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

Master 1 Informatique and Master 1 MOSIG

Parallel Algorithms and Programming

Lab 1 – 2022

This lab is an introduction to programming with OpenMP.

Main information about this lab:

- This lab is not graded. You don't have to submit anything at the end of this lab.

1 Compiling and executing OpenMP programs

Before jumping into the exercises, a few comments about the requirements to be able to compile and execute OpenMP programs.

On your laptop

OpenMP is supported by default by the the main C compilers (including `gcc` and `clang`). Hence, if you run on a machine with Linux, you should be able to compile and execute OpenMP programs without any problems. It might also work on MacOS.

In the lab rooms of UFR

The compilers `gcc` or `clang` can be used to compile OpenMP programs on the machines from the lab rooms.

Connecting to the UFR server using ssh

An alternative is to run this lab on the server of the university called `Mandelbrot`.

To connect to this server, simply run:

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

This server also has the advantage to give you access to a machine with more cores. However, this is a virtualized shared platform. The activity of others can impact your performance evaluations and the platform might not deliver very good performance.

2 Resources for OpenMP

Beyond the lecture slides, you can find plenty of resources regarding OpenMP on the Web. Although they might look complex at first sight, the documents published by the OpenMP Board are the main source of information:

- The Reference Guide summarizes all the directives included in OpenMP: `https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5-2-web.pdf`
- If you have questions about the exact semantic of a directive or an option, answers can be found in the API specification: `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf`

3 Compiling and executing OpenMP programs

The source files for this lab are in the `lab1.tar.gz` archive available on Moodle[1].

To compile the code of the first exercise, run:

```
gcc -O0 -fopenmp -o ex1 ex1.c
```

The `ex1` program takes one argument that corresponds to the maximum number of threads to be used by OpenMP.

```
$ ./ex1 8
I am the master thread 0 and I start
Starting Region 1
Region 1  thread 0 of team 8 (max_num_threads is 8)
Region 1  thread 7 of team 8 (max_num_threads is 8)
Region 1  thread 4 of team 8 (max_num_threads is 8)
Region 1  thread 2 of team 8 (max_num_threads is 8)
Region 1  thread 1 of team 8 (max_num_threads is 8)
Region 1  thread 6 of team 8 (max_num_threads is 8)
Region 1  thread 5 of team 8 (max_num_threads is 8)
Region 1  thread 3 of team 8 (max_num_threads is 8)
End of Region 1
Starting Region 2
Region 2  thread 3 of team 4 (max_num_threads is 8)
Region 2  thread 0 of team 4 (max_num_threads is 8)
Region 2  thread 2 of team 4 (max_num_threads is 8)
Region 2  thread 1 of team 4 (max_num_threads is 8)
End of Region 2
Region 3  thread 0 of team 2 (max_num_threads is 8)
Region 3  thread 1 of team 2 (max_num_threads is 8)
End of Region 3
I am the master thread 0 and I complete
$
```

1. Explain the output of this OpenMP program.
2. Compare and analyze the output of different runs of this program.

4 Performance Analysis of Vector and Matrix Operations

In this exercise, we will study how basic numerical functions can be simply parallelized with OpenMP. The functions to be studied are defined in file `ex2.c`.

[1]To extract the archive, you can execute: `tar zxvf lab1.tar.gz`

A `vector` and `matrix` types are defined in this file. The size of the manipulated vectors and matrices are defined by the single argument taken by the program.

The `init_vector()` and `init_matrix()` functions allocate and initialize vectors and matrices. The computing functions we are going to study are performing the following operations:

- Operations on vectors:
  - addition of two vectors (`add_vectors`)
  - scalar product of two vectors (`dot`).
- Operations on matrices and vectors
  - Multiplication between a matrix and a vector (`mult_mat_vector`).
- Operations on matrices
  - multiplication between two matrices (`mult_mat_mat`).

The main function of this `ex2` program calls the different functions.

For each call, the number of processor cycles is measured using the intrinsic _rdtsc (). Several runs are performed to compute an average.

Operations on vectors Several implementations of the operations on vectors are provided to you.

1. Measure the execution time of each function in cycles (measuring the number of cycles is already implemented)
2. Modify the main function to compute the number of floating point operations per second (MFLOPS or GFLOPS) achieved by each function.
   - To measure the FLOPS, you need to know the execution time and the total number of floating point operations executed by each function.
   - You can deduce the number of floating point operations from the size of the vectors and the operations executed inside the functions.
   - Deducing the time from the number of cycles can be tricky, especially because the frequency of modern processors vary with the load. To simplify the problem, we will assume that the processor always runs at its nominal frequency.
   - You can get the nominal frequency of your processor in the file `/proc/cpuinfo`.
3. Analyze the speedups with 2, 4, 8 and 16 threads[2].
   - In `bash`, setting the environment variable can be done with: `export OMP_NUM_THREADS=2`
4. Compile with the `-O2` option (Optimization level 2). Analyze the performance results with this option.

Operations on matrices The operations on matrices are not yet implemented. It is your work to implement them.

1. Implement the multiplication functions between a matrix and a vector (`mult_mat_vector`), and run the same evaluation as before.
2. Implement the multiplication between functions two matrices (`mult_mat_mat`, and run the same evaluation as before.

[2]You can select the number of threads to be used by using the environment variable `OMP_NUM_THREADS`.

5 OpenMP Loop Scheduling

M is a lower triangular matrix. It means that all the values of M 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 thanks to the code provided in `ex3.c`. The goal of this exercise is to observe the impact of the different loop scheduling policies on such a problem.

A default size is used for the vector and the matrix in the provided code. However, to run meaningful performance tests, you might want to use bigger problem sizes. The problem size can be specified as a parameter of the program `ex3`.

1. Implement the sequential version of `mult_mat_vect_tri_inf` that take advantage of M being a lower triangular [3].
2. Implement the parallel OpenMP function `mult_mat_vect_tri_inf1` with **static** scheduling.
3. Implement the parallel OpenMP function `mult_mat_vect_tri_inf2` with **dynamic** scheduling.
4. Implement the parallel OpenMP function `mult_mat_vect_tri_inf3` with **guided** scheduling.
5. Draw a figure with the speedups for 2, 4, 8 and 16 threads.
6. For the 3 scheduling policies, 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.

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