PAR Laboratory Assignment
Lab 2: Geometric decomposition – solving the heat equation

E. Ayguadé, J. R. Herrero, D. Jiménez, N. Navarro, J. Tubella and G. Utrera

Fall 2013-14

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Index

1

Sequential heat diffusion program

In this session you will work on the parallelization of a sequential code (`heat.c`)[1] that simulates heat diffusion in a solid body using two different solvers for the heat 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 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 maximum number of simulation steps (`iterations`), the size of the body (`resolution`), the solver to be used and the heat sources (`Algorithm`). The program generates performance measurements and a file `heat.ppm` providing the solution as image (as portable pixmap file format).

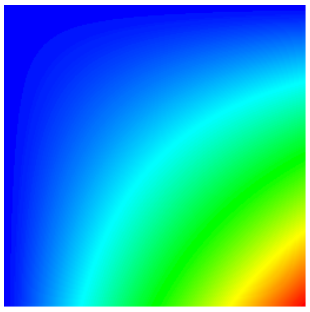

Figure 1.1: Image representing the temperature in each point of the 2D solid body

1. Compile the sequential version of the program using `"make heat"` and execute the binary generated (`"./heat test.dat"`). The execution reports the execution time (in seconds), 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 copy `heat.ppm` to a different name, e.g. `heat-jacobi.ppm` for validation purposes.

2. Change the solver from *Jacobi* 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 the two 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.

[1] Copy the file in `/scratch/boada-1/par0/sessions/lab2.tar.gz`.

2

Analysis with Tareador

1. Use *Tareador* to analyze the task graphs generated when using the two different solvers. We already provide you with the instrumented versions ready to be compiled (`"make heat-tareador"`). Take a look at the instrumentation performed in order to identify the parallel tasks we are proposing, trying to identify them with the geometric decomposition of the data in blocks that is shown in the following figure. Either submit the `submit-tareador.sh` script or directly execute the `./run-tareador.sh` script to run the binary generated, changing the configuration file to use the different solvers. Notice we are using `small.dat` as the configuration file for the Tareador instrumented executions (which just performs one iteration on a very small image).

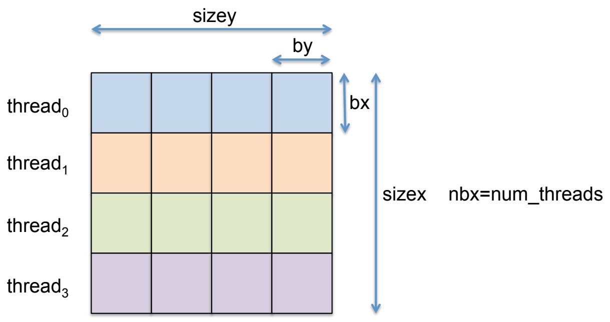

Figure 2.1: Geometric decomposition for matrix `u` (and `utmp`) in blocks

2. Reason about the following aspects **BEFORE** writing the `OpenMP` parallel version of the code.

   (a) Which accesses to variables are causing the serialization of all the tasks when using the *Jacobi* solver? You can temporarily filter the analysis for one object (variable) by using the calls to:

   ```
   tareador_disable_object(&var)
   // ... memory accesses to var
   tareador_enable_object(&var)
   ```

   Insert these calls into the source code and obtain a new task graph. Are you increasing the parallelism? How will you handle the dependences caused by the accesses to these variables in `OpenMP`?

   (b) Use *Dimemas* to simulate the potential speed–up of your proposed task decompositions (directly execute the `./run-dimemas.sh` script with the appropriate arguments). Plot the speed–up achieved when when using 2, 4, 8 and 16 processors with respect to the simulation with just 1 processor. This should give you hints about possible scalability bottlenecks in your task decomposition.

(c) Repeat the process with the *Gauss-Seidel* solver, identifying the causes for the dependences that appear between the blocks. How will you guarantee these dependences in your `OpenMP` parallelization? Complete the previous speed–up table from the simulated execution with *Dimemas* for *Gauss-Seidel.*

3

Parallelization with OpenMP

1. Parallelize the sequential code using `OpenMP`, following the geometric block decomposition suggested in Figure 2.1. We recommend that you start with the parallelization of the Jacobi solver using the `heat-omp.c` and `solver-omp.c` files, and that you follow these steps:

   - Parallelize the *Jacobi* solver, 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.
   - Use the `submit-omp.sh` script to queue the execution and the original `test.dat` configuration file to analyze the scalability of the parallelization.
   - Is the scalability appropriate? Is the number of blocks appropriate to scale to the number of cores available in the architecture? Parallelize other parts of the code, or simply rewrite them in a different way, in order to improve the parallel efficiency?

   In case you need to instrument your `OpenMP` code with *Extrae*, just type `make heat-omp-i`. The `OpenMP` original code is already prepared to use *Extrae* and generate the traces required by *Paraver*.

2. Repeat the steps in the previous bullets with the parallelization of the *Gauss-Seidel* solver.

4

Deliverable

Deliver a compressed tar file (`GZ` or `ZIP`) with the `PDF` that contains the answers to the questions below and the C source codes with the *Tareador* instrumentation and with the OpenMP parallelization. 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.

4.1 Analysis of dependences for the heat equation

Write a text, similar to the one provided below, with one or more paragraphs explaining the dependences that appear in the task graph for each one of the solvers and how do you plan to enforce them in the OpenMP parallel version. We strongly suggest that you include tables and figures to summarize and/or graphically support the information presented.

The analysis of the dependencies caused by the proposed task decomposition for the three solvers of the heat equation has been done using Tareador. Dependences appear when a task reads a memory position previously written by another task. Some dependences may impose task ordering constraints while other may impose data access constraints. For solving the heat equation three different solvers have been used, each with different dependence patterns. For the Jacobi solver, {... describe the dependences in Jacobi here...}. In the parallelization with OpenMP, we plan to guarantee these dependences { ... describe here how to enforce the dependences in Jacobi ...}. { ... Other paragraphs for Gauss-Seidel solver ...}.

The following table summarizes the predicted speed–up for the parallel execution for the two solvers, with 2, 4, 8 and 16 processors with respect to the simulation with just 1 processor. {... include table generated from the Dimemas simulations ...}.

Generic competence "Tercera llengua". As you know, this course contributes to the generic competence "Tercera llengua", in particular G3.2 "To study using resources written in English. To write a report or a technical document in English. To participate in a technical meeting in English." If you want this competence to be evaluated, write THIS section in english.

4.2 Execution analysis

1. Plot and comment the speedup achieved by the `OpenMP` parallel version, with respect to sequential execution time, for the different solvers (*Jacobi* and *Gauss-Seidel*).

2. In the parallelization of *Gauss-Seidel*, analyze the impact on the parallel execution time of the block size "`by`" (or equivalently the number of blocks "`nby`")[1]. For example try with `nby=2*nbx`, `nby=4*nbx`, ... Is there any performance impact? Is there an optimal point? Justify your answer.

[1]The resulting image should be the same.