

# Paralleles Höchstleistungsrechnen



## Parallele Programmiermodelle I

Stefan Lang

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen  
Universität Heidelberg  
INF 368, Raum 425  
D-69120 Heidelberg  
phone: 06221/54-8264  
email: [Stefan.Lang@iwr.uni-heidelberg.de](mailto:Stefan.Lang@iwr.uni-heidelberg.de)

28. Oktober 2009



# Parallele Programmiermodelle I

## Kommunikation über gemeinsamen Speicher

- Kritischer Abschnitt
- Wechselseitiger Ausschluss: Petersons Algorithmus
- OpenMP
- Barriere – Synchronisation aller Prozesse
- Semaphore



# Kritischer Abschnitt

Was ist ein kritischer Abschnitt?

Wir betrachten folgende Situation:

- Anwendung besteht aus  $P$  nebenläufigen Prozessen, diese werden also zeitgleich bearbeitet
  - von einem Prozeß ausgeführte Anweisungen werden in zusammenhängende Gruppen eingeteilt
    - ▶ kritische Abschnitte
    - ▶ unkritische Abschnitte
  - kritischer Abschnitt: Folge von Anweisungen, welche lesend oder schreibend auf *gemeinsame Variable* zugreift.
  - Anweisungen eines kritischen Abschnitts dürfen nicht gleichzeitig von zwei oder mehr Prozessen bearbeitet werden
- man sagt nur ein Prozeß darf sich im kritischen Abschnitt befinden



# Wechselseitiger Ausschluß

2 Arten der Synchronisation sind unterscheidbar

- Bedingungssynchronisation
- Wechselseitiger Ausschluß

Wechselseitiger Ausschluß besteht aus *Eingangsprotokoll* und *Ausgangsprotokoll*:

Programm (Einführung wechselseitiger Ausschluß)

**parallel** *critical-section*

```
{
  process  $\Pi$  [int  $p \in \{0, \dots, P - 1\}$ ]
  {
    while (1)
    {
      Eingangsprotokoll;
      kritischer Abschnitt;
      Ausgangsprotokoll;
      unkritischer Abschnitt;
    }
  }
}
```

# Wechselseitiger Ausschluß

Folgende Kriterien sollen erfüllt werden:

- 1 *Wechselseitiger Ausschluss.* Höchstens ein Prozess führt den kritischen Abschnitt zu einer Zeit aus.
- 2 *Verklemmungsfreiheit.* Falls zwei oder mehr Prozesse versuchen in den kritischen Abschnitt einzutreten muss es genau einer nach endlicher Zeit auch schaffen.
- 3 *Keine unnötige Verzögerung.* Will ein Prozess in den kritischen Abschnitt eintreten während alle anderen ihre unkritischen Abschnitte bearbeiten so darf er nicht am Eintreten gehindert werden.
- 4 *Schließliches Eintreten.* Will ein Prozess in den kritischen Abschnitt eintreten so muss er dies nach endlicher Wartezeit auch dürfen (dazu nimmt man an, daß jeder den kritischen Abschnitt auch wieder verlässt).



# Petersons Algorithmus: Eine Softwarelösung

Wir betrachten zunächst nur zwei Prozesse und entwickeln die Lösung schrittweise ...

Erster Versuch: Warte bis der andere *nicht* drin ist

```
int in1=0, in2=0;    // 1=drin
```

$\Pi_1$ :	$\Pi_2$ :
<b>while</b> ( <i>in2</i> ) ;	<b>while</b> ( <i>in1</i> ) ;
<i>in1</i> =1;	<i>in2</i> =1;
krit. Abschnitt;	krit. Abschnitt;

- Es werden keine Maschinenbefehle benötigt
- Problem: Lesen und Schreiben ist nicht atomar



# Petersons Algorithmus: Zweite Variante

Erst besetzen, dann testen

```
int in1=0, in2=0;
```

$\Pi_1$ :

```
in1=1;
```

```
while (in2) ;
```

```
krit. Abschnitt;
```

$\Pi_2$ :

```
in2=1;
```

```
while (in1) ;
```

```
krit. Abschnitt;
```

Problem: Verklemmung möglich



# Petersons Algorithmus: Dritte Variante

Löse Verklemmung durch Auswahl eines Prozesses

Programm (Petersons Algorithmus für zwei Prozesse)

