Wissenschaftliches Rechnen – Paralleles Höchstleistungsrechnen

Christian Engwer
http://wwwmath.uni-muenster.de/num/Vorlesungen/WissenschaftlichesRechnen_SS11/

Abgabe 16. 5. 2011. Abgabe der Programmieraufgaben bis per Email an christian.engwer@uni-muenster.de, schriftliche Abgabe Dienstags in der Vorlesung.

- Alle Programmierübungen müssen per Email und in ausgedruckter Form abgegeben werden.
- Achten sie darauf, ihr Programm ordentlich zu formatieren und gut zu kommentieren.
  In Zukunft wird die Form mit in die Bewertung eingehen.

In der Vorlesung haben Sie spezielle Hardware Befehle kennengerlernt, die dabei helfen den Eintritt in einen *kritischen Abschnitt* zu koordinieren, indem sie stärkere Garantien bzgl. der Speicherkonsistenz machen, als dies bei gewöhnlichen read/write Operationen der Fall ist.

Operationen, die Sie in der Vorlesung kennengelernt haben sind:

- test-and-set
- atomic-swap
- fetch-and-increment
- compare-and-swap

Übung 1 Queue Lock

In der Vorlesung wurde das TAS-Lock vorgestellt. Dabei fiel auf, dass durch die Kokurenz um das Lock ständig Cache Misses erzeugt werden, die wiederum den Bus belasten und dadurch den Eintritt in den kritischen Abschnitt unnötig verzögern.

Als Verbesserung wurde nun das TTAS-Lock konstruiert. Durch dieses lässt sich die Anzahl an Cache Misses deutlich verringern, da erst nach Freigabe des Locks überhaupt versucht wird das Lock zu bekommen. Trotzdem kommt es in diesem Moment zu starken Verkehr auf dem Speicherbus, da alle Prozesse auf die gleiche Speicherstelle zugreifen möchten.

Eine Idee den Algorithmus weiter zu verbessern ist das sogenannte *Queue Lock*. Hierbei stellen sich alle Prozesse brav der Reihe nach an. Es gibt eine Liste von Speicherstellen (als Lock Variablen) und jeder Prozess testet auf eine andere Stelle, bis das Lock verschwindet, sprich der Wert der Speicherstelle *false* ist. Verläßt ein Prozess seinen kritischen Abschnitt, so setzt er das Lock des nächsten Prozesses auf *false*. Bei diesem Verfahren ist darauf zu achten, dass garantiert ist, dass nur ein Prozess auf eine Speicherstelle testet, d.h. dass ein Platz in der Queue an nur einen Prozess vergeben wird.

- Implementieren Sie das *Queue Lock* in der Pseudo Sprache, wie sie im Skript eingeführt wurde.
- Verwenden Sie einen der Hardware Befehle, um die Speicherkonsistenz zu garantieren.
- Begründen Sie die Wahl des Hardware Befehls.

4 Punkte

Übung 2 Effekte bei Software Locks

Betrachten Sie folgende einfache parallele Anwendung bestehend aus zwei Prozessen, welche eine gemeinsame Variable jeweils 100000 mal hochzählen:

Programm 1 (Paralleles Hochzählen einer gemeinsamen Variable)

```
parallel increment
{
    int sum = 0;

    process Π1                          process Π2
    {                                   {
        for (i = 1; i ≤ 100000; i++)        for (i = 1; i ≤ 100000; i++)
            sum = sum + 1;    // KA            sum = sum + 1;    // KA
    }                                   }
}
```

Die sequentielle Ausführung beider Prozesse würde eine Anwendung liefern, welche die gemeinsame Variable `sum` schrittweise bis zum Wert 200000 hochzählt. Die beiden Prozesse werden jedoch im Allgemeinen nicht sequentiell nacheinenander abgearbeitet und die Anweisung `sum = sum+1` bildet jeweils in beiden Prozessen einen kritischen Abschnitt. Daher ist zu erwarten, dass `sum` eine Zahl enthält, die kleiner ist als 200000, nachdem beide Prozesse ihre Arbeit getan haben.

