ALP4 - Nichtsequentielle Programmierung 7

Tobias Kranz (414 71 30)
Johannes Rohloff (470 34 87)

7. Juni 2013

Aufgabe 1

Prinzipiell eignen sich insbesondere solche Algorithmen zur Nebenläufigkeit, die auf dem Teile-und-Herrsche (Devide and conquer) Paradigma aufsetzen, da die Probleme hier nativ bereits in getrennt lösbare Teilprobleme zerlegt werden. Es stellt sich dabei die grundlegende Frage nach dem Laufzeitgewinn durch die nebenläufige Abarbeitung, da diese ein Mehr an Verwaltungsaufwand (bei großen Eingabefolgen wird z.B. eine unvertretbar große Anzahl an Prozessen verwaltet) mit sich bringt und die jeweils gewonnen Teilergebnisse (mittels Kommunikation oder Nutzung gemeinsamen Speichers) zusammengesetzt werden müssen (wobei sich wiederum diverse Synchronisationsprobleme ergeben können).

Quick-, Merge- (Implementiert in [MAURER] S.10 f.) und Bucketsort (Generalisierung von Counting Sort) sind somit wohl die am ehesten geeigneten Verfahren für nichtsequentielle Konstrukte. Dennoch sind natürlich auch weitere Verfahren parallelisierbar (z.B. bitonic sort, sample sort, radix sort), wobei sich wie oben erwähnt immer die Frage nach der Laufzeit stellt [1]

Aufgabe 2

Implementierung des Vergleiches von zwei Bäumen in drei nebenläufigen Prozessen, siehe A2.zip:

src/binTree/def.go gibt das Interface des hierzu (in imp.go mit geringer Funktionalität) implementierten Binärbaums wieder.

src/main/U7A2.go implementiert den eigentlichen Vergleich.

Aufgabe 3

Variationen (mit Ausgabe) zum Programm in [MAURER] S. 13 siehe A3.go.

Es zeigt sich schon mit der unveränderten Implemmentierung werden Zählerstände von 0 erreicht. Das Heraufsetzen der Prozessanzahl resultiert in den geringsten Zählerhöchstständen bei gleichzeitig niedrigsten Durchschnittszählerständen.

[1] siehe hierzu bspw. `parasol.tamu.edu/publications/download.php?file_id=191`

Aufgabe 5

a) Der Im Buch gegeben Code sieht wie folgt aus:

```
TEXT TestAndSet(SB),7,$0
        MOVL valptr+0(FP), BP    // BP = &b
        MOVL $1, AX                                  // AX = 1
        LOCK                                         // Bus sperren
        XCHGL AX, 0(BP)                        // AX = *b || *b =1 (=true)
        MOVL AX, ret+4(FP)                     // AX ist Rueckgabewert
        RET
```

Dieser soll nun Zeile für Zeile erläutert werden:

1. Mit dieser Zeiel wird die Unterfunktion benannt, auf den Wert TestAndSet. Der Wert in der Klammer ist der Stack Pointer. Dieser verweist auf die Stelle an der die Testvariable liegt. Das erste Argument legt dabei die Länge der Unterfunktion fest.
2. Hier wird der BasPointer Auf den wert des Aktuellen Stacks gestzt. Dies ist Notwendig um den Aktuellen wert der Testvariable zu bekommen.
3. Setzt den Wert des Registers AX auf 1. (AX fasst dabei ein long, wegen MOVL)
4. Hier wird der Bus gesperrt. Dies ist nötig, damit der nächste aufruf (das eigentliche Austauschen) atomar geschehen kann.
5. Hier wird die Testvariable und der Wert 1 (im Regiszer AX) gegeneinander getauscht. Somit liegt in AX der Wert vor dem schreiben und auf dem Stack liegt eine 1. Diese Befehl kann atomar ausgeführt werden wegen des Locks.
6. Hier wird als Rückgabe der wert des Registers AX an die oberste Stelle des Stacks geschrieben
7. Die Funktion ist bendet und es wird zurückgesprunegn.

Somit erfüllt diese Implementation die Spezifikationen der TestAndSet Funktion.

b) XADD ist eine Instruktion des x86 Befehlsatzes. Sie tauscht die beiden Operanden und vertauscht dann schreibt sie in den ersten Operanden die Summe der beiden Operanden. Mit dem Lock Befehl kann sie atomar ausgeführt werden. Somit kann mit diesem Befehl eine FetchAndAdd Operation atomar realisiert werden.

```
// Definition des Buches fuer die FetchAndAdd Funktion
// inertn realisiert mit XADD
FetchAndAdd(k,n) = AddUint32(k,n) - n

var interested uint     // Wert fuer die aktuelle Warteschlange
var turn uint                   // wert in der Wartschleife des Aktuellen Proz

func Lock() {
        // "zieht" die naechst hoehere Wartenummer und wartet bis er an der Tu
```

```
        myTurn = FetchAndAdd(*interested,1)
        for myTurn != turn {Null()}
}

func Unlock(){
        // beim Austritt die den aktuellen Prozess erhoehen
        FetchAndAdd(*turn,1)
}
```