CPDS (Parallelism) Laboratory Assignment 2: Solving the Heat Equation using several Parallel Programming Models

Josep-Ramon Herrero[1]

Fall 2023

[1]Materials adapted and extended from others developped together with other instructors, mainly Professors Eduard Ayguadé and Daniel Jiménez.

Index

Index **1**

1 Sequential heat diffusion program **2**

2 Shared–memory parallelization with `OpenMP` **4**

3 Message–passing parallelization with `MPI` **6**

4 Parallelization with CUDA **8**

5 Deliverables **9**
5.1 Parallelization 9
5.2 Parallel execution 9
5.3 Source codes 9

1

Sequential heat diffusion program

In this assignment you will work on the parallelization of a sequential code that solves the heat equation. The code simulates the diffusion of heat in a solid body using several solvers for the equation (*Jacobi* and *Gauss–Seidel*). Each solver has different numerical properties which are not relevant for the purposes of this laboratory assignment; we use them because they show different parallel behaviors. The goal of this assignment is to have you learn about the parallelization of codes with different characteristics using several parallel programming models (`OpenMP`, `MPI` and `CUDA`). **Note:** We will be working with small matrices, so don't expect good speed-ups. Nevertheless, the important is to master the concepts and techniques neecessary to exploit parallelism for problems with similar patterns as the ones arising here.

The pictures below show the result of two simulations of the heat distribution when a single heat source is placed in the lower right corner (left); or two heat sources located in the top left corner and bottom respectively (right). The program is executed with a configuration file (`test.dat`) that specifies the size of the body, the maximum number of simulation steps, the solver to be used and the heat sources. The program generates performance measurements and a file `heat.ppm` providing the solution as image (as portable pixmap file format).

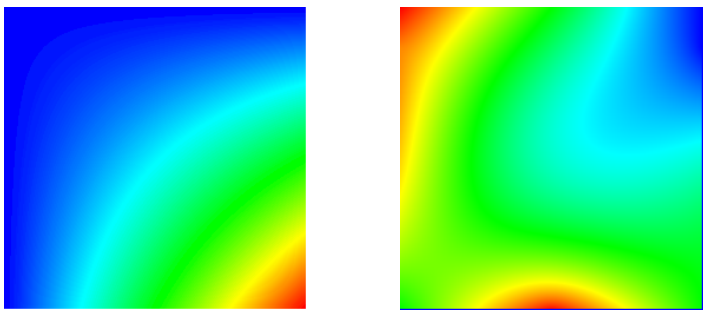

Figure 1.1: Images representing temperature in each point of a 2D solid body

1. Login into the `boada` cluster at the Computer Architecture Department at UPC. To do that open a terminal window in your laptop and connect to `boada` using `ssh -X cpds1XYZ@boada.ac.upc.edu`.

2. Extract the files from file `a2.tar.gz` from ~cpds0/sessions by uncompressing it into your home directory in `boada`. From your home directory unpack the files with the following command line: `"tar -zxvf /scratch/nas/1/cpds0/sessions/a2.tar.gz"`.

3. Access the `part1` directory. Compile the sequential version of the program using `"make heat"` and execute the binary generated by submitting a job to the queue (`"sbatch submit-seq.sh"`). The execution reports the execution time, the number of floating point operations (Flop) performed, the average number of floating point operations performed per second (Flop/s), the residual and the number of simulation steps performed to reach that residual. Visualize the image file generated with an image viewer (e.g. `"display heat.ppm"` and save it for validation purposes with a different name, e.g. `heat-jacobi.ppm`.

4. Change the solver from *Jacobi*(0) to *Gauss-Seidel* (2) by editing the configuration file provided (`test.dat`), execute again the sequential program and observe the differences with the previous execution. Note: the images generated when using different solvers are slightly different. Again, save the `.ppm` files generated with a different name, we will need them later to check the correctness of the parallel versions you will program.

IMPORTANT NOTES:

1) Contrary to the course slides, the codes implementing each numerical solver provided in all four parts already have 4 loops. This has additional overhead, and is not the most efficient in terms of execution time, neither for the sequential code nor the *Jacobi* solver, which does not require *blocking*. However, they are provided in this way in order to ease the parallelization of the *Gauss-Seidel* solver, which requires *blocking*.

2) These codes work on a number of blocs in the rows and the columns defined as:

```
#define NB 8
nbx = NB;
bx = sizex/nbx;
nby = NB;
by = sizey/nby;
```

You can/should change the definition of the blocks most appropiate to your parallelization. However, you don't need to worry about making the code general to work with a number of threads/processes that does not divide exactly the problem size. Thus, you can simply provide a parallelization which works using number of threads/processes and image resolutions which are powers of 2.

