CPDS (Parallelism) laboratory assignment: Solving the heat equation

Eduard Ayguadé and Josep-Ramon Herrero

Course 2014-15, Spring Term

Index

Index **1**

1 Sequential heat diffusion program **2**

2 Analysis with Tareador **4**

3 Shared–memory parallelization with `OpenMP` (homework) **6**

4 Message–passing parallelization with `MPI` **8**

5 Message–passing parallelization with `MPI` (continuation) **10**

6 Deliverables **11**
6.1 Analysis with Tareador 11
6.2 Source codes 11
6.3 Parallel execution 11

1

Sequential heat diffusion program

In this session 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*, *Red–Black* 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 picture below shows the resulting heat distribution when a single heat source is placed in the lower right corner. 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).

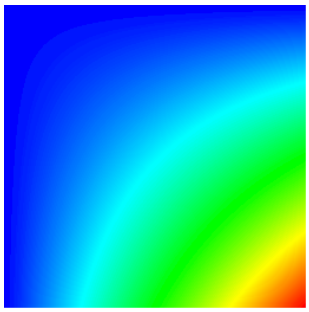

Figure 1.1: Image representing the temperature in each point of the 2D solid body

1. Login into the *Minotauro* machine at the Barcelona Supercomputing Center (BSC-CNS). To do that open a terminal window in your laptop and connect to *Minotauro* using `ssh -X cpds92xxx@mt1.bsc.es` (the professor will assign each group a number xxx in the range 001-004). The first time you log in the system will ask you to change the password.

2. Copy the tarball with all files needed to do this laboratory assignment from `/home/cpds92/cpds92000/LabCPDS.tar.gz` in Minotauro and uncompress it. Go into the `Lab` directory generated and source the `environment.bash` file to appropriately define paths and environment variables (`"source ./environment.bash"`). Don't forget to always source this environment file every time you log into your account.

3. Go into the `part1` directory. Compile the sequential version of the program using `"make heat"` and execute the binary generated (`"./heat test.dat"`). 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. `"eog heat.ppm"` or `"gimp heat.ppm"`) and save it for validation purposes with a different name, e.g. `heat-jacobi.ppm`.

4. Change the solver from *Jacobi* to *Red-Black* and to *Gauss-Seidel* 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.

2

Analysis with Tareador

Next we will use *Tareador* to analyze the potential parallelism that we can achieve for the three different solvers. We already provide you with an incomplete instrumented version for the *Jacobi*; take a look at the instrumentation performed in order to identify the parallel tasks we are proposing.

1. Compile the initially proposed task decomposition using `"make heat-tareador"`. Execute the `"run_tareador.sh"` script to run the binary generated (`"./run_tareador.sh heat-tareador testgrind.dat"`). Notice that we are using the `testgrind.dat` as the configuration file for the Tareador instrumented executions (which just performs one iteration on a very small image). The script will open a new window to display the task graph obtained from the instrumented execution.

2. Which accesses to variables are causing the serialization of all the tasks when using the *Jacobi* solver? If you were able to protect them, what would be the task graph that would be generated? Insert the calls to `tareador_disable_object` and `tareador_enable_object` in the source code and obtain the new task graph. Are you increasing the parallelism? Have you obtained the task graph you were expecting?

3. Simulate the parallel execution by executing the `run_dimemas.sh` script, in which you will have to specify the name of the instrumented binary (`heat-tareador`) and the number of processors you want to simulate, for example 1, 2, 4, 8 and 16. As an example, if we want to perform the simulation with 4 processors: `./run_dimemas.sh heat-tareador 4`

4. Each simulation generates a trace file that can be open with *Paraver*, using `wxparaver trace.prv tareador.cfg`. Change `trace.prv` with the name of the trace, as generated by the previous `run_dimemas.sh` script. The command tells *Paraver* to open the trace and use a configuration file that opens two windows: one showing a timeline with the execution of the tasks and another one with the parallelism profile. The same colors used in the graph are used now to display the temporal execution of the tasks. On the bottom-right corner you have the execution time as simulated by *Dimemas*. Draw a plot or do a table with the simulated execution times

and the speed-up achieved with respect to the execution with just one processor.

5. After analyzing the simulated executions, do you think there are other parts of the code involved in the execution of the *Jacobi* solver that can be decomposed into tasks? Instrument the code to create these new tasks and repeat the process.

6. Repeat the previous steps for the *Red–Black* solver. When using the *Red-Black* solver, which accesses to variables are causing the dependences among *Red* tasks, among *Black* tasks, and among *Red* and *Black* tasks? Complete the previous performance plots or tables with the simulated execution times and the speed-up achieved with *Red–Black*.