**parallel** *Peterson-2*

{

int *in1=0, in2=0, last=1;*

**process**  $\Pi_1$

{

**while** (1) {

*in1=1;*

*last=1;*

**while** (*in2*  $\wedge$  *last==1*);

*krit. Abschnitt;*

*in1=0;*

*unkrit. Abschnitt;*

}

}

}

**process**  $\Pi_2$

{

**while** (1) {

*in2=1;*

*last=2;*

**while** (*in1*  $\wedge$  *last==2*);

*krit. Abschnitt;*

*in2=0;*

*unkrit. Abschnitt;*

}

}



# Konsistenzmodelle

Bisherige Beispiele basieren auf dem Prinzip der *Sequentiellen Konsistenz*:

- 1 *Lese- und Schreiboperationen werden in Programmordnung beendet*
- 2 *diese Reihenfolge ist für alle Prozessoren konsistent sichtbar*

Hier erwartet man, dass  $a = 1$  gedruckt wird:

```
int a = 0, flag=0;           // wichtig!  
process  $\Pi_1$                 process  $\Pi_2$   
    ...                       ...  
    a = 1;                     while (flag==0) ;  
    flag=1;                     print a ;
```

Hier erwartet man, dass nur für einen die **if**-Bedingung wahr wird:

```
int a = 0, b = 0;           // wichtig!  
process  $\Pi_1$                 process  $\Pi_2$   
    ...                       ...  
    a = 1;                     b = 1;  
    if (b == 0) ...           if (a == 0) ...
```



# Konsistenzmodelle

Warum keine sequentielle Konsistenz?

- *Umordnen von Anweisungen*: Optimierende Compiler können Operationen aus Effizienzgründen umordnen. Das erste Beispiel funktioniert dann nicht mehr!
- *Out-of-order execution*: z. B. Lesezugriffe sollen langsame Schreibzugriffe (invalidate) überholen können (solange es nicht die selbe Speicherstelle ist). Das zweite Beispiel funktioniert dann nicht mehr!

*Total store ordering*: Lesezugriff darf Schreibzugriff überholen

*Weak consistency*: Alle Zugriffe dürfen einander überholen

Reihenfolge kann durch spezielle Maschinenbefehle erzwungen werden, z. B. *fence*: alle Speicherzugriffe beenden bevor ein neuer gestartet wird.

Diese Operationen werden eingefügt,

- bei annotieren von Variablen („Synchronisationsvariable“)
- bei parallelen Anweisungen (z. B. FORALL in HPF)
- vom Programmierer der Synchronisationsprimitive (z. B. Semaphore)



# Peterson für $P$ Prozesse

Idee: Jeder durchläuft  $P - 1$  Stufen, Jeweils der letzte auf einer Stufe ankommende muss warten

Variablen:

- $in[i]$ : Stufe  $\in \{1, \dots, P - 1\}$  (!), die  $\Pi_i$  erreicht hat
- $last[j]$ : Nummer des Prozesses der zuletzt auf Stufe  $j$  ankam

Programm (Petersons Algorithmus für  $P$  Prozesse)

```
parallel Peterson-P
{
  const int P=8;
  int in[P] = {0[P]};
  int last[P] = {0[P]};
  ...
}
```



# Peterson für $P$ Prozesse

## Programm (Petersons Algorithmus für $P$ Prozesse cont.)

parallel Peterson-P cont.

```
{
  process  $\Pi$  [int  $i \in \{0, \dots, P - 1\}$ ]
  {
    int  $j, k$ ;
    while (1)
    {
      for ( $j=1; j \leq P - 1; j++$ )           // durchlaufe Stufen
      {
         $in[i] = j$ ;                          // ich bin auf Stufe  $j$ 
         $last[j] = i$ ;                        // ich bin der letzte auf Stufe  $j$ 
        for ( $k = 0; k < P; k++$ )           // teste alle anderen
          if ( $k \neq i$ )
            while ( $in[k] \geq in[i] \wedge last[j] == i$ );
      }
      kritischer Abschnitt;
       $in[i] = 0$ ;                            // Ausgangsprotokoll
      unkritischer Abschnitt;
    }
  }
}
```

