PFP-Team
pfp@i2.cs.fau.de

Übungsblatt 6

Abgabe bis 12.06.2019, 18:00 Uhr

Aufgabe 6.1: Nebenläufigkeitsgrenzen

a) Welche Eigenschaften beeinflussen die effiziente Parallelisierung von Programmen?

b) Wie wirken sich diese auf die Laufzeit eines parallelen Programms aus?

Aufgabe 6.2: Effiziente parallele Programmierung

Denken Sie zurück an Aufgabe 3.4 c), in der Sie parallel ein Bild durch Aufteilung der zu verrichtenden Arbeit einfärben sollten. Betrachten Sie dafür zwei Lösungsansätze:

I: Es werden n `Threads` gestartet und jeder erhält einen etwa gleich großen Bereich der zu färbenden Fläche.

II: Es wird ein `ExecutorService` mit n Arbeitern erzeugt, dem für jeden Punkt ein eigenes `Callable` zur Ausführung übergeben wird.

a) Welche der Lösungen (I/`Thread` oder II/`Callable`) ist vermutlich für große Flächen schneller? Warum?

b) Welche allgemeinen Regeln für die effiziente parallele Programmierung können Sie daraus ableiten?

Aufgabe 6.3: Gesetz von Amdahl

a) Was sagt das Gesetz von Amdahl aus (informell)?

b) Welche Größen verbindet das Gesetz und was stellen diese Größen im Einzelnen dar?

c) Leiten Sie das Gesetz von Amdahl aus oben beschriebenen Größen her.

d) Zeichnen Sie die Speedup-Kurven für drei Programme, bei denen der ideal parallelisierbare Anteil 100%, 90% und 80% beträgt und keine zusätzlichen Kosten durch die Parallelisierung entstehen.

Aufgabe 6.4: Klient/Dienstleister, Chef/Arbeiter

Beschreiben Sie die Gemeinsamkeiten und Unterschiede der beiden Konzepte Klient/Dienstleister (Client/Server) und Chef/Arbeiter (Master/Worker).

Aufgabe 6.5: Monte-Carlo-Simulation

a) Beschreiben Sie die Funktionsweise der in der Vorlesung vorgestellten Monte-Carlo-Simulation zur Berechnung von π.

b) Wie kann das Verfahren angepasst werden, um die Schnittfläche von zweidimensionalen Polygonen zu bestimmen?

Bonusaufgabe 6.6: QueueingThreadPool **6 Punkte**

Implementieren Sie in der Datei `FixedThreadQueueingThreadPool.java` eine Variante des `ExecutorService`. Dieser soll wie ein `ExecutorService` Arbeitsaufträge bearbeiten. Im Unterschied zum `ExecutorService` besteht beim `QueueingThreadPool` die Möglichkeit, fertig bearbeitete Aufträge durch die Methode `take` in der Reihenfolge ihrer Fertigstellung abzurufen. Sie müssen also nicht mehr selbst überprüfen, ob ein Auftrag bereits fertig ist. Benutzen Sie für Ihre Implementierung das vorgegebene `QueueingThreadPool`-Interface.
Gehen Sie dabei so vor, dass Sie zunächst die geforderten Methoden *ohne* `shutdown` implementieren. Ergänzen Sie erst dann Ihren Code so, dass er auch mit `shutdown` funktioniert. Achten Sie darauf, dass Ihre Implementierung thread-sicher ist.

a) Erzeugen Sie im Konstruktor die angegebene Anzahl an Threads, die für die Auftragsabarbeitung verwendet werden sollen. Ansonsten sollen Sie keine weiteren Threads erzeugen.

b) Die Methode `submit` soll mit Hilfe von `FutureTask`-Objekten Arbeitsaufträge erzeugen, die an die Threads verteilt werden. Verwenden Sie zur Verteilung der Aufträge die thread-sichere `LinkedBlockingQueue` aus dem `java.util.concurrent`-Paket.

c) Die `doWork`-Methoden der Threads entnehmen Aufträge aus der Queue und arbeiten sie ab. Ist die Queue leer, so sollen die Threads warten, bis neue Aufträge in der Queue gelandet sind.

d) Fertig bearbeitete Aufträge landen in einer zweiten Queue, die von `poll` und `take` entnommen werden können.

Erweitern Sie Ihre Implementierung nun so, dass sie auch mit `shutdown` umgehen kann.

e) Nachdem `shutdown` aufgerufen wurde, sollen die Threads noch so lange weiterlaufen, bis alle Arbeitsaufträge bearbeitet wurden. Stellen Sie sicher, dass nach dem Aufruf von `shutdown` keine weiteren Arbeitsaufträge mehr angenommen werden und `submit` eine `RejectedExecutionException` wirft. Vermeiden Sie Wettlaufsituationen zwischen `shutdown` und `submit`.

f) Sorgen Sie dafür, dass `take` nicht mehr blockiert, sobald sich der letzte Thread beendet hat (`terminated == true`). Stattdessen soll `take` immer sofort einen Wert (möglicherweise auch `null`) zurückliefern. Stellen Sie außerdem sicher, dass alle anderen Aufrufer, die sich nach Beenden des letzten Threads noch in `take` befinden, auch garantiert zurückkehren. Gehen Sie dazu folgendermaßen vor: `take` soll mitzählen, wie viele Aufrufer sich gerade in der Methode befinden. Wenn sich der letzte Thread beendet, ruft dieser `terminationHandler` auf. Dieser fügt so viele „Giftpillen“ (`poisonpill`) in die Queue fertig bearbeiteter Aufträge, wie sich Aufrufer in `take` befinden. Entnehmen `take` oder `poll` eine Giftpille aus der Queue, dann sollen `take` und `poll` `null` zurückliefern. Vermeiden Sie Wettlaufsituationen zwischen `terminationHandler` und `take`.

Für die Aufgabe stehen Ihnen alle Möglichkeiten des `java.util.concurrent`-Pakets (mit Ausnahme der schon implementierten `ExecutorService`-Varianten) zur Verfügung.

Aufgabe 6.7: Monte-Carlo

Implementieren Sie in der Klasse `MonteCarloImpl` die `computeIntersection`-Methode. Die Methode erwartet zwei Ellipsen (`Ellipse2D.Double`) und soll deren Schnittfläche als `double`-Wert zurückgeben. Zur Berechnung der Schnittfläche soll ein paralleler Monte-Carlo-Algorithmus verwendet werden (ähnlich der Vorlesung, siehe https://en.wikipedia.org/wiki/Monte_Carlo_method). Dabei sollen zufällige Punkte innerhalb einer geeigneten Fläche A ausgewählt werden. Das Verhältnis der Anzahl der Punkte, die innerhalb beider Ellipsen liegen, zur Gesamtzahl der Punkte entspricht für viele Punkte ungefähr dem Verhältnis der Schnittfläche zur Fläche A. Der `computeIntersection`-Methode wird, zusätzlich zu den Ellipsen, die Anzahl der zu verwendenden Threads und die Anzahl der zu betrachtenden Zufallspunkte übergeben.

6 Punkte