Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Informatik

Algorithmen & Datenstrukturen (AD)

Übung: Parallelisierungsframeworks (N4)

Themen: Fork-Join Konzept und Programmiermodell, Einsatz von `RecursiveAction`, `RecursiveTask` und `CountedCompleter`; zusätzlich einige Themen aus den weiterführenden Konzepten (N3), wie Java Thread Pools
Zeitbedarf: ca. 240min.

Roger Diehl, Version 1.0 (FS 2017)

1 Performance-Messungen an Mergesort (ca. 30')

1.1 Lernziele

- Beispiele aus den Unterlagen AD Input N41 nachvollziehen und modifizieren können.
- Performance-Messungen mit dem parallelisierten Mergesort Algorithmus durchführen.

1.2 Grundlagen

Für diese Aufgabe benötigen Sie den AD Input N41 (Das Fork-Join Programmiermodell und die folgenden Abschnitte – insbesondere „Einsatz von `RecursiveAction`“). Nehmen Sie für die Performance-Messungen den abgegeben Source Code. Ferner benötigen Sie den Code aus den Sortier-Aufgaben der Übungsserie „Grundlagen, einfache Sortieralgorithmen (A1)“.

1.3 Aufgabe

Das Herzstück des parallelisierten Mergesort Algorithmus ist die Methode `compute` der Klasse `SortTask`. Dort wird als erstes die Schwelle zur sequentiellen Ausführung überprüft. In den Unterlagen wurde die Schwelle `THRESHOLD` willkürlich festgelegt. Aber die Auswahl der Grösse des Basisfalles ist beim Fork-Join Konzpet kritisch. Dies wollen wir nachvollziehen.

- Ergänzen Sie den abgegeben Source Code mit einer Performance-Messung für den parallelisierten Mergesort Algorithmus. Vergrössern Sie das zu sortierende Array massiv (auf mehrere Hundert-Millionen Elemente).
- Führen Sie Messungen für verschiedene `THRESHOLD` Schwellen durch. Protokollieren und vergleichen Sie die Messungen. Gehen Sie dabei systematisch vor.
- Führen Sie Performance-Messungen mit einem schnellen sequentiellen Sortier-Algorithmus aus der Übungsserie „Grundlagen, einfache Sortieralgorithmen (A1)“.

1.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Was stellen Sie bei den Performance-Messungen mit verschiedenen `THRESHOLD` Schwellen fest?
    - Welche `THRESHOLD` Schwelle ist optimal?
    - Gibt es eine Formel oder einen Algorithmus mit dem Sie die `THRESHOLD` Schwelle für Ihren Computer bestimmen können?
- Was stellen Sie beim Vergleich paralleles vs. sequentielles Sortieren fest?

2 Quicksort parallelisieren (ca. 90')

2.1 Lernziele

- Einen klassischen Divide-and-Conquer-Algorithmus mit Hilfe des Fork-Join-Frameworks implementieren können.
- Den Sortieralgorithmus "Qicksort" verstehen und parallelisieren können.

2.2 Grundlagen

Für diese Aufgabe benötigen Sie den AD Input N41 (Das Fork-Join Programmiermodell und die folgenden Abschnitte – insbesondere „Einsatz von `RecursiveAction`“). Zudem benötigen Sie den AD Input A21 (Quicksort), die Resultate aus den Übungen „Höhere Sortieralgorithmen (A2)“ und die Hilfe des Internets.

2.3 Aufgabe

In dieser Aufgabe wollen wir richtig Gas geben. Erstellen Sie mit Hilfe des Fork-Join-Frameworks einen parallelisierten Quicksort. Sie haben dabei mehrere Möglichkeiten:

- Sie nehmen den Source Code aus dem AD Input A21 – Quicksort und parallelisieren diesen.
- Sie nehmen Ihre Lösung Quick-Insertion-Sort der Aufgabe 3 aus der Übungsserie „Höhere Sortieralgorithmen (A2)“ und parallelisieren diese.
- Suchen Sie im Internet nach parallelisierten Quicksort Verfahren. **Aber Achtung**: Viele der Algorithmen und/oder Source Codes laufen entweder nicht richtig, strapazieren den Stack zu sehr, laufen nicht parallel oder sind schlicht keine Quicksort Verfahren. Aber es gibt korrekte Lösungen, man muss sie nur finden und eventuell anpassen. Benutzen Sie in jedem Fall das Fork-Join-Framework für die Umsetzung. Alles andere ist viel zu kompliziert und nicht effizient!

Egal für welche Variante Sie sich entscheiden, prüfen Sie, ob der Quicksort Algorithmus nicht nur richtig funktioniert, sondern auch **wirklich parallel läuft**. Führen Sie Performance-Messungen mit Ihrem parallelisierten Quicksort und dem Quick-Insertion-Sort durch. Machen Sie auch Vergleichsmessungen mit dem Java-eigenen `Arrays.sort` (der übrigens ein Dual-Pivot Quicksort ist).

Sie werden Interessantes feststellen.

Optional: Geben Sie noch mehr Gas!

2.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Um welchen Faktor müsste Ihr parallelisierter Quicksort schneller sein als der sequentielle Quicksort?
- Was stellen Sie bei den Performance-Messungen fest?
- Wie erklären Sie sich das Verhalten des Ihres parallelisierten Quicksort?
- Wie könnte man Quicksort (parallelisiert) noch mehr beschleunigen?

3 Fibonacci-Zahlen (ca. 60')

3.1 Lernziele

- Eine einfache Rekursion mit Rückgabewert mit Hilfe des Fork-Join-Frameworks implementieren können.
- Rekursionsbasis und Rekursionsvorschrift identifizieren und umsetzen können.