- $O(P^2)$  Tests notwendig für Eintritt
- Strategie ist fair, wer als erstes kommt darf als erster rein



# Hardware Locks

Hardwareoperationen zur Realisierung des wechselseitigen Ausschlusses:

- *test-and-set*: Überprüfe ob eine Speicherstelle den Wert 0 hat, wenn ja schreibe den Inhalt eines Registers in die Speicherstelle (als unteilbare Operation).
- *fetch-and-increment*: Hole den Inhalt einer Speicherstelle in ein Register und erhöhe den Inhalt der Speicherstelle um eins (als unteilbare Operation).
- *atomic-swap*: Vertausche den Inhalt eines Registers mit dem Inhalt einer Speicherstelle in einer unteilbaren Operation.

In allen Maschinenbefehlen muß ein Lesezugriff gefolgt von einem Schreibzugriff ohne Unterbrechung durchgeführt werden!



# Hardware Locks

Ziel: Maschinenbefehl und Cache-Kohärenz Modell stellt alleiniges Eintreten in den kritischen Abschnitt sicher und erzeugt geringen Verkehr auf dem Verbindungsnetzwerk

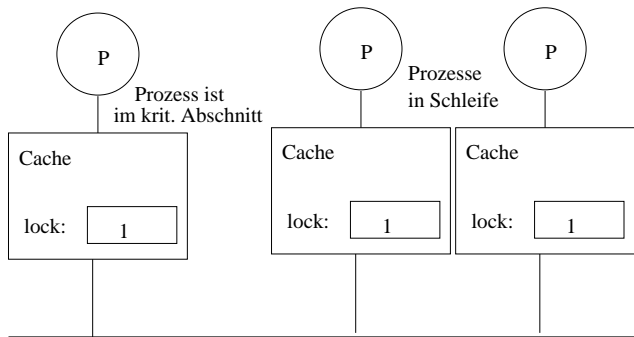
## Programm (Spin Lock)

```
parallel spin-lock
{
    const int P = 8;           // Anzahl Prozesse
    int lock=0;                // Variable zur Absicherung

    process  $\Pi$  [int  $p \in \{0, \dots, P - 1\}$ ]
    {
        ...
        while ( atomic --swap(& lock)) ;
        ...                    // kritischer Abschnitt
        lock = 0;
        ...
    }
}
```

# Hardware Locks

Was passiert im System?



Die beiden wartenden Prozesse erzeugen hohen Busverkehr



# Hardware Locks

Ablauf bei MESI Protokoll: Variable *lock* ist 0 und ist in keinem der Caches

- Prozess  $\Pi_0$  führt die *atomic – swap*-Operation aus
- Lesezugriff führt zu read miss, Block wird aus dem Speicher geholt und bekommt den Zustand E (wir legen MESI zugrunde).
- Nachfolgendes Schreiben ohne weiteren Buszugriff, Zustand von E nach M.
- anderer Prozess  $\Pi_1$  führt *atomic – swap*-Operation aus
- Read miss führt zum Zurückschreiben des Blockes durch  $\Pi_0$ , der Zustand beider Kopien ist nun, nach dem Lesezugriff, S.
- Schreibzugriff von  $\Pi_1$  invalidiert Kopie von  $\Pi_0$  und Zustand in  $\Pi_1$  ist M.
- *atomic – swap* gibt 1 in  $\Pi_1$  und kritischer Abschnitt wird von  $\Pi_1$  nicht betreten.
- Führen beide Prozesse die *atomic – swap*-Operation gleichzeitig aus entscheidet letztendlich der Bus wer gewinnt.
- Angenommen Cache  $C_0$  von Prozessor  $P_0$  als auch  $C_1$  haben je eine Kopie des Cache-Blockes im Zustand S vor Ausführung der *atomic – swap*-Operation
- Beide lesen zunächst den Wert 0 aus der Variable *lock*.
- Beim nachfolgenden Schreibzugriff konkurrieren beide um den Bus um eine invalide Meldung zu platzieren.
- Der Gewinner setzt seine Kopie in Zustand M, Verlierer seine in den Zustand I. Die Cache-Logik des Verlierers findet beim Schreiben den Zustand I vor und muss den *atomic – swap*-Befehl veranlassen doch den Wert 1 zurückzuliefern (der *atomic-swap*-Befehl ist zu dieser Zeit ja noch nicht beendet).