7. Repeat the previous steps for the *Gauss–Seidel* solver. Identify the causes for the dependences that appear when using the *Gauss-Seidel* solver. Complete the previous performance plots or tables with the simulated execution times and the speed-up achieved with *Gauss–Seidel*.

3

Shared–memory parallelization with OpenMP (homework)

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 3.1. Go into the `part2` directory.

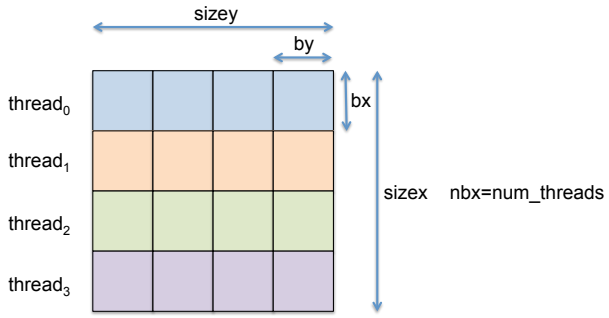

Figure 3.1: Geometric decomposition for matrix `u` (and `utmp`) in blocks

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 heatomp"` and execute with a certain number (power of 2) of OpenMP threads (e.g. `"OMP_NUM_THREADS=4 ./heatomp 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 `"mnsubmit submit-omp.sh"`) to execute the binary in the queue system (the usual procedure in the production machine). The script executes the parallel version using from 2 to 12 processors. You can check the status of your job by doing `mnq`[1]. The execution generates two files with the standard output (`heatomp.<job_id>.out`) and standard error (`heatomp.<job_id>.err`). Draw a plot or do a table with the speed-up that is achieved. Is the scalability appropriate?
- Instrument the parallel code using *Extrae* and visualize the execution with *Paraver*. To do that, just submit the `submit-extrae.sh` script, look at the trace files generated and open with `wxparaver`. The instrumentation is done with 8 threads, but you can change this number if necessary.
- Parallelize other parts of the code, or simply rewrite them in a different way, in order to improve the parallel performance. Complete the previous speed-up plot or table with the results obtained with your new version.

2. Repeat the steps in the previous bullet with the parallelization of the *Red–Black* solver and the *Gauss–Seidel* solver.

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

4

Message–passing parallelization with MPI

Next you will parallelize the heat diffusion sequential code with the *Jacobi* solver using a distributed-memory (message-passing) paradigm: `MPI`. 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 heatmpi"` and execute the binary submitting the `submit-mpi.sh` script using `"mnsubmit submit-mpi.sh"`. You can check the status of your job by doing `mnq`[1]. The execution generates two files with the standard output (`heatmpi.<job_id>.out`) and standard error (`heatmpi.<job_id>.err`).

2. Use Extrae to instrument the `MPI` parallel execution and use Paraver to visualize the parallel execution and understand where the execution time is spent (computation vs. communication). To that end, submit the `submit-extrae-mpi.sh` script using the `mnsubmit` command. We recommend that you use *Extrae* and *Paraver* whenever possible to understand how the parallelization evolves.

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

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

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.

4. 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`.

5. 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`).

6. Finally, 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.

7. With this final version, obtain the speedup when running with 1, 2, 4 and 8 processors, with respect to the sequential time (edit the `submit-mpi.sh` script and do all necessary changes).

5

Message–passing parallelization with `MPI` (continuation)

Next you will 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 that you have identified with Tareador and make sure that you guarantee them with the communication you use. Whenever possible, instrument your parallel codes using *Extrae* and visualize with *Paraver*.

6

Deliverables

Two deadlines are set to deliver the work done in this laboratory assignment:

- May 21st, 2015: before this date you should submit what is described in this page (minimum). With this deliverable the maximum mark that you can obtain for this laboratory assignment will be 7.0. Reason about all the results presented.
- June 15th, 2015: before this date you can send any additional work done on this assignment, following what is described in its documentation and not requested explicitly in this page (minimum).

Deliver a compressed tar file (`GZ` or `ZIP`) with a `PDF` that contains the answers to the questions below and the requested C source codes. In the `PDF` file clearly state the name of all components of the group. Only one file has to be submitted per group through the Raco website.

6.1 Analysis with Tareador

1. Include a view of the the task graphs obtained for the three solvers.
2. Complete a table or draw a plot in which you show the simulated execution time and speedup (from 1 to 16 processors, with respect to the execution with one processor) for the three solvers used to solve the heat equation.

6.2 Source codes

3. Include the source codes with the Tareador instrumentation and the `OpenMP` and `MPI` parallelizations of the heat equation application **only** for the *Jacobi* solver.

6.3 Parallel execution

4. Complete a table or draw a plot in which you show the execution time and speedup (from 1 to 8 processors, with respect to the serial execution time) for the OpenMP and MPI parallel versions (only for the Jacobi solver).