Nichtsequentielle und verteilte Programmierung (SS 2017)

Prof. Dr. Margarita Esponda

Übungsblatt 1

Sergelen Gongor, Boyan Hristov

Tutor: Lilli Walther DI. 12.00-12.00

Aufgabe 1

a) Datenparallelität (DLP Data-Level-Parallelism)

Datenparallelität ist eine Form der Parallelisierung über mehrere Prozessoren in parallel-computing-Umgebungen. Es konzentriert sich auf die Verteilung der Daten auf verschiedene Knoten, die auf die Daten parallel zu betreiben. Es kann auf regelmäßige Datenstrukturen wie Arrays und Matrizen angewendet werden, indem Sie auf jedes Element parallel arbeiten.

In einem Multiprozessorsystem einen SIMD (Single Struction Multiple Data) ausführen wird Datenparallelität erreicht, wenn jeder Prozessor die gleiche Aufgabe auf verschiedene Stücke von verteilten Daten führt. In einigen Situationen steuert einzige Thread (single execution thread) Operationen auf alle Teile der Daten. In anderen Fällen verschiedene Threads steuern die Operation, aber sie führen den gleichen Code.

Datenparallelität basiert auf einer Zerlegung der Datenbasis.

Source: https://en.wikipedia.org/wiki/Data_parallelism

b) Funktionsparallelität (TLP Task-Level-Parallelism / control-Parallelism)

Task-Parallelität (auch bekannt als Funktionsparallelität und Steuerungsparallelität) ist eine Form der Parallelisierung von Computer-Code über mehrere Prozessoren in parallel computing-Umgebungen. Funktionsparallelität konzentriert sich auf die Verteilung von Tasks/Funktionen, die gleichzeitig auf verschiedenen Prozessoren Prozesse oder Threads durchführen. Es steht im Gegensatz zu Datenparallelität als eine andere Form der Parallelität.

Source: https://en.wikipedia.org/wiki/Task_parallelism

Aufgabe 2

Der Unterschied zwischen Prozessen und Threads (nach der Vorlesungsdefinition).

Die Prozesse haben ein eigenes Adressraum (eigene Umgebung) und werden durch das Betriebssystem verwaltet, wobei Threads existieren innerhalb eines Prozesses, haben ein gemeinsames Adressraum (dieses des Prozesses) und werden durch das Prozess verwaltet.

Ein Thread (sequentieller Prozess) ist Teil eines Prozesses, der leichtgewichtig ist.

- Thread ist ein einfacher Kontrollfluss mit einzigen Befehlszähler. Prozess kann mehrere Befehlszähler (PC) haben.

Aufgabe 3

a) Wann ist ein Algorithmus determiniert?

Ein Algorithmus ist determiniert wenn bei gleichem Startzustand oder Eingabewert den gleichen Endzustand oder Ausgabewerte hat.

b) Wann ist ein Algorithmus nicht deterministisch?

Die Reihenfolge der Zustände in einem Algorithmus ist zu jedem Zeitpunkt unterschiedlich und nicht festgelegt und damit auch sein Ablauf nicht reproduzierbar.

c) Ist jeder deterministische Algorithmus determiniert? Begründen Sie Ihre Antwort.

Ja, jeder deterministische Algorithmus determiniert. Weil da der Algorithmus bei gleichem Startzustand oder Eingabewert den gleichen Endzustand oder Ausgabewerte bekommen.

d) Ist jeder determinierte Algorithmus auch deterministisch? Begründen Sie Ihre Antwort anhand von Beispielen.

Nein, Beispiel von Quicksort. Da wird zufällig den pivot Element ausgewählt. Obwohl es jedes mal ein andere Pivot gibt, das Ergebnis ist immer gleich. Nur es ist nicht möglich die Abläufe reproduzieren. D.h. Abläufe sind immer anders.

Nein. Ein gutes Beispiel dagegen wurde in der Vorlesung erwähnt – Quicksort mit zufällig gewählten Pivot. Dabei ist die Zustandsreihenfolge immer unterschiedlich, die Rückgabe ist aber für jede Eingabe immer gleich – ein sortiertes Array.

Weiter könnte man die Aufsummierung aller Elemente in einer Liste durch mehrere Threads als Beispiel geben. Welche Indizes in der Liste zuerst zusammenaddiert werden ist nicht bekannt, das Ergebniss ist aber immer die aufsummierte Liste.

Aufgabe 4

Eine nebenläufige, asynchrone Ausführung von Prozessen führt immer zu nicht deterministischen Abläufen? Begründen Sie Ihre Antwort.

(aus dem VL-Folien) Eine nebenläufige asynchrone Ausführung von Prozessen führt in der Regel zu nichtdeterministischen Abläufen, die trotzdem zu determinierten Ergebnissen führen können.

Eine nebenläufige, asynchrone Ausführung von Prozessen führt in der Regel immer zu nicht deterministischen Abläufen, wie schon in der Vorlesung erwähnt wurde. Es ist so, weil die Verwaltung der Prozessen von dem Betriebssystem gemacht wird, wobei mehrere Fakten für die Entscheidung welches Prozess vor welches ausgeführt wird, diese Entscheidung ist für jede Ausfuhrung unterschiedlich. Man kann aber immer ein Scheduler konstruieren, der die nebenläufige Prozessen immer in der selben Reihenfolge ausführen kann, deswegen ist das theoretisch keine feste Regel, obwohl in der Praxis immer so ist.