# Verbessertes Lock

Idee: Mache keinen Schreibzugriff solange der kritische Abschnitt besetzt ist

## Programm (Verbesserter Spin Lock)

```
parallel improved-spin-lock
```

```
{
```

```
    const int P = 8;           // Anzahl Prozesse
```

```
    int lock=0;              // Variable zur Absicherung
```

```
    process  $\Pi$  [int p  $\in$  {0, ..., P - 1}] {
```

```
        ...
```

```
        while (1)
```

```
            if (lock==0)
```

```
                if (read - and - set(& lock)==0 )
```

```
                    break;
```

```
        ...
```

```
        lock = 0;
```

```
        ...
```

```
    }
```

```
}
```

```
        // kritischer Abschnitt
```

# Verbessertes Lock

- 1 Problem: Strategie gewährleistet keine Fairness
- 2 Situation mit drei Prozessen: Zwei wechseln sich immer ab, der dritte kommt nie rein
- 3 Aufwand falls  $P$  Prozesse gleichzeitig eintreten wollen:  $O(P^2)$ , Zuweisung  $lock = 0$  bedingt  $P$  Bustransaktionen für Cache Block Kopien
- 4 Lösung ist ein Queuing Lock: Bei Austritt aus dem kritischen Abschnitt wählt der Prozess seinen Nachfolger aus

Ticketalgorithmus:

- Fairness mit Hardware-Lock
- Idee: Vor dem Anstellen an der Schlange zieht man eine Nummer. Der mit der kleinsten Nummer kommt als nächster dran.



# Ticketalgorithmus

Programm (Ticket Algorithmus für  $P$  Prozesse)

**parallel** *Ticket*

```
{  
  const int  $P=8$ ;  
  int  $number=0$ ;  
  int  $next=0$ ;  
  
  process  $\Pi$  [int  $i \in \{0, \dots, P - 1\}$ ]  
  {  
    int  $mynumber$ ;  
    while (1)  
    {  
      [ $mynumber=number$ ;  $number=number+1$ ];  
      while ( $mynumber \neq next$ ) ;  
      kritischer Abschnitt;  
       $next = next+1$ ;  
      unkritischer Abschnitt;  
    }  
  }  
}
```

# Ticketalgorithmus

- 1 Fairness basiert darauf, daß Ziehen der Zahl nicht lange dauert.  
Möglichkeit einer Kollision ist kurz.
- 2 Geht auch bei Überlauf der Zähler, ( $\text{MAXINT} > P$ )
- 3 Inkrementieren von *next* geht ohne Synchronisation, da dies immer nur einer machen kann



# Bedingter kritischer Abschnitt

- Erzeuger-Verbraucher Problem:
  - ▶  $m$  Prozesse  $P_i$  (Erzeuger) erzeugen Aufträge, die von  $n$  anderen Prozessen  $C_j$  (Verbraucher) abgearbeitet werden sollen.
  - ▶ Die Prozesse kommunizieren über eine zentrale Warteschlange (WS) mit  $k$  Plätzen.
  - ▶ Ist die WS voll müssen die Erzeuger warten, ist die WS leer müssen die Verbraucher warten.
- Problem: Wartende dürfen den (exklusiven) Zugriff auf die WS nicht blockieren!
- Kritischer Abschnitt (Manipulation der WS) darf nur Betreten werden *wenn* WS nicht voll (für Erzeuger), bzw. nicht leer (für Verbraucher) ist.
- Idee: Probeweises Betreten und busy-wait:



# Bedingter kritischer Abschnitt

Programm (Erzeuger–Verbraucher Problem mit aktivem Warten)

**parallel** *producer-consumer-busy-wait*

```
{  
  const int m = 8; n = 6; k = 10;  
  int orders=0;  
  process P [int 0 ≤ i < m]  
  {  
    while (1) {  
      produziere Auftrag;  
      CSenter;  
      while (orders==k){  
        CSexit;  
        CSenter;  
      }  
      speichere Auftrag;  
      orders=orders+1;  
      CSexit;  
    }  
  }  
}
```

```
process C [int 0 ≤ j < n]  
{  
  while (1) {  
    CSenter;  
    while (orders==0){  
      CSexit;  
      CSenter;  
    }  
    lese Auftrag;  
    orders=orders-1;  
    CSexit;  
    bearbeite Auftrag;  
  }  
}
```

# Bedingter kritischer Abschnitt

- Ständiges Betreten und Verlassen des kritischen Abschnittes ist ineffizient wenn mehrere Warten (Trick vom verbesserten Lock hilft nicht)
- (Praktische) Abhilfe: Zufällige Verzögerung zwischen `CSenter/CSexit`, *exponential back-off*.



# OpenMP (Open Multi Processing) I: Ansatz

OpenMP ist ein paralleles Programmiermodell auf der Grundlage der folgenden Annahmen:

- ein Prozeß besteht aus mehreren Threads (Fäden)
- alle Threads teilen die gleichen Statusvariablen des Programms
- jeder Thread kann zusätzliche private Variablen besitzen
- Threads können auf unterschiedlichen Prozessoren laufen
- Es gibt Mechanismen zur Synchronization und zum Sperren





# OpenMP II

## Was ist OpenMP?

- API (application programming interface) um multi-threaded Anwendungen zu schreiben
  - ▶ Compilerdirektiven and Bibliotheksfunktionen
  - ▶ standardisiert für C und Fortran
- neuer Standard oder wenigstens ein Standardisierungsversuch
- wichtiges Modell, da beinahe viele wichtige Firmen partizipieren
- Parallelisierungsprozeß unter OpenMP
  - ▶ Programm wird in Stufen parallelisiert
  - ▶ Startpunkt ist die serielle Version, welche in vielen Fällen unverändert erhalten bleibt
  - ▶ Parallelismus ist nicht direkt codiert, sondern wird mit Direktiven beeinflusst



# OpenMP Beispiel I

Hello World:

- das ursprüngliche Programm bleibt erhalten
- Ausführung beim Setzen einer Umgebungsvariable:

```
void main(void)
{
#pragma omp parallel
  {
    printf(„Hallo Welt\n“);
  }
}
```

```
(~): export OMP_NUM_THREADS=4
(~): ./hello-openmp
Hallo Welt
Hallo Welt
Hallo Welt
Hallo Welt
```



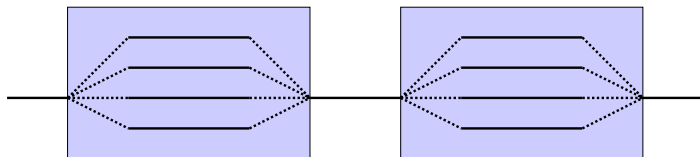
# OpenMP Beispiel II

## Parallele Regionen (Blöcke)

- bei Verwendung der Compilerdirektive `#pragma omp parallel` wird der folgende Block parallel ausgeführt
- dazu wird eine Menge von Threads gestartet
  - ▶ die Threadanzahl hängt von der Umgebungsvariable `OMP_NUM_THREADS` ab, die kann vom Programm geändert werden
  - ▶ man spricht von fork-join Parallelismus
- nachdem alle Threads abgearbeitet sind, werden diese terminiert oder verbleiben wartend

Kontrollfluß in Block 1

Kontrollfluß in Block 2



# OpenMP Beispiel III

- primärer Einsatzbereich von OpenMP ist das Parallelisieren von Schleifen
- **#pragma omp parallel for**
- i-loop will be executed simultaneously by `OMP_NUM_THREADS` threads