2

Shared–memory parallelization with OpenMP

In this section we will guide you through the parallelization of the heat diffusion sequential code using the `OpenMP` shared-memory paradigm, following the geometric block decomposition suggested in Figure 2.1. Go into the `part2` directory.

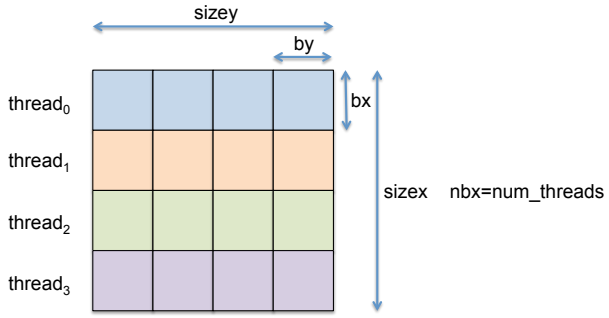

Figure 2.1: Geometric decomposition for matrix `u` (and `utmp`) in blocks

Due to the increasing complexity of the parallelization, we recommend that you start with the parallelization of the *Jacobi* solver using the `heat-omp.c` and `solver-omp.c` files that we provide you, and that you follow these steps:

1. Parallelize the *Jacobi* solver by inserting the appropriate `OpenMP` pragmas to create threads, distribute work among them and perform the necessary synchronization.

    - Compile using `"make heat-omp"` and execute with a certain number (power of 2) of `OpenMP` threads (e.g. `"OMP_NUM_THREADS=4 ./heat-omp test.dat"`). Validate the parallelization by visually inspecting the image generated and making a `diff` with the file generated with the original sequential version.

- Submit the `submit-omp.sh` script (using `"sbatch submit-omp.sh 8"`) to execute the binary in the queue system (the usual procedure in the production machine). The script executes the parallel version using from the number of threads specified as a parameter in the command line, or 8 as a default. You can check the status of your job by doing `squeue`[1]. The execution generates two files with the standard output (`submit-omp.sh.o<job_id>`) and standard error (`submit-omp.sh.e<job_id>`). Additionally, a file named `time-heat-omp-${OMP_NUM_THREADS}-${HOST}.txt` for the number of threads and the node in `boada` used. Draw a plot or do a table with the speed-up that is achieved. Is the scalability appropriate?
- Parallelize other parts of the code amenable to parallelization, or simply rewrite them in a different way, in order to reduce the execution time and improve the parallel performance. Complete the previous speed-up plot or table with the results obtained with your new version.
- Once your parallel code is correct and generates the correct image you can test the execution with 1, 2, 4, 8 and 16 threads. It can be interesting to increase the resolution to a larger value (also a power of 2). You can do so by by editing the configuration file provided (`test.dat`) and changing the 256 in the 2^{nd} line into a 512 o 1024. Remember to submit your jobs for batched execution using `sbatch`.

2. Repeat the steps in the previous bullets with the parallelization of the *Gauss–Seidel* solver.
   - Parallelize using *explicit tasks with dependencies* (`task + depend (in, out, inout)`)
   - Parallelize using a *do-across* (`ordered + depend (sink, source)`)

[1] In case you need to remove a job from the execution queue, just use `"scancel <job_id>"`.

3

Message–passing parallelization with `MPI`

Next you will parallelize the heat diffusion sequential code with [the *Jacobi* and] the *Gauss–Seidel* solvers using a distributed-memory (message-passing) paradigm: `MPI`. Since you've already got exposed to an `MPI` code working on a *Jacobi* like method in the previous *Assignment 1*, maybe you can work directly on the *Gauss–Seidel* solver. Otherwise, start with the *Jacobi* solver which should be easier to parallelize correctly, and once you have a correct parallel version for it, proceed into parallelizing the *Gauss–Seidel* solver. In both cases, it's advisable to consider the items below.

Go into the `part3` directory. We suggest that you proceed through the following versions in order to get to the final code:

1. First you should edit the `heat-mpi.c` code, containing a first parallel version, and understand how it works. On one side, the master first reads the configuration file, allocates memory and provides the necessary information to the workers. Then it computes the heat equation on the whole 2D space, reports the performance metrics and generates the output file. On the other side, each worker receives from the master the information required to solve the heat equation, allocates memory, performs the computation on the whole 2D space and finishes. Observe that this version is correct but we don't benefit from the parallel execution since workers replicate the work done in the master. Compile this version using `"make heat-mpi"` and execute the binary submitting the `submit-mpi.sh` script using `"sbatch submit-mpi.sh"`. By default it executes the program with 4 `MPI` processes. You can change it by providing a parameter from the command line. You can check the status of your job by doing `squeue`[1]. The execution generates two files with the standard output (`submit-mpi.sh.o<job_id>`) and standard error (`submit-mpi.sh.e<job_id>`).

