Rechnerstrukturen

Vorlesung im Sommersemester 2006

Prof. Dr. Wolfgang Karl

Universität Karlsruhe (TH)

Fakultät für Informatik

Institut für Technische Informatik

- **Kapitel 3: Multiprozessoren – Parallelismus auf Prozess/Thread-Ebene**

3.5: Multiprozessoren mit gemeinsamem Speicher

- Shared Memory Programmierung – Überblick
  - Beispiel: lineare Rekursion R(n,n-1)

$$x_i = 0, i \leq 0,$$

$$x_i = c_i + \sum_{j=i-m}^{i-1} a_{ij} x_j, \quad 1 \leq i \leq n, \quad \text{mit } m = n - 1$$

  - Kann als implizite MMatrix-Vektor-Gleichung geschrieben werden:

$$x = c + A \cdot x,$$

wobei $a_{ij} = 0$ falls $i \leq j$ oder $i > j + m$

  - Gleichungssystem

$$x_1 = c_1$$
$$x_2 = c_2 + a_{21} x_1$$
$$x_3 = c_3 + a_{31} x_1 + a_{32} x_2$$

Multiprozessor mit gemeinsamem Speicher

- Shared Memory Programmierung – Überblick
  - Paralleles Programm (V.1)

```
procedure dorow (value i, done, n, a, x, c)
  shared n, a[n,n], x[n], c[n], done;
  private i,j;
  x[i]:=c[i];
  for j=1 step 1 until i-1
          x[i]:=x[i]+a[i,j]*x[j];
  done:=done-1;
  return;
end procedure

shared n, a[n,n], x[n], c[n], done;
private i;
done:=n;
for I:=1 step 1 until n-1
          create dorow (i,done,n,a,x,c);
i:=n;
call dorow (i,done,n,a,x,c)
while (done ≠ 0);
```

- Shared Memory Programmierung – Überblick
  - Paralleles Programm
    - Alle Prozesse müssen eine private Kopie der Variablen i zum Zeitpunkt des Aufrufs vom initialen Prozess erhalten
    - Alle Prozesse müssen zusammen müssen zusammen die j-Iteration bearbeiten, so dass der endgültige Wert eines Elements x[k] verfügbar ist, bevor dieses von einem Prozess zugegriffen wird, der die Werte von i>k hat
      - Synchronisationsproblem:
        - » Das Programm arbeitet nur korrekt, wenn ein Prozess i die j-te Iteration der Schleife über die Spalten ausführt und den Wert von x[j] liest, nachdem Prozess j die (j-1)-te Iteration beendet hat und damit die Berechnung von x[j] beendet hat.
        - » Explizite Synchronisation notwendig!

- Shared Memory Programmierung – Überblick
  - Synchronisation
    - Eine Synchronisationsoperation verzögert das Fortschreiten eines oder mehrerer Prozesse bis eine bestimmte Bedingung erfüllt ist.
    - Beispiel: Daten-basierte Synchronisation
      - Erzeuger/Verbraucher-Synchronisation
        - » Verbindet einen 1 Bit *full/empty*-Zustand mit jeder Variablen und verwendet synchronisierte Lese- und Schreiboperationen um auf den Wert zuzugreifen, nur dann, wenn dieser einen bestimmten Zustand hat.
        - » **`produce:`** wartet, bis die Variable im Zustand *empty* ist, schreibt den Wert und setzt den Zustand auf full.
        - » **`consume`**: wartet, bis die Variable im Zustand *full* ist, liest den Wert, und setzt den Zustand auf empty.
        - » **`void`**: initialisiert den Zustand auf empty
        - » **`copy`**: wartet auf full und liest, setzt aber nicht auf empty

Multiprozessor mit gemeinsamem Speicher

- Shared Memory Programmierung – Überblick
  - Paralleles Programm (V.2)

```
procedure dorow (value i, done, n, a, x, c)
  shared n, a[n,n], x[n], c[n], done;
  private i,j,sum,priv;
  sum = c[i];
  for j=1 step 1 until i-1
          {copy x[j] into priv; /*Get x[j] wenn verfügbar*/
          sum := sum + a[i,j]*priv;}
  produce x[i]:= sum; /*Make x[i] available to others*/
  done:=done-1;
  return;
end procedure

shared n, a[n,n], x[n], c[n], done;
private i;
done:=n;
for I:=1 step 1 until n-1
          {void x[i];
          create dorow (i,done,n,a,x,c);
i:=n;
void x[i];
call dorow (i,done,n,a,x,c)
while (done ≠ 0);
```

