Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences
Fachbereich Informatik
Lehrgebiet Programmierung komplexer Systeme
Prof. Dr. Rudolf Berrendorf

Programmierung paralleler Anwendungen
SS 14
Ausgabe : 24.04.2014
Bearbeiten bis : 14.05.2014

Übung 3 (Java Threads)

Aufgabe 1 (7 Punkte) :
In den Unterlagen zu dieser Übung ist ein Java-Programm in `TreeCalculation.java` vorgegeben, mit dem ein vollständiger Binärbaum erzeugt und anschließend traversiert wird (siehe letzte Übung). Die *Verwaltung* der Berechnung geschieht in der Klasse `TreeCalculation`, während die Klasse `Tree` den eigentlichen Binärbaum bzw. einen Knoten davon darstellt und die Traversierunsgmethode(n) bereitstellt.

In der Methode `processTree` wird sequentiell der Baum traversiert und dabei eine Summe aller Knotenwerte berechnet. Weiterhin ist bereits der Rahmen einer Methode `processTreeParallel` vorhanden, die für eine parallele Traversierung gedacht ist. Analog zur letzten Übung können Sie in den Methoden `preProcess` und `postProcess` der Klasse `TreeCalculation` geeignete Programmlogik einbauen, die vor bzw. nach dem parallelen Traversieren durchgeführt werden soll. Ebenso können Sie bei Bedarf auch weitere Variablen / Methoden / Klassen definieren.

Die parallele Traversierung soll so erfolgen, dass Sie über das Executor-Framework einmalig einen Thread-Pool mit n Threads erzeugen. Diese Thread-Anzahl n wird aus der Kommandozeile gelesen und der Methode `preProcess` als Parameter übergeben. Kommen Sie in der Traversierung in der Methode `processTreeParallel` zu einem vorgegebenen Level (aus der Kommandozeile gelesen und über `tc.levelParallel` verfügbar), so sollen die beiden Unterbäume als unabhängige Tasks aufgefasst werden, die von Threads des Threads-Pools parallel bearbeitet werden sollen. Innerhalb dieser Teilbäume soll aber sequentiell vorgegangen werden (also alles analog zur letzten Übung).

Die parallelen Tasks sollen jeweils als Ergebnis der Berechnung den Summenwert zu diesem Teilbaum liefern. Eine `CompletionService`-Instanz soll alle Ergebnisse sammeln und zur Gesamtsumme hinzuführen. Sie dürfen damit zum Beispiel *nicht* in einer Task direkt eine globale Summe verändern.

Die Programmlogik soll so sein, dass Sie dynamisch auf der erzeugenden Seite die Anzahl der erzeugten Tasks mit jeder neu erzeugten Berechnungsinstanz für einen Teilbaum erhöhen und auf der verarbeitenden Seite der `CompletionService` solange auf Ergebnisse wartet, solange diese Anzahl noch ungleich 0 ist. Beim Entnehmen eines Resultatwertes soll demzufolge dann auch die Anzahl der Tasks um 1 verringert werden. Nutzen Sie dazu die vorhandenen Methoden, die Sie ggfs. noch modifizieren müssen.

In den Unterlagen ist ein Job-Skript vorhanden, das ihr Programm auf `wr9` mit unterschiedlichen Thread-Zahlen und Parallelitätsleveln startet. Weiterhin ist ein python-Skript angegeben, mit dem Sie aus der Ergebnisdatei des Job-Laufs (die Job-o-Datei) eine übersichtlich Tabelle und ein Diagramm anfertigen sollen, worin der Speedup in Bezug zur Threadzahl gestellt wird und pro fester Task-Zahl eine Meßreihe dargestellt wird. Aufruf dazu: `python speedup.py <Job-o-Datei` (Ignorieren Sie dabei eine Warnung bzgl. eines fehlenden Fonts).

Nutzen Sie für diese Aufgabe Oracle Java, indem Sie einmalig ausführen: `module load java` (im Job-Skript bereits enthalten).

Aufgabe 2 (3 Punkte) :
Die Untersuchungen zu dieser Aufgabe können Sie interaktiv auf `wr0` durchführen. Wählen Sie für ihre Tests zu dieser Aufgabe ihr paralleles Java-Programm und eine kleinere Baumhöhe von 27. Nutzen Sie zu Beginn die Oracle Java-Umgebung: `module load java`

Wenn Sie für diese Aufgabe einen Programmlauf durchführen, so geben Sie in der Kommandozeile zusätzlich den Präfix `time` an, der zu einem beliebigen nachfolgenden Kommando Informationen zur Gesamtausführungszeit des Kommandoaufrufs gibt (die Zeitmessungen im Programm beziehen sich ja nur auf die Teile des Programms, die den Baum traversieren). In der Ergebnisausgabe steht unter `real` dann die Gesamtlaufzeit (die verstrichene Zeit), die von Interesse ist. Ein Beispielaufruf wäre demzufolge: `time java TreeCalculation 27 4 8`

Für die folgenden Aufgabenstellungen ermitteln Sie jeweils die folgenden Leistungsangaben: die reale Gesamtzeit (Ausgabe `time ...`) sowie über die Programmausgabe die sequenzielle Zeit des Traversierens und die parallele Zeit des

Traversierens.

1. Führen Sie einige Testberechnungen mit verschiedenen Kombinationen ihrer Wahl von Parallelitätslevel (2. Aufrufparameter) und Threadanzahl (3. Aufrufparameter) durch und merken sich die Parameterkombinationen und Leistungsangaben.

2. Im Job-Skript gibt es beim Aufruf des Java-Programms zusätzliche Laufzeitoptionen `-Xms80G -Xmx80G`. Was bewirken diese Optionen? Was passiert, wenn Sie diese bei ihren Testaufrufen einfügen bzw. weglassen?

3. Starten Sie das Java-Programm mit der zusätzlichen Laufzeitoption `java -verbose:gc ...`. Was sehen Sie, wenn Sie `-Xms80G -Xmx80G` nutzen bzw. nicht nutzen?

4. Fügen Sie in den Programmcode nach der Baumerzeugung zusätzlich ein: `System.gc();` (siehe Kommntar dort). Was bewirkt dies? Wie wirkt sich dies auf die Leistungsangaben aus? Mit den zusätzlichen Laufzeitoptionen `-Xm...` / ohne diese?

5. Auf dem WR-Cluster sind die beiden Java-Umgebungen OpenJDK und Oracle Java installiert. Per Default wird das OpenJDK genutzt. Oracle Java können Sie mit `module load java` wählen und mit `module unload java` wieder zurück zum OpenJDK schalten. Sehen Sie Leistungsunterschiede zwischen den beiden Umgebungen, wenn Sie innerhalb einer Umgebung das Java-Programm übersetzen (`make clean; make`) und ausführen?