Parallellisering av existerande kod

Målsättningen med denna uppgift är att demonstrera de grundläggande premisserna för ”task-based parallelism” och Javas ramverk för att utföra parallella strukturerade beräkningar. Laborationen ska lösas med fork/join-ramverket i JSR166, vilket är inkluderat i Java 7. För dig som sitter med Java 6 finns en jar-fil att ladda ner från Doug Lea’s hemsida:

`http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166.jar`

och instruktioner för hur du talar om för Java att använda JSR166 finns att läsa exempelvis på Dan Grossmans hemsida:

`http://www.cs.washington.edu/homes/djg/teachingMaterials/grossmanSPAC_forkJoinFramework.html`

Del 1

1. Börja med att ta reda på vilken version av Java du kör. Googla om det behövs. I terminalen är detta enkelt med `java -version`.
2. I filen `Quicksort.java` finns en relativt enkel implementation av sorteringsalgoritmen quicksort[1].
3. Skriv en main-metod och anropa `sQsort(int[])` med en godtycklig heltalsarray.
4. Skriv en metod som kontrollerar om en array är sorterad. Metoden ska alltså ta en heltalsarray som argument och returnera `true` om elementen är sorterade i stigande ordning, annars returnera `false`.
5. Använd ovannämnda metod för att kontrollera att arrayen är sorterad efter anropet till `sQsort(int[])`.
6. Skriv en metod som skapar en `int[]` av en viss storlek och fyller den med slumpvisa värden (använd klassen `java.util.Random`).
7. Lägg till kod kring anropet till `sQsort(int[])` som mäter hur lång tid sorteringen tar. Använd metoden i punkt 7 för att generera ganska stora arrayer, det är både roligare och nödvändigt för att kunna mäta tiden. Tips: i klassen `System` finns metoden `currentTimeMillis()`.
8. Quicksort är en såkallad “divide and conquer”-algoritm, och just denna implementation är rekursiv. Det finns en punkt, när datat är tillräckligt litet, då det lönar sig att byta till en enklare algoritm. I filen Quicksort.java finns en metod `insertionSort(int[], int, int)`. Utöka `sQsort(int[], int, int)` så att `insertionSort` anropas, och slutför sorteringen, när datat är tillräckligt litet (glöm inte att kontrollera med metoden i punkt 5 att sorteringen fortfarande fungerar.) Observera att med storleken på datat avses det intervall som utgörs av `start` och `end`. Var (ungefär) ligger denna punkt då det är fördelaktigt att byta till `insertionSort`?

Del 2

1. Implementera metoden `pQsort(int[])`, alltså en parallell version av Quicksort. Sorteringen beräknar inget värde utan utför sorteringen i den aktuella arrayen, så det är rimligt att använda sig av klassen `RecursiveAction`.
2. Precis som det i den sekvensiella versionen av quicksort finns en punkt då det lönar sig att byta till en enklare algoritm finns det i den parallella en punkt då konstanta faktorer (minnesallokering framförallt) gör att det lönar sig att byta till den sekvensiella versionen. Modifiera den parallella implementationen så att den sekvensiella anropas när datat är tillräckligt litet. Var (ungefär) ligger denna punkt då det lönar sig att byta till den sekvensiella versionen?

[1] `http://en.wikipedia.org/wiki/Quicksort`