- Shared Memory Programmierung – Überblick
  - Paralleles Programm
    - Weiteres Problem
      - Die Anweisung **done:=done-1** kann gleichzeitig von mehreren Prozessen ausgeführt werden.
      - Aber die Anweisung besteht aus mehreren Operationen:
        - » Lesen des Werts von **done** in ein Register
        - » Dekrement um 1
        - » Zurückschreiben in Speicher
      - Problem:
        - » Wenn zwei Prozesse den Wert lesen, jeweils um 1 dekrementieren und den Wert zurück schreiben in einer bestimmten Ordnung. Die Variable **done** ist dann aber n ur um den Wert 1 dekrementiert und nicht um 2!

- Shared Memory Programmierung – Überblick
  - Paralleles Programm
    - Lösung: Atomare Operation
      - Unteilbare Operation in Bezug auf andere parallele Operationen
    - Implementierungsmöglichkeit einer atomaren Operation auf einer gemeinsamen Variablen:
      - Wechselseitiger Ausschluss (mutual exclusion)
        - » Kritischer Bereich: nur ein Prozess darf zu einem Zeitpunkt den kritischen Bereich ausführen
        - » **`critical`**, **`end critical`** kennzeichnen kritischen Code-Bereich

Multiprozessor mit gemeinsamem Speicher

- Shared Memory Programmierung – Überblick
  - Paralleles Programm (V.3)

```
procedure dorow (value i, done, n, a, x, c)
  shared n, a[n,n], x[n], c[n], done;
  private i,j,sum,priv;
  sum = c[i];
  for j=1 step 1 until i-1
            {copy x[j] into priv; /*Get x[j] wenn verfügbar*/
            sum := sum + a[i,j]*priv;}
  produce x[i]:= sum; /*Make x[i] available to others*/
  critical
            done:=done-1;
  end critical
  return;
end procedure

shared n, a[n,n], x[n], c[n], done;
private i;
done:=n;
for I:=1 step 1 until n-1
            {void x[i];
            create dorow (i,done,n,a,x,c);
i:=n;
void x[i];
call dorow (i,done,n,a,x,c)
while (done = 0);
```

Multiprozessor mit gemeinsamem Speicher

- Synchronisation
  - Synchronisationsmechanismen werden typischer Weise mit Software-Routinen auf Benutzerebene aufgebaut.
  - Diese beziehen sich auf Synchronisationsbefehle in Hardware
    - „Kleinere“ Multiprozessoren, oder wenn kaum Blockierung:
      - Nicht-unterbrechbare Instruktion oder Instruktionsfolge, die in atomarer Weise den Wert lesen und verändern, und möglicherweise auch zurück schreiben
    - Größere Maschinen, Gefahr der Blockierung
      - Komplexere Synchronisationsmechanismen sind notwendig

Multiprozessor mit gemeinsamem Speicher

- Synchronisation
  - Grundlegende Hardware-Primitive (Beispiele)
    - Unteilbarer Austausch (atomic exchange, swap)
      - Tauscht den Wert in einem Register mit einem Wert im Speicher
        - » Beispiel: Aufbau einer Synchronisationsoperation *Sperre,* bei der der Wert 0 anzeigt, dass die Sperre offen ist und der Wert 1 anzeigt, dass die Sperre nicht erreichbar ist.
    - **`test-and-set`**-Befehl
      - Prüft den Wert einer Speicherstelle und setzt einen Wert, falls die Bedingung erfüllt ist
    - Problem: Lesen und Schreiben einer Speicherstelle in einer unteilbaren Instruktion
      - Macht Kohärenz komplizierter, da die Hardware keine andere Operation erlauben darf und kein Deadlock entstehen darf!

