem aufgerufenen Unterprogramm sind *private*
  - die Schleifenvariable der parallelen *for*-Schleife ist immer *private*

OpenMP-Beispiel: Berechnung von Pi

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <omp.h>
5
6  #define PI 3.14159265358979323L
7
8  double f (double a)
9  {
10   return (double)4.0/((double)1.0+(a*a));
11 }
12
13 int main(int argc, char *argv[])
14 {
15   long  n, i;
16   double   h, pi, sum, x;
17
18   for (;;) {
19     printf("Enter the number of intervals: (0 quits) ");
20     scanf("%u",&n);
21     if (n==0)
22       break;
23     h = ((double)1.0)/(double)n;
24     sum = 0.0;
25
26 #pragma omp parallel for private(i,x) reduction(+:sum)
27     for(i=1;i<=n;i++) {                    /* parallel loop */
28       x = h*((double)i-(double)0.5);
29       sum += f(x);
30     }
31                                            /* end omp parallel for */
32     pi = h*sum;
33     printf("pi is approximatly: %.16f  Error is: %.16f\n",
34            pi, fabs(pi-PI));
35   }
36
37   return EXIT_SUCCESS;
38 }
```


Weitere Direktiven

- *critical* Direktive
  - Beschränkung des Zugriffs auf den eingeschlossenen Code auf nur einen Thread zur selben Zeit

    #pragma omp ***critical*** *[(name)]*
    structured block
  - Jeder Thread wartet am Anfang einer *critical*-Region bis kein anderer Thread des Teams die Region dieses Namens ausführt. Alle unbenannten Regionen *critical*-Anweisungen zeigen auf den selben unspezifizierten Namen

- *parallel for* Direktive
  - Parallele Region mit nur einer einzigen parallelen *for*-Schleife

    #pragma omp ***parallel for*** *[clause [[,] clause] …]*
    for loop

Weitere Direktiven

- *single* Direktive
  - Ausführung eines Teils einer parallelen Region durch nur einen zwar den schnellsten Thread:

    #pragma omp ***single***

    structured block

- *barrier* Direktive
  - Synchronisation innerhalb einer parallelen Region

  #pragma omp ***barrier***

- *sections* Direktive
  - Synchronisation innerhalb einer parallelen Region
  - Verteilt unabhängige Programmteile auf Threads.

  #pragma omp ***sections*** *[clause [[,] clause] …]*

  {

  structured block 1

  structured block 2

Parallele Bibliotheken: Beispiele

- Intel-Bibliotheken:
  - Math Kernel Library
    - mathematische Funktionen der Linearen Algebra
    - über OpenMP parallelisierte Routinen
    - Funktionen: Matrix-Operationen, Löser für Gleichungssysteme, FFTs, …
  - Threading Building Blocks
    - Konstrukte: parallel_for, parallel_do, parallel_reduce, pipeline, parallel_sort, parallel_scan
- NAG-Bibliothek:
  - Differentialgleichungen, Optimierung, Lineare Algebra, statistische Methoden, …