3.2 Grundlagen

Diese Aufgabe setzt die Aufgabe 4 „Fibonacci-Zahlen“ aus der Übungsserie „Einführung (E1)“ voraus. Zudem benötigen Sie den AD Input N41 (Das Fork-Join Programmiermodell und die folgenden Abschnitte).

3.3 Aufgabe

Die Fibonacci-Zahlen rekursiv zu berechnen ist naheliegend und elegant.

```
/**
 * Berechnet den Fibonacci Wert für n.
 *
 * @param n für die Fibonacci Berechnung.
 * @return Resultat der Fibonacci Berechnung.
 */
public static int fiboRecursive(final int n) {
    return n > 1 ? fiboRecursive(n - 1) + fiboRecursive(n - 2) : n;
}
```

Aber auch suboptimal, weil insbesondere wiederholt dieselben Zwischenresultate berechnet werden.

- Erstellen Sie deshalb mit Hilfe des Fork-Join-Frameworks eine parallelisierte Lösung. Welche abstrakte Klasse wählen Sie?
- Messen Sie die Performance zwischen Ihrer Fork-Join-Lösung, der einfachen rekursiven Lösung (Beispiel oben) und der iterativen Lösung.

3.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Welche Variante (einfach rekursiv, parallel rekursiv, iterativ) ist am schnellsten?
- Was stellen Sie bei der Performance fest?
- Wie erklären Sie sich die Resultate aus der Performance-Messung?
- Haben Sie für die Fibonacci-Berechnung den `ForkJoinPool` oder den Common Pool benutzt?
    - Welcher Pool ist für die Fibonacci-Berechnung die bessere Wahl?
    - Merken Sie einen Unterschied?

4 Finde das File (ca. 60')

4.1 Lernziele

- Eine einfache Rekursion mit Hilfe des Fork-Join-Frameworks implementieren können.
- Eine parallele Rekursion nach gefundenem Resultat beenden können.

4.2 Grundlagen

Für diese Aufgabe benötigen Sie den AD Input N41 (Das Fork-Join Programmiermodell – Einsatz von `CountedCompleter`).

4.3 Aufgabe

Das rekursive Suchen nach einem bestimmten File in einer Verzeichnis-(Baum)Struktur ist einfach zu implementieren. Wie die Suche gemacht wird, könnte so aussehen:

```
public static void findFile(final String name, final File dir) {
    final File[] list = dir.listFiles();
    if (list != null) {
        for (File file : list) {
            if (file.isDirectory()) {
                findFile(name, file);
            } else if (name.equalsIgnoreCase(file.getName())) {
                LOG.info(file.getParentFile());
            }
        }
    }
}
```

Sobald das File gefunden wurde, wird hier sein Verzeichnis in den LOG geschrieben. Eigentlich könnte man dann die Suche abbrechen. Wird sie aber nicht. Zudem dauert die Suche bei grossen Verzeichnissen ziemlich lange.

Parallelisieren Sie die Suche und beenden Sie die Suche nach dem ein Verzeichnis gefunden wurde. Geben Sie das gefundene Verzeichnis als String zurück.

Die ideale Umsetzung gelingt mit der Klasse `CountedCompleter` aus dem Fork-Join Modell. Diese Klasse bietet mit der Methode `quietlyCompleteRoot()` die Möglichkeit die parallele Rekursion vorzeitig abzubrechen. Denken Sie daran: Resultate können parallel gefunden werden.

Optional: Geben Sie alle gefundenen Verzeichnisse als String-Liste zurück. Ist die Klasse `CountedCompleter` immer noch hilfreich?

4.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Was stellen Sie bei der Performance fest?
- Erhalten Sie immer die gleichen Resultate (Verzeichnisse) bei mehreren gleichnamigen vorhandenen Files?
- Wie erklären Sie sich dieses Verhalten?

5 Optional: Türme von Hanoi (ca. 60')

5.1 Lernziele

- Eine einfache Rekursion mit Hilfe des Fork-Join-Frameworks implementieren können.
- Grenzen des Fork-Join-Frameworks für rekursive Aufgaben erkennen.

5.2 Grundlagen

Diese Aufgabe benötigt den AD Input E12 (Beispiel «Türme von Hanoi»). Zudem benötigen Sie den AD Input N41 (Das Fork-Join Programmiermodell und die folgenden Abschnitte).

5.3 Aufgabe

Eine klassische Aufgabe für eine Rekursion ist die Aufgabenstellung der «Türme von Hanoi». Hier der Auszug aus dem AD Input E12:

```
public static void moveDisks(final String from, final String via,
                             final String to, final int n) {
    if (n == 1) {
        System.out.println("move disk from " + from + " to " + to);
    } else {
        moveDisks(from, to, via, n - 1);
        System.out.println("move disk from " + from + " to " + to);
        moveDisks(via, from, to, n - 1);
    }
}
```

```
public static void main(final String[] args) {
    moveDisks("A", "B", "C", 4);
}
```

Erstellen Sie mit Hilfe des Fork-Join-Frameworks eine parallelisierte Lösung. Welche abstrakte Klasse wählen Sie? Führen Sie Ihre parallelisierte Lösung aus, wählen Sie dazu eine kleine Anzahl Scheiben (z.B. wie oben).

5.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Was stellen Sie bei der Ausführung Ihrer parallelisierten Lösung fest?
- Wie erklären Sie sich das Verhalten?
- Können Sie sich eine andere Variante der parallelisierten Lösung vorstellen, wo das beobachtete Verhalten anders wäre? (Sie müssen keinen Code schreiben)