Wie Sie in der Vorlesung gelernt haben, lässt sich dieses Verhalten dadurch korrigieren, dass die kritischen Abschnitte durch wechselseitigen Ausschluss behandelt werden. Dafür haben Sie beispielsweise den Peterson Algorithmus kennengelernt, welcher die folgende Modifikation der obigen Anwendung liefert:

Programm 2 (Variante mit Peterson Lock)

```
parallel increment_peterson
{
    int in1 = 0, in2 = 0, last = 1;
    int sum = 0;

    process Π1                              process Π2
    {                                       {
        for (i = 1; i ≤ 100000; i++)            for (i = 1; i ≤ 100000; i++)
        {                                       {
            in1 = 1;                                in2 = 1;
            // (*)                                  // (*)
            last = 1;                               last = 2;
            // (*)                                  // (*)
            while (in2 ∧ last == 1) ;               while (in1 ∧ last == 2) ;
            sum = sum + 1;    // KA                sum = sum + 1;    // KA
            in1 = 0;                                in2 = 0;
        }                                       }
    }                                       }
}
```

Aufgabenteil a)

- Implementieren Sie die Variante mit Peterson Lock in `C++` unter Verwendung der Bibliothek `Pthreads`. Ihr Programm soll
  1. die beiden Prozesse als Threads erzeugen,
  2. auf die Beendigung der beiden Prozesse warten
  3. und schließlich den Wert der gemeinsamen Variable `sum` ausgeben.
- Übersetzen Sie das Programm ohne Optimierung durch den Compiler und führen Sie es mehrmalig hintereinander aus (beachten Sie die Hinweise). Sie werden feststellen, dass die berechnete Summe oftmals kleiner ist als der erwartete Wert 200000, obwohl durch den Peterson Algorithmus der wechselseitige Ausschluss garantiert sein sollte.

Hinweise: Beim `GNU C/C++` Compiler wird die Optimierung des Compilers durch die Option `-O0` ausgeschaltet. Die benötigen Compileroptionen zum Übersetzen des Programms sind also `-lpthread -O0`. Das mehrmalige Ausführen des gesamten Codes können Sie bequem durch eine zusätzliche Schleife in der Methode `main` erreichen.

Aufgabenteil b)

In der Vorlesung haben Sie gelernt, dass moderne CPUs ihre Rechenleistung steigern, indem sie die Reihenfolge der Maschinenbefehle umsortieren. Ein Maschinenbefehl kann vorgezogen werden, falls sich die zur Ausführung des Befehls benötigten Daten bereits fertig berechnet im Speicher befinden (*out-of-order execution*). Sogenannte *Memory Barrier* können dazu benutzt werden manuell Einfluss auf die out-of-order execution zu nehmen. Diese sind spezielle Maschinenbefehle, welche der CPU eine Bedingung auferlegen, in welcher Reihenfolge sie Speicheroperationen auszuführen hat. Dazu folgendes Beispiel:

```
OP_1;
...
OP_n;
memory_barrier;
OP_{n+1}
...
```

Der Befehl `memory_barrier` sorgt dafür, dass die CPU die Operationen `OP_1`, ..., `OP_n` vollständig ausführt, bevor die nachfolgenden Befehle ausgeführt werden.

- Binden Sie die auf der Vorlesungsseite erhältliche Header-Datei `membarrier.hh` in ihren Code ein. Diese enthält einen Befehl `memory_barrier()`, welcher das gerade beschriebene Konzept realisiert. Fügen Sie den Befehl an den beiden durch `(*)` markierten Stellen in ihren Code ein und führen Sie anschließend wieder den unoptimierten Code mehrmalig aus.
- Welche Veränderung beobachten Sie? Wie lässt sich dieser Effekt erklären? Geben Sie durch ein Beispiel an, wie es ohne die Memory Barrier dazu kommen kann, dass der Peterson Lock versagt.

Aufgabenteil c)

- Übersetzen Sie das Programm nun mit voller Optimierung (beachten Sie den Hinweis) und testen Sie es durch mehrmaliges Ausführen sowohl ohne die Memory Barrier als auch mit ihnen. Welchen Effekt können Sie beobachten?

In `C/C++` gibt es die Möglichkeit Speicherbereiche so zu markieren, dass die Reihenfolge von Lese- und Schreiboperationen auf diesen Speicherbereichen beim Compilieren des Programms nicht verändert werden kann. Dies geschieht mit dem Schlüsselwort `volatile`. Lese- und Schreiboperationen auf Variablen, die mit diesem Schlüsselwort versehen sind, gelangen in exakt der gleichen Reihenfolge in das übersetzte Programm, wie es im Quellcode steht und dürfen auch nicht wegoptimiert werden.

- Versehen Sie die vier gemeinsamen Variablen `in1`, `in2`, `last` und `sum` mit dem zusätzlichen Schlüsselwort `volatile`. Beispiel für die Syntax: `volatile int in1 = 0;`
- Übersetzen Sie das Programm erneut mit voller Optimierung und testen Sie es wieder durch mehrmaliges Ausführen sowohl ohne die Memory Barrier als auch mit ihnen. Erklären Sie den von Ihnen beobachteten Effekt.

Hinweis: Beim `GNU C/C++` Compiler wird die volle Optimierung des Compilers durch die Option `-O3` eingeschaltet.

8 Punkte