Multiprozessor mit gemeinsamem Speicher

- Synchronisation
  - Grundlegende Hardware-Primitive
    - Paar von Instruktionen, bei der die zweite Instruktion den Wert liefert, von dem abgeleitet werden kann, ob das Paar ausgeführt worden ist so als ob die Instruktionen atomar wären
      - Wenn ein Paar von Instruktionen effektiv atomar ist, dann kann kein anderer Prozessor den Wert zwischen dem Instruktionspaar ändern
      - **load linked (load locked) – store conditional**
        - » Falls der Inhalt der Speicherstelle, die in der load linked Instruktion spezifiziert worden ist geändert worden ist, bevor der store conditional Befehl auftaucht, dann scheitert der store conditional.
        - » store conditional ist definiert, einen Wert zu liefern der anzeigt, ob die Speicheroperation erfolgreich war oder nicht

Multiprozessor mit gemeinsamem Speicher

- Synchronisation
  - Grundlegende Hardware-Primitive
    - Beispiel mit Instruktionspaar: Implementierung einer atomaren Austauschoperation auf einer Speicherstelle, die im Register R1 spezifiziert ist

```
try:    OR    R3,R4,R0     ;mov exchange value
        LL    R2,0(r1)     ;load linked
        SC    R3,o(r1)     ;store conditional
        BEQZ  R3, try      ;branch store fails
        MOV   R4,R2        ;put load value in R4
```

    - Am Ende dieser Folge sind der Inhalt von R4 und der Inhalt der Speicherstelle, die in Register R1 adressiert worden ist, atomar ausgetauscht worden
    - Falls ein Prozessor den Wert im Speicher zwischen **LL** und **SC** modifiziert, dann liefert SC den Wert 0 in R3 und die Folge wird wiederholt

- Synchronisation
  - Implementierung von Spin Locks
    - Spin Locks:
      - Sperren, bei denen ein Prozessor ständig versucht, die Sperre zugeteilt zu bekommen
        - » Werden von einem Programmierer dann verwendet, wenn zu erwarten ist, dass die Sperre nur für eine kurze Zeit geschaltet ist und wenn das Setzen der Sperre wenig Zeit kostet, wenn die Sperre frei ist
        - » In manchen Fällen ungeeignet, da der Prozesor in einer Schleife wartet, bis die Sperre frei ist

Multiprozessor mit gemeinsamem Speicher

- Synchronisation
  - Implementierung von Spin Locks
    - Lock-Variable steht im Speicher (im Beispiel steht die Adresse der Speicherstelle in Register R1)
    - Prozessor kann fortlaufend mit Hilfe einer atomaren Exchange-Operation versuchen, die Sperre zu erhalten, und testen, ob die Exchange Operation die Sperre als frei anzeigt:

```
          DADDUI R2,R0,#1
Lockit:   EXCH   R2,0(R1)     ;atomic exchange
          BNEZ   R2,Lockit
```

    – Um die Sperre frei zu geben, schreibt der Prozessor eine 0

Multiprozessor mit gemeinsamem Speicher

- Synchronisation
  - Implementierung von Spin Locks mit Hilfe eines Kohärenzmechanismus für einen Multiprozessor
    - Die Lock-Variable kann unter Zuhilfenahme des Cache-Kohärenzmechanismus im Cache gehalten werden
    - Vorteil:
      - Spinning kann auf einer Kopie im lokalen Cache durchgeführt werden
      - Beobachtung: zeitliche Lokalität von Zugriffen auf Lock-Variable von einem Prozessor
    - Aber: Änderung der Spin-Routine ist notwendig
      - Falls mehrere Prozessoren versuchen, die Sperre zu bekommen, wird jeder eine Schreib-Operation generieren. Die meisten dieser Schreib-Operationen führen zu einem Write-Miss, da jeder Prozessor versucht, die Lock-Variable in einem Exclusive-Zustand zu erhalten

Multiprozessor mit gemeinsamem Speicher

- Synchronisation
  - Implementierung von Spin Locks mit Hilfe eines Kohärenzmechanismus für einen Multiprozessor
    - Änderung der Spin-Routine

```
Lockit:   LD     R2,0(R1)      ;load of lock
          BNEZ   R2, Lockit    ;not available – spin
          DADDUI R2,R0,#1      ;load locked value
          EXCH   R2,0(R1)      ;atomic exchange
          BNEZ   R2,Lockit     ;branch if lock wasn‘t 0
```