Matrix-Matrix Multiplikation

```
#pragma omp parallel for
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < K; k++)
            C[i,j] = A[i,k] * B[k,j];
```



# OpenMP Beispiel IV

## Laufzeitbedingungen

- falls mehrere Threads die selbe Variable lesen und schreiben, können Inkonsistenzen entstehen
- dies ist mit den bekannten Inkonsistenzen in Shared-Memory Architekturen vergleichbar

## Beispiel: Skalarprodukt

```
sum = 0.0;
#pragma omp parallel for
for (i = 0; i < N; i++)
    sum = sum + x[i] * y[i];
```



# OpenMP Beispiel V

## Lösung 1: Locking

- Um die Addition bei Summenbildung abzusichern kann man diese als atomar oder kritisch deklarieren
- Nachteil: das ist ineffizient, weil die Threads nicht mehr parallel arbeiten können

```
sum = 0.0;
#pragma omp parallel for
for (i = 0; i < N; i++)
#pragma omp critical
{
    sum = sum + x[i] * y[i];
}
```



# OpenMP Beispiel VI

## Lösung 2: Privat Variablen

- in parallelen Regionen können bestimmte Variablen als privat deklariert werden
- dies kann man noch kompakter schreiben

```
sum = 0.0;
#pragma omp parallel private(local_sum)
{
    local_sum = 0.0;
#pragma omp parallel for
    for (i = 0; i < N; i++)
        local_sum = local_sum + x[i] * y[i];

#pragma omp critical
    { sum = sum + local_sum; }
}
```



# OpenMP Beispiel VII

## Lösung 3: Reduktionsvariablen

- solche Fälle sind typisch, man kann kritische Variable als Reduktionsvariable deklarieren
- innerhalb von Threads werden diese als privat generiert und dann mit einer geeigneten Operation am Schleifenende verbunden (synchronisiert)

```
sum = 0.0;
#pragma omp parallel for reduction (+ : sum)
for (i = 0; i < N; i++)
    sum = sum + x[i] * y[i];
```





# OpenMP Pragmas I

- directives (in C):
  - ▶ **#pragma omp clauses** ...
  - ▶ are ignored by non-OpenMP compilers
- parallel regions (blocks):
  - ▶ **#pragma omp parallel**
  - ▶ following block ({...}) is executed in parallel
- variable scoping
  - ▶ **#pragma omp private(...) shared(...) reduction(...) firstprivate(...)**  
**lastprivate(...)**
  - ▶ defines which variables are used together and which are used as copies in each thread
  - ▶ **shared is default value**



# OpenMP Pragmas II

- synchronization
  - ▶ **#pragma omp atomic, critical, ordered, barrier, flush**
  - ▶ essential for program correctness
- parallel loops (work-sharing)
  - ▶ **#pragma omp parallel for**
  - ▶ following **for** is parallelized
  - ▶ type of distribution can be determined with **schedule** clause
  - ▶ e.g. **schedule(dynamic,4)**: each thread is assigned four loop iterations (1..4, 5..8) and new ones, as soon as a thread is ready
  - ▶ other variants are **static, guided** and **runtime**



# OpenMP Run Time Environment I

## run time environment

- processor count
  - ▶ **omp\_get\_num\_procs()**
- thread count
  - ▶ **omp\_set\_num\_thread(int)**
  - ▶ **omp\_get\_num\_thread()**  
same as environment variable **OMP\_NUM\_THREADS**
  - ▶ **omp\_get\_thread\_num()**



# OpenMP Run Time Environment II

## run time environment

- dynamic mode: Is in different blocks a different number of threads allowed?
  - ▶ **omp\_set\_dynamic(), omp\_get\_dynamic()**
  - ▶ equal to **OMP\_DYNAMIC (TRUE / FALSE)**
- nesting: Are in parallel regions new thread teams allowed? (nested threads)
  - ▶ **omp\_set\_nested(), omp\_get\_nested()**
  - ▶ equal to **OMP\_NESTED (TRUE / FALSE)**