Aufgabe 5

a) Wir haben ein kurzes Script mit JavaScript hergestellt, der automatisch alle mögliche Varianten herstellt. Dafür muss man die letzte Version von Node.js haben und das folgende in dem selben Ordner ausführen

   node generator.js > out.txt

   Das führt das Programm aus und schreibt das Ergebniss in einer Datei out.txt.

   Ergebniss:

   p1 -> p2 -> p3 -> r1 -> r2 -> r3 -> (x = 1, y = 1, z=2)

   p1 -> p2 -> r1 -> p3 -> r2 -> r3 -> (x = 2, y = 2, z=4)

   p1 -> p2 -> r1 -> r2 -> p3 -> r3 -> (x = 2, y = 2, z=4)

   p1 -> p2 -> r1 -> r2 -> r3 -> p3 -> (x = 2, y = 2, z=1)

   p1 -> r1 -> p2 -> p3 -> r2 -> r3 -> (x = 0, y = 0, z=0)

   p1 -> r1 -> p2 -> r2 -> p3 -> r3 -> (x = 0, y = 0, z=0)

   p1 -> r1 -> p2 -> r2 -> r3 -> p3 -> (x = 0, y = 0, z=1)

   p1 -> r1 -> r2 -> p2 -> p3 -> r3 -> (x = 0, y = 2, z=2)

   p1 -> r1 -> r2 -> p2 -> r3 -> p3 -> (x = 0, y = 2, z=1)

   p1 -> r1 -> r2 -> r3 -> p2 -> p3 -> (x = 0, y = 2, z=1)

   r1 -> p1 -> p2 -> p3 -> r2 -> r3 -> (x = 1, y = 1, z=2)

   r1 -> p1 -> p2 -> r2 -> p3 -> r3 -> (x = 1, y = 1, z=2)

   r1 -> p1 -> p2 -> r2 -> r3 -> p3 -> (x = 1, y = 1, z=1)

   r1 -> p1 -> r2 -> p2 -> p3 -> r3 -> (x = 1, y = 2, z=3)

   r1 -> p1 -> r2 -> p2 -> r3 -> p3 -> (x = 1, y = 2, z=1)

   r1 -> p1 -> r2 -> r3 -> p2 -> p3 -> (x = 1, y = 2, z=1)

   r1 -> r2 -> p1 -> p2 -> p3 -> r3 -> (x = 1, y = 2, z=3)

r1 -> r2 -> p1 -> p2 -> r3 -> p3 -> (x = 1, y = 2, z=1)

r1 -> r2 -> p1 -> r3 -> p2 -> p3 -> (x = 1, y = 2, z=1)

r1 -> r2 -> r3 -> p1 -> p2 -> p3 -> (x = 1, y = 2, z=1)

b) 20 verschiedene verzahnte Ausführungen sind möglich.

$$\binom{n}{n_1, n_2, \ldots n_k} = \frac{(n_1+n_2+\ldots+n_k)!}{(n_1! * n_2! * \ldots * n_k!)}$$

mit der Formel Abläufe $\binom{3,3}{} = \frac{(3+3)!}{(3! * 3!)} = 20$

Aufgabe 6

a) Das kann man nicht mit Sicherheit sagen, da es total von dem Prozess-Scheduler abhängig ist. Falls dieser total fair ist, dann kann es sogar sein, dass kein Prozess überhaupt beendet wird. Falls der Counter aber schon z.B. <= -7 ist und Thread 2 als erstes ausgeführt wird, dann wird dieses auch zuerst beendet.

b) Analog wie bei a) ist es vom Scheduler abhängig. Falls dieser total fair ist, dann werden mit Sicherheit die beide Prozessen nie beendet (Falls nur bis zu 7 Schritte je Thread ausgeführt werden).

In der Praxis aber sind die Schedulers nicht so fair und führere mehrere Schritte eines Threads auf einmal, deswegen werden irgendwann beide Prozesse beendet.
Man kann das aber nie beweisen.

Aufgabe 7

Java source Code ist attached.

Aufgabe 8

Ein Array mit n Zahlen zusammen addieren a) Wie viele Prozesse können sinnvollerweise maximal verwendet werden? b) Analysieren Sie die Zeitkomplexität des Algorithmus.

Angenommen, es sind n Zahlen $\binom{n}{a_1, a_2, a_3, \ldots, a_n}$. Und wir addieren die Zahlen

paarweise (Datenparallelität). D.h. Schritt-1 hat dann $\lfloor \frac{n}{2} \rfloor$ Additionen. Und es

bleiben $\lceil \frac{n}{2} \rceil$ Zahlen, die wir weiter paarweise addieren wollen. Schritt-2 hat dann $\lfloor \frac{n}{4} \rfloor$ Additionen und es sind dann $\lceil \frac{n}{4} \rceil$. Und so weiter addieren wir bis die Schritte-*k*, dass das letzte Paar aufaddieren. D.h. es sind dann k parallele Schritte gewesen. Wenn n eine gerade Zahl ist dann $n=2^k$.

a) Wir brauchen maximal $\lceil \frac{n}{2} \rceil$ Prozesse um ein Array mit n Zahlen zusammen zu addieren
b) Wir berechnen die Zeitkomplexität mit der maximalen Anzahl von $\frac{n}{2}$ Prozessen. Wir wissen dass wir für die Addition insgesamt k Schritte benötigen. Angenommen n ist eine gerade Zahl dann ist $n=2^k$. Daraus folgt $k=\log_2 n$.
   Zeitkomplexität ist $T(n)=O(\log_2 n)$