Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Programming
Assignment 5: Task Parallelism
Spring Semester 2017

Assigned on: **21.03.2017** Due by: **27.03.2017**

Overview

This week's assignment is about Task Parallelism. Task parallelism emphasizes the distributed (parallelized) nature of the processing (i.e., threads), as opposed to the data (data parallelism). We will use the fork/join framework to implement task parallelism. The fork/join framework helps to take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

The first step for using the fork/join framework is to identify the simplest task. Your code should look similar to the following pseudo-code:

```
if (work is small)
  do the work directly
else
  split work into pieces
  invoke the pieces and wait for the results
```

In Section 1, we will encounter a real-world situation were you are hired to parallelize code using the fork/join framework. You will have to make yourself familiar with the given code and find a way to parallelize it.

Section 2, you will be presented with a sorting algorithm and implement it a sequential form, as well as in a parallel form.

Getting Prepared

- Download the ZIP file named `assignment5.zip` on the course website.
- Import the project in Eclipse: Click on *File* in the top-menu, then select *Import*. In the dialog, select *Existing Projects into Workspace* under the *General* directory, then click on Next. In the new dialog, select the radiobox in front of *Select archive file* to import a ZIP file. Then, click Browse on the right side of the text-box to select the ZIP file you just downloaded from the website (`assignment5.zip`). After that, you should see assignment5 as a project under *Projects*. Click Finish.
- If you have done everything correctly, you should now have a project named assignment5 in your *Package Explorer*.

In the previous exercises, there's a close connection between the task being done by a new thread, as defined by its Runnable Java Object, and the thread itself, as defined by a Thread object. This works well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application.

One approach for breaking a problem into smaller and smaller pieces is the join/fork framework. The `java.util.concurrent.ForkJoinPool` class is a good framework to deal with this type of parallelism. This framework is designed to work with multi-core systems, ideally with dozens or hundreds of processors. The usual user platforms (desktop/laptops) lack full support for this type of concurrency, but future machines are envisioned to have it. In other words with fewer than four processors, there will be little performance improvement. Objects that encapsulate these functions are known as executors. The following subsections describe executors in detail.

This package is part of the `java.util.concurent` **`package`**, about which you can have an overview here:

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html

In this exercise you will use the `ForkJoinPool` class that allows execution of `ForkJoinTask`s. In particular, we will use two subclasses of `ForkJoinTask` – `RecursiveTask` for Section 1 and `RecursiveAction` for Section 2.

1 Search and Count

When hiring new people for a job position, the first task is the screening of CVs:

The Human Resources (HR) manager has to read all the CVs which have been previously validated by the automatic CV checker. The manager checks whether the applicants have a specific set of required qualities. If the applicant is qualified, then they mark those CVs as being valid for a certain job. However, some features are more easy to check then others. For example, it is easier to get the age of the candidate based on the birth year, than to evaluate if the candidate has the necessary technical and social skills a job requires. We can say that the first task requires a *light* workload for each candidate, while the second task requires a *heavy* workload.

In our case, the CVs for the HR manager are randomly generated numbers and the desired features can be faster or slower to compute. We classify two types of workloads: `HEAVY` and `LIGHT` (e.g. check if the number is non zero - light workload and check if the number is prime - heavy workload). The code to check the features is given to you and should be not changed (it is located in the class `Workload` in the `doWork` function). The task is to count for each feature how often it appears in the given input.

Task 1: The HR manager knew about the advantages to automate the CV screening process and bought a single threaded program to solve his tasks. The relevant code is located in the `SearchAndCountSingle` class. But with a rising number of applicants and long computation time the HR manager hopes that the program can be made faster.

The HR manager hired you to create a multi-threaded version that helps the HR manager to screen through the CVs faster. Your goal is to implement the `SearchAndCountMultiple` – a multi-threaded version of `SearchAndCountSingle` – using the join/fork framework.

Task 2: The HR manager obviously wants tentative proof that your program really computes the same results as the original sequential implementation. Being a knowledgeable software developer trained at ETH you easily spot the *JUnit* test class the previous developers left behind in `SearchAndCountTest`. Use them to verify that your implementation of `SearchAndCountMultiple` indeed computes the same results as the original implementation `SearchAndCountSingle`.

Task 3: Next, the HR manager wants you to show him that your program indeed works faster. Further, he does not want to meddle with its settings. Find the value for the `cutOff` (i.e., the threshold that controls whether the work is split into smaller tasks or solved directly), and the number of threads that maximize the speedup for input size $100,000$ and for both workloads (`LIGHT` and `HEAVY`). Include the optimal parameters and how you found them in your report.