      - Prozessor prüft laufend die Lock-Variable (Lesen einer Kopie im lokalen Cache) bis er sieht, dass die Variable verfügbar ist
      - Dann versucht er, die Sperre zu erhalten durch Ausführen der Swap-Operation
      - Rennen mit allen anderen Prozessoren, die ebenfalls die Sperre erhalten wollen
        - » Alle verwenden Swap-Operation
        - » Nur der Gewinner sieht eine 0

Multiprozessor mit gemeinsamem Speicher

- Synchronisation
  - Implementierung von Spin Locks mit Hilfe eines Kohärenzmechanismus für einen Multiprozessor
    - Wie verwendet der Prozessor das (implementierte) Cache-Kohärenzprotokoll
      - Wenn ein Prozessor mit der Sperre ein 0 schreibt, werden alle anderen Caches invalidiert und müssen den neuen Wert holen, um ihre Kopie der Lock-Variablen zu aktualisieren
      - Ein dieser Caches bekommt die Kopie des Wertes 0 (nicht gesperrt) zuerst und führt die Swap-Operation aus
      - Wenn die Aktualisierungen aufgrund der Cache-Fehlzugriffe der anderen Prozessoren erledigt sind, dann sehen sie, dass die Variable wieder gesperrt ist

- Synchronisation
  - Implementierung von Spin Locks mit Hilfe eines Kohärenzmechanismus für einen Multiprozessor
    - P0 hat den Lock und gibt ihn im Schritt 2 frei

Step	Processor 0	Processor 1	Processor 2	Coherence state of lock	Bus activity
1	Has lock	Spins, testing if lock =0	Spins, testing if lock =0	Shared	none
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0

Beispiel: Hennessy/Patterson: A Quantative Approach. 593, ff.

Multiprozessor mit gemeinsamem Speicher

- Synchronisation
  - Implementierung von Spin Locks mit Hilfe eines Kohärenzmechanismus für einen Multiprozessor
    - Schritte 3-5: die beiden anderen Prozessoren konkurrieren um das Lesen des nicht gesperrten Werts während des Swappings

Step	Processor 0	Processor 1	Processor 2	Coherence state of lock	Bus activity
3		Cache miss	Cache miss	Shared	Bus services P2 cache miss, write back from P0
4		(waits while bus busy)	Lock=0	Shared	Cache Miss for P2 satisfied
5		Lock=0	Executes swap, gets cache miss	Shared	Cache Miss for P1 satisfied

Beispiel: Hennessy/Patterson: A Quantative Approach. 593, ff.

Multiprozessor mit gemeinsamem Speicher

- Synchronisation
  - Implementierung von Spin Locks mit Hilfe eines Kohärenzmechanismus für einen Multiprozessor
    - Schritte 6 und 7: P2 gewinnt und erreiht kritischen Bereich während der Versuch von P1 fehlschlägt

Step	Processor 0	Processor 1	Processor 2	Coherence state of lock	Bus activity
6		Executes swap, gets cache miss	Completes swap: returns 0 and set Lock =1	Exclusive (P2)	Bus services P2 cache miss; generates invalidate
7		Swap completes and returns 1, and set Lock =1	Enter critical section	Exclusive (P1)	Bus services P1 cache miss, generates write miss
8		Spins, testing if lock=0			

Beispiel: Hennessy/Patterson: A Quantative Approach. 593, ff.

- Synchronisation
  - Barrier Synchronisation
    - Erzwingt, dass die Prozesse warten, bis alle die Schranke (Barrier) erreicht haben, und gibt dann alle frei
    - Typische Implementierung mit Hilfe von zwei Spin Locks
      - Einer schützt den Zähler, der die ankommenden Prozesse abhakt
      - Einer hält die Prozesse bis der letzte Prozess die Schranke erreicht

- Literatur
  - Hennessy, J.; Patterson, D.: Computer Architecture A Quantative Approach. Morgan Kaufmann Publishers, San Francisco, CA, 2003, 3. Auflage: Kap. 6.7