M1 Info/MOSIG: Parallel Algorithms and Programming

Spring 2024

Lab sheet #2: Open MP

20.02.2024

Contents

1) About this lab

- This assignment will be graded.
- The assignment is to be done by groups of at most **2** students.
- Deadline: **March, 17, 2024 at 23:59 CET (UTC+1/Paris)**.
- The assignment is to be turned in on Moodle

1.a) Collaboration and plagiarism

You are encouraged to discuss ideas and problems related to this project with the other students. You can also look for additional resources on the Internet. However, we consider plagiarism very seriously. Hence, if any part of your final submission reflects influences from external sources, you must cite these external sources in your report. Also, any part of your design, your implementation, and your report should come from you and not from other students/sources. We will run tests to detect similarities between source codes. Additionally, we will allow ourselves to question you about your submission if part of it looks suspicious, which means that you should be able to explain each line you submitted. In case of plagiarism, your submission will not be graded and appropriate actions will be taken.

1.b) Running OpenMP programs

Different solutions exists to compile and execute your OpenMP programs. Each comes with advantages and drawbacks. We briefly describe each of them hereafter.

On your laptop OpenMP programs can be compiled and executed on any Linux machine. OpenMP is supported by default by `gcc`.

The main limitation when running on your laptop is the limited number of physical cores that can be used for performance evaluation.

On the server of the university You can run your OpenMP programs on the UGA server called `Mandelbrot`.

To connect to this server, simply run:

```
ssh [LOGIN]@im2ag-mandelbrot.univ-grenoble-alpes.fr
```

Running on `Mandelbrot` allows you to run on a server with a large number of cores. The main drawback is that the activity of other users can impact your performance measurements, as the platform is shared with other users.

On Google Cloud

- The instructions to get your Google Cloud Education credits will be published in the coming days on Moodle.
- A tutorial dedicated to running OpenMP applications on Google Cloud is available here: `https://roparst.gricad-pages.univ-grenoble-alpes.fr/cloud-tutorials/openmp/`.

Using Google Cloud, you can run tests in environments with a significant number of cores while being isolated from other users.

2) Your submission

Your submission is to be turned in on Moodle as an archive named with the last name of the two students involved in the project: `Name1_Name2_lab2.tar.gz`.

The archive will include:

- A report (in `pdf` or `markdown` format[1]) that should include the following sections:
  - The name of the participants.
  - For each exercise:
    - A short description of your work including:
      - The completed tasks
      - The limitations and known bugs, if any.
    - The answer to questions (when applicable)
    - A performance analysis and graphs (when applicable)
- Your source code for each exercise

3) Bubble Sort (6 points)

During this lab, you will implement several algorithms to sort an array of integer elements. You will integrate the *bubble sort* algorithms in the file `bubble.c`.

Bubble sort is one of the most inefficient sorting algorithms. However, it is probably the simplest to understand. At each step, if two adjacent elements of an array are not in order, they will be swapped. Thus, smaller elements will "bubble" to the front, (or bigger elements will be "bubbled" to the back, depending on implementation) and hence the name.

The bubble sort algorithm can be simply described by the following algorithm for an array T of size N to be sorted:

```
do
  sorted = true
  for i in 0..N-2:
    if T[i] > T[i+1]:
      swap T[i] and T[i+1]
      sorted = false
while (sorted == false)
```

[1]Other formats will be rejected

(a) Implement the sequential bubble sort algorithm.

(b) Implement the parallel version of the algorithm.

- Hint:The idea of the parallel version of the bubble sort algorithm is to split the array into several chunks and to have one *bubble* running in each of the chunks. In each iteration, after the parallel step, the elements at the border of two consecutive chunks have to be compared and possibly swapped. A high-level description of the algorithm is as follows:

```
while the vector is not sorted:
   for all chunks in parallel:
       execute bubble sort
   for all chunks in parallel:
       swap the values at the border of the chunks
```

Comment 1 Use the scheduling policy that seems to you the more appropriate for this problem.

Comment 2 The provided program (described in file `bubble.c`) takes one parameter N which is the size of the array to be sorted: At the beginning of the program an unsorted array of size 2^N is created.

Comment 3 You are provided with a Makefile to compile your code. By default, the array to be sorted is initialized in such a way that it contains elements sorted in descending order (worse case). The variable `RAND_INIT` can be used to test the algorithm on a randomly filled array (call to `make -B RAND_INIT=1`).

4) MergeSort with tasks (5 points)

We will now focus on the `mergesort` algorithm. The sequential specification of the `mergesort` algorithm makes it a good candidate for a parallel implementation using tasks.

The sequential `mergesort` algorithm is as follows; its execution is illustrated in Figure 1.

(a) If the input sequence has fewer than two elements, return.

(b) Partition the input sequence into two halves.

(c) Apply the `mergesort` algorithm to the two subsequences.

(d) Merge the two sorted subsequences to form the output sequence.

The code of a `merge` operation is provided to you in the file `mergesort.c`.

In Figure 1, the `MergeSort` is used to sort the sequence 9, 4, 6, 2. The two partitioning phases **P** split the input sequence. The two merging phases **M** combine two sorted subsequences generated in the previous phases.

