Université Grenoble Alpes, Grenoble INP, UFR IM^{2}AG

Master 1 Informatique and Master 1 MOSIG

UE Parallel Algorithms and Programming

Lab 2 – 2019

1 Important information

- This assignment will be graded.
- The assignment is to be done by groups of at most **2** students.
- Deadline: **March, 4, 2019**.
- The assignment is to be turned in on Moodle (M1 Mosig Course: Parallel Algorithms and Programming, pwd: PAP- -2019)
- The number of points associated with each exercise is only here to give you an idea of the weight associated with each exercise (The maximum grade is 20, there are extra points).

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` format[1]) that should include the following sections:
  - The name of the participants.
  - For each exercice:
    - A short description of your work
    - Answers to questions (when applicable)
    - A performance analysis and graphs (when applicable)
- Your source code for each exercice

[1]Other formats will be rejected

3 OpenMP Loop Scheduling (5 points)

M is a lower triangular matrix. It means that all the values of the matrix above the diagonal are zero. All the values below (and also) the diagonal are useful for the computation. We are going to study the matrix-vector product. You are provided with the sequential implementation of the matrix-vector product optimized for lower triangular matrices[2] (see file `triangular_matrix.c`).

1. Implement the parallel OpenMP function `mult_mat_vect_tri_inf1` with **static** scheduling.
2. Implement the parallel OpenMP function `mult_mat_vect_tri_inf2` with **dynamic** scheduling.
3. Implement the parallel OpenMP function `mult_mat_vect_tri_inf3` with **guided** scheduling.
4. Draw a figure with the speedups (compared to sequential execution) for 2, 4, 8 and 16 threads.
5. Comment the performance you obtain (taking into account the characteristics of the processor you run the experiments on may help).
6. **Bonus:** Study the impact of the chunk size on the performance results. For this, you can use the **runtime** schedule to set the chunk size at runtime.

4 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
   swapped = false
   for i in 0..N-1
     {
      if T[i] > T[i+1]
        {
          swap T[i] and T[i+1]
          swapped = true
        }
     }
while (swapped == true)
```

7. Implement the sequential bubble sort algorithm.
8. Implement the parallel version of the algorithm.

Comment 1 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.

[2]To optimize performance, the idea is to exclude from the computation the entries that are zero by construction.

Comment 2 Use the scheduling policy that seems to you the more appropriate for this problem.

Comment 3 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.

5 Quick Sort (6 points)

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[3] 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.

9. In file `qsort.c`, use the `qsort` function to implement a sequential sorting function. This requires implementing the `compare` function first.
10. Implement the parallel version of the *Quicksort* algorithm. In this first step, the merging part (see Comment 4) will be done sequentially.
11. Implement the fully parallel version of the *Quicksort* algorithm.
12. Comment on the performance difference you observe between the different versions of the algorithm.

Comment 4 To implement the parallel version of the *Quicksort* algorithm, the principle is to split the array into several chunks, to sort each chunk separatly, and finally, to merge consecutive chunks.

Comment 5 The code of a `merge` function is provided to you in the file `qsort.c`.

6 MergeSort with tasks (5 points)

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

1. If the input sequence has fewer than two elements, return.
2. Partition the input sequence into two halves.
3. Apply the `mergesort` algorithm to the two subsequences.
4. Merge the two sorted subsequences to form the output sequence.

[3] `man 3 qsort`

The `Merge` operation employed in step (4) combines two sorted subsequences to produce a single sorted sequence. The same `merge` function as in Exercise 5 can be reused.

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.

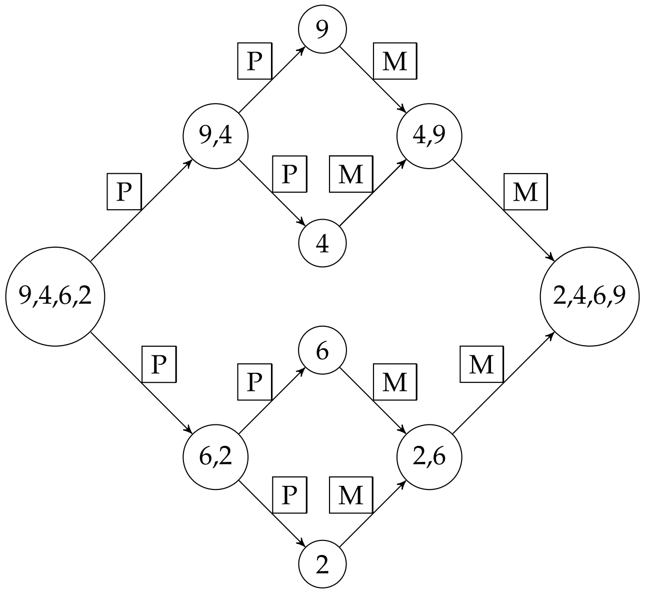

Figure 1: Mergesort execution on a sequence of four elements.

11. Implement the sequential `MergeSort` function in file `mergesort.c`.
12. Implement the parallel algorithm *Mergesort* using OpenMP tasks.
13. Depending of the array size (N), how many tasks are created?
14. Propose a modification of the algorithm to reduce the number of created tasks