Task decompositions analysis and OpenMP parallelization for the heat equation

E. Ayguadé

November 25, 2012

Index

Index **1**

1 Sequential heat diffusion program **2**

2 Analysis with Tareador **4**

3 Shared–memory parallelization with `OpenMP` **6**

1

Sequential heat diffusion program

In this document we guide you thorough 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).

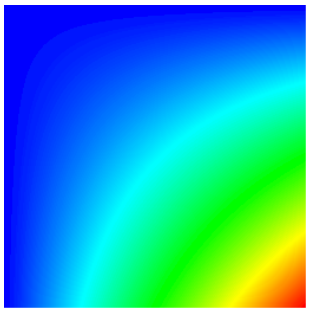

Figure 1.1: Image representing the temperature in each point of the 2D solid body

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 the login credentials already sent to you by Email. Copy the tarball with all files needed to do this laboratory session

from `/home/nct/nct00001/heat.tar.gz` in Minotauro and uncompress it. Go into the `heat` directory generated and source the `environment.bash` file to appropriately define paths and environment variables (`"source ./environment.bash"`).

2. Go into the `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. Visualize the image file generated with an image viewer (e.g. `"eog heat.ppm"`) and save it 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 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 (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. 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. Do a table 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 *Jacobi* solver.

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, adding a new column to the previous table for the new simulated results.

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 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 table with the simulated execution times and the speed-up achieved with *Gauss–Seidel*.

3

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 3.1. Go into the `openmp` directory.

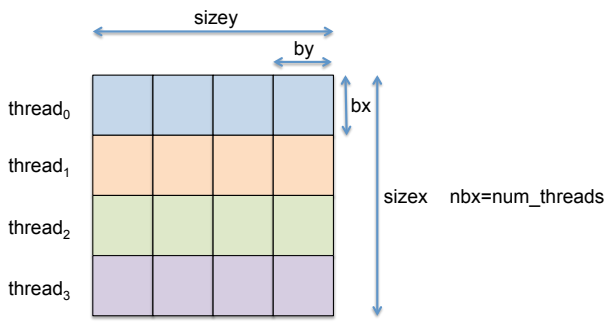

Figure 3.1: Geometric decomposition for matrix `u` (and `utmp`) in blocks

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. Do a table in which you show the parallel execution time and speedup (from 1 to 8 processors, with respect to the serial execution time) for the *Jacobi* solver.

3. Repeat the steps in the previous bullet with the parallelization of the *Red–Black* and the *Gauss–Seidel* solvers, completing the previous table with the parallel execution time and speedup for them.

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