Task decompositions analysis for the heat equation

E. Ayguadé

October 14, 2013

Index

1

Sequential heat diffusion program

In this document we guide you through the analysis of potential task decompositions for the three solvers available in a sequential code that solvers 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, which can be visualized using `"display heat.ppm"`).

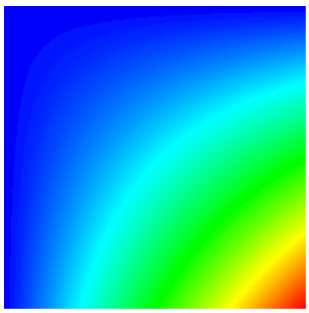

Figure 1.1: Image representing the temperature in each point of the 2D solid body

1. Login into the *Marenostrum* machine at the Barcelona Supercomputing Center (BSC-CNS). To do that open a terminal window in your laptop and connect to *Marenostrum* using the login credentials already provided

to youl. Copy the tarball with all files needed to do this laboratory session from `/home/nct00/nct00002/heatEquation.tar.gz` in Marenostrum and uncompress it. Remember to source the `environment.bash` file to appropriately define paths and environment variables (`"source ./environment.bash"`).

2. Go into the `heatEquation/tareador` 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. Save the output image file generated for validation purposes with a different name, e.g. `heat-jacobi.ppm`.

3. You can change the solver from *Jacobi* to *Red-Black* and to *Gauss-Seidel* by editing the configuration file provided (`test.dat`). The result files generated when using different solvers are slightly different; rename the `.ppm` files so that you can use them to check the correctness of the parallel versions you will program later in the course.

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* solver; take a look at the instrumentation in `heat-tareador.c` in order to identify the parallel tasks we are proposing.

1. Compile the initial task decomposition using `"make heat-tareador"`. Execute the `"run_tareador.sh"` script to run the binary generated (using `"./run_tareador heat-tareador small.dat"`). Notice that we are using the `small.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. Edit `solver-tareador.c` in order to look for potential tasks inside the *Jacobi* solver. Uncomment lines 24, 25 and 35. Save the file, compile and execute again. Finer grained tasks appear inside the initial outer task. 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. The simulation opens a couple of *Paraver* 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*.

4. 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.

5. 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?

6. Repeat the previous steps for the *Gauss–Seidel* solver. Identify the causes for the dependences that appear when using the *Gauss-Seidel* solver.