2 Sorting

In this exercise you are asked to implement merge-sort algorithm using task parallelism. The merge-sort algorithm sorts an array by partitioning it in smaller arrays. Once the size of the arrays becomes 1 or 2, they are trivially sorted. Sorted sub-arrays are combined by *merging* them.

The simplest task in merge-sort is the trivial sort of small arrays. This is the following:

```
if (array.length == 1) {

  // An array of size 1 is implicitly sorted
  return array;

} else if (array.length == 2) {

  // Re-arrange elements if they are not in proper order
  if (array[0] > array[1]) {
    temp = array[0];
    array[0] = array[1];
    array[1] = temp;
  }
  return array;

}
```

Merging two sorted arrays into a bigger sorted array is also a simple procedure. Consider the following two sorted arrays:

```
array1: [5, 11, 12, 18, 20]
array2: [2, 4, 7, 11, 16, 23, 28]
```

The resulting merged array is computed by comparing the head of the two arrays. The smallest one is removed and it is appended at the end of the output. In this case, the merged output is:

```
result: [2, 4, 5, 7, 11, 11, 12, 16, 18, 20, 23, 28]
```

The code to merge two ordered arrays is already provided to you in `ethz.ch.pp.util.ArrayUtils`. Futher, we also provide an sequential merge-sort in `ethz.ch.pp.mergeSort.MergeSortSingle`.

Your task is to implement the task-parallel version of merge-sort.

Task 1: The code you downloaded and imported contains relevant parts in `MergeSortMulti` that you are expected to implement. For merge-sort with task parallelism, the following is one of the possible strategies to divide the work:

```
sort(input) {

  if (input.length <= 2) {

    // Execute the simple task locally
    return simpleSort(input);

  } else {

    // Split the input in two parts by forking to two tasks

    fork(firstPart);
    fork(secondPart);

    waitForResults();

    // Join results
    return merge(firstPartResults, secondPartResults);

  }

}
```

Implement the strategy outlined above using the `ForkJoinPool` and `RecursiveAction` Java APIs.

Task 2: Verify that the multi-threaded merge-sort computes the correct results by running the provided set of test-cases.

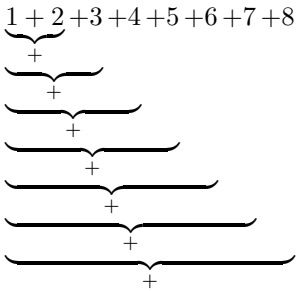
Questions: Besides implementing the parallel version of merge-sort you are also asked to include the answer to these two questions in your report:

- What is the run-time complexity of this algorithm ?
- What is the speedup of parallelism (time) ?

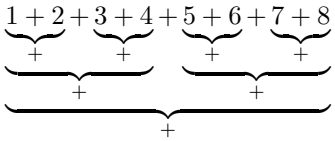
EXTRA CHALLENGE Think of how much work are you doing in simple task? Can you increase the work done there? What impact will it have on speedup? Write up and plot your evaluations with different size of work in the simple task, and explain the behavior.

3 Task Graph

Assuming you want to add eight numbers, then two options to do this are



and



a) Given those two variants, determine the length of the critical path for both computations

b) For a sequence of length n, determine the length of the critical path using the two approaches from above (accumulator method and divide and conquer).

Submission

In order for us to grade your exercises and give you feedback, you need to submit your code to the Subversion repository. You will find detailed instructions on how to install and set-up Eclipse for use with Subversion in Exercise 1.

Once you have completed the skeleton, commit it to SVN in a directory named `assignment5` by following the steps described below. The questions that require written answers should all be recorded in a single file named `report.pdf` and placed in the base directory of your project (i.e., in folder `assignment5`).

- **Check-in your project for the first time**
  - Right click your created project called **assignment5**.
  - In the menu go to **Team**, then click **Share Project**.
  - In the dialog that now appears, select **SVN** as a repository type, then click **Next**.
  - In case you have submitted Exercise 1, choose **Use existing repository location** and select the pre-defined URL in the dialog that should look like this
    https://svn.inf.ethz.ch/svn/vechev/pprog17/students/NETHZ_USERNAME
    Click Finish. Otherwise follow the steps in Exercise 1 to set-up a repository location.
- **Commit changes in your project**
  - Now that your project is connected with the SVN server, you need to make sure that every time you change your code or your report, at the end you submit it to the SVN server as well.
  - Right click your project called **assignment5**.
  - In the menu go to **Team**, then click **Commit**.
  - In the Comment field, enter a comment that summarizes your changes.
  - Then, click on **Ok**.