2. Modify the parallelization so that the master and each worker solve the equation for a subset of consecutive rows (i.e. *resolution*/*numprocs*).

[1] In case you need to remove a job from the execution queue, just use `"scancel <job_id>"`.

Now workers should return the part of the matrix they have computed to the master in order to have the complete 2D data space. Compile and execute the binary generated as done before. This version should not generate a correct result (check it by comparing the `ppm` files generated with `diff`) since communication during the execution of the solver is not performed in each iteration of the `iter` while loop.

3. Add the necessary communication so that at each iteration of the `iter` while loop the boundaries are exchanged between consecutive processors. Compile and execute the binary generated as done before. This version should generate a correct solution, although not necessarily the same as the one generated by the sequential execution because the total number of iterations done in each process is controlled by the `maxiter` parameter and its local `residual`.

4. Add the necessary communication with the master so that the total number of iterations done is also controlled by the value of the global `residual` variable in all workers as it is in the sequential code. Compile and execute the binary generated as done before. This version should generate the same result as the one generated by the sequential code (check it by comparing the `ppm` files generated with `diff`).

5. Next, modify the code so that each worker just allocates the amount of memory required to do the computation, instead of allocating all the matrix as done until now. Obviously, this version should generate the same result as the one generated by the sequential code.

6. Once your parallel code is correct and generates the correct image you can test the execution with an increasing number of `MPI` processes: 1, 2, 4 and 8 processes. Please, don't execute with 8 if the executuon time with 4 is already larger than the executuon time with 2 processors. It can be interesting to increase the resolution to a larger value (also a power of 2). You can do so by by editing the configuration file provided (`test.dat`) and changing the 256 in the 2^{nd} line into a 512 o 1024. Remember to submit your jobs for batched execution using `sbatch`.

7. Finally, in view of the items above, parallelize the heat diffusion sequential code with the *Gauss–Seidel* solver. This time, since you are already familiar with the code, we will not give you any additional guidelines. Just pay attention to the **dependences** and make sure that they are fulfilled.

4

Parallelization with CUDA

In this part of the assignment you will parallelize the heat diffusion code with the *Jacobi* solver using CUDA, the programming paradigm developed by NVidia to program GPU devices. Go into the `part4` directory. We suggest that you proceed through the following versions in order to get to the final code:

1. Edit the initial version in `heat-CUDA.cu` and `kernels.cu`, identify the parts of the code that do the computation on the CPU and the GPU and understand how both work.

2. Complete the `gpu_Heat` kernel so that each invocation performs the computation of a single element in the matrix. Compile this version using `"make heat-CUDA"` and execute it with `sbatch submit-cuda.sh`. Compare the execution time on the CPU version and the GPU version.

3. Do you think the computation of the `residual` is causing the biggest overheads in the GPU version? Comment everything related with the computation of the `residual` and execute again. Do the results demonstrate our hypothesis? Observe that now you execute more iterations in much less time.

4. Implement the computation of the residual on the GPU. Review the course slides to see how to create a fast reduction kernel. Is this improving the exception time with respect to the original GPU code? And with respect to the original CPU version?

5. Finally, modify the kernel so that it makes use of the shared memory within each SM and perform the necessary modifications in the host code accordingly.

5

Deliverables

You have to deliver **TWO files**: 1) a `PDF` that contains your **report** explaining the parallelization of the heat equation code and speedups obtained; and 2) a compressed file (`tar.gz` or `zip`) with the requested **C source codes**. In the `PDF` file clearly state the name of all components of the group. Only one submission per group must be done through the Raco website.

5.1 Parallelization

1. Explain briefly the parallelization of the heat equation application that you have done for each programming model (`OpenMP`, `MPI` and `CUDA`).

5.2 Parallel execution

2. Complete a table or draw a plot in which you show the execution time and speedup (from 1, 2, 4 and 8 processors, with respect to the serial execution time) for the OpenMP and MPI parallel versions that you have developped. Indicate clearly the solver being used and the problem size.

5.3 Source codes

3. Include the source codes with the `OpenMP`, `MPI` and `CUDA` parallelizations of the heat equation application for the solvers that you have studied.