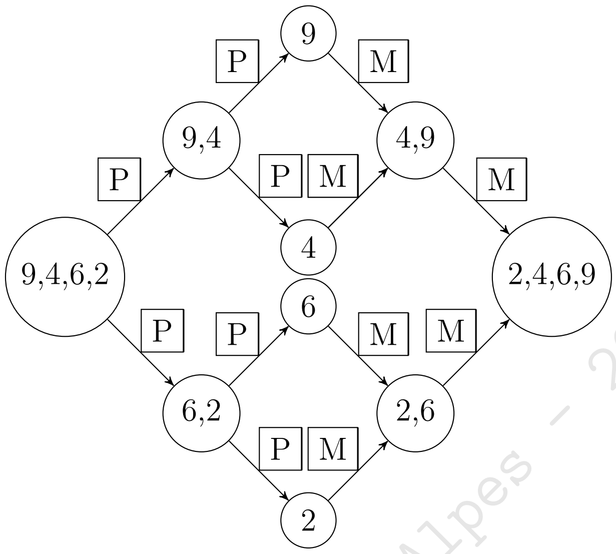

Figure 1: Mergesort execution on a sequence of four elements.

(a) Implement the sequential `MergeSort` function in file `mergesort.c`.

(b) Implement the parallel algorithm *Mergesort* using OpenMP tasks.

(c) Propose (= describe in your report) a modification to improve the performance of your parallel algorithm. Implement this modification to check its impact on performance.

5) Odd-even sort (4 points)

As you may have noticed, the specification of the *bubble* sort algorithm makes it difficult to parallelize it.

In this exercise, we will study the *odd-even* sort algorithm. This algorithm can be seen as very similar to *bubble* sort. Indeed, the idea is still to iterate over the elements of the array and swap adjacent elements if needed. The difference is that instead of iterating over the elements one-by-one, iterating is done with a step of 2. To ensure that all elements will be sorted, it is then required alternate between iterating over the odd and even indexes.

The odd-even sort algorithm can be described by the following algorithm for an array T of size N to be sorted:

```
do
  sorted = true
```

```
        for i in 0..N-2 with step 2:
          if T[i] > T[i+1]:
            swap T[i] and T[i+1]
            sorted = false
        for i in 1..N-2 with step 2:
          if T[i] > T[i+1]:
            swap T[i] and T[i+1]
            sorted = false
      while (sorted == false)
```

(a) Explain (in your report) why the odd-even sort algorithm is more adapted than the bubble sort algorithm for a parallel implementation.

(b) Implement the sequential odd-even sort function in file `odd-even.c`.

(c) Implement the parallel odd-even sort function in file `odd-even.c` using OpenMP.

6) Performance analysis (5 points)

For each of the studied sorting algorithm, run an evaluation of its performance.

Suggestions Here are a few suggestions about how to conduct your performance evaluation:

- You can plot one or several graphs to analyze and compare the performance of algorithms, including:
  - A figure with the speedups (compared to sequential execution) for increasing number of threads (For instance: 2, 4, 8, 16 threads).
  - A figure with the execution time as a function of the problem size for different number of threads.
  - Other graphs than can illustrate some important characteristics of your algorithms.

Take into account the characteristics of the processor you run the experiments on to analyze the results you obtain.

Important: You should include a description of the processor you are running on in your report.

Plotting graphs To help you in this task, we provide you with a python script based on Matplotlib (`https://matplotlib.org/`) to generate graphs out of data stored in a CSV file.

The script `plotting_data.py`, together with an example of input file (`data.csv`) is provided in the directory `plotting`.

The main information to use this script are:

- A file `requirements.txt` is provided to help you installing the packages the script depends on. Simply run:

  ```
  pip3 install -r requirements.txt
  ```

- The provided script takes a CSV file as input. Data stored in a spreadsheet can simply be stored as CSV by editors such as `LibreOffice`.
  - The script assumes that the delimiter is ';'.
- To generate the graph corresponding to the data stored in the file `data.csv`, run:

  ```
  python3 plotting_data.py -i data.csv
  ```

- Additional information:
  - The script assumes that the first column corresponds to the indexes on the x-axis.
  - The script saves the generated graph as a pdf file with the same filename as the input (`data.pdf`). Use option `-o` to change the name of the output file.

Feel free to modify the provided script to adapt it to your needs or to use other tools for generating graphs if your prefer.

7) Bonus step: Quick sort algorithm

The *Quick Sort* algorithm is implemented by the `qsort` function of `libc` library.

```
#include <stdlib.h>

  void qsort (void *base, size_t nmemb, size_t size,
             int (*compar)(const void *, const void *));
```

The `qsort()` function[2] sorts an array with `nmemb` elements of size `size`. The `base` argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by `compar`, which is called with two arguments that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

To implement the parallel version of *Quicksort*, the principle is to split the array into several chunks, to sort each chunk separately, and finally, to merge consecutive chunks. Some parallelism can also be found during the merging phase: pairs of chunks can be merged in parallel.

Comment The code of the `merge` function used for *mergesort* can be reused here.

[2] `man 3 qsort`