PAR Laboratory Assignment
Lab 5: Geometric (data) decomposition: heat diffusion equation

E. Ayguadé, R. M. Badia, J. R. Herrero, J. Morillo, J. Tubella and G. Utrera

Spring 2019-20

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 two heat sources are placed in the borders of the 2D solid (one in the upper left corner and the other in the middle of the lower border). 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 (`Algorithm`) and the heat sources, their position, size and temperature. The program generates performance measurements and a file `heat.ppm` providing the solution as image (as portable pixmap file format), a gradient from red (hot) to dark blue (cold).

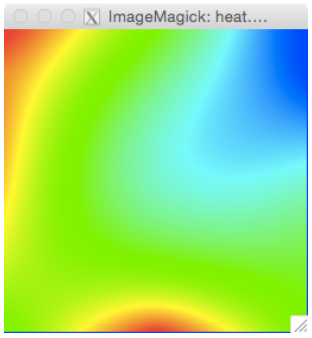

Figure 1.1: Image representing the temperature in each point of the 2D solid body

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 (you can check this by applying `diff` to the two image files generated). 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/lab5.tar.gz`.

2

Analysis with Tareador

1. In this section we will use *Tareador* to analyze the task graphs generated when using the two different solvers. We already provide you with an initial coarse-grain task definition ready to be compiled (`"make heat-tareador"`). Take a look at the instrumentation performed in order to identify the parallel tasks that are initially proposed. Compile with the appropriate `make` target and execute with `./run-tareador.sh`, 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 a couple of iterations on a very small problem).

2. Next explore the dependences that happen when a much finer-grain task decomposition is used: **one task for each iteration of the body of the innermost loop**. Change the original *Tareador* instrumentation for *Jacobi* to reflect the new proposed task granularity. Compile again, execute and analyze the task graph generated.

   (a) Which accesses to variables are causing the serialization of all the tasks? Use the *Dataview* option in *Tareador* to identify them.

   (b) Use the appropriate calls to temporarily filter the analysis for the variables you suspect are causing the serialization and obtain a new task graph. Are you increasing the parallelism? How would you guarantee these dependences in your `OpenMP` parallelization if using a parallelization strategy based on `#pragma omp for`?

3. Repeat the process with the *Gauss-Seidel* solver, identifying the causes for the dependences that appear between the tasks. How would you guarantee these dependences in your `OpenMP` parallelization if using a parallelization strategy based on `#pragma omp for`? What if based on `#pragma omp task`?

3

Parallelization of *Jacobi* with OpenMP `parallel`

In this section you will parallelize the sequential code for *Jacobi* using `#pragma omp parallel`, following a given geometric data decomposition. **Important:** you can not make use of `#pragma omp for` or combined `#pragma omp parallel for` to parallelize *Jacobi*. To start with, **make a copy** of the sequential `heat.c` and `solver.c` into `heat-omp.c` and `solver-omp.c`, respectively. Next, follow the steps enumerated below.

1. Try to understand how the C macros defined in `heat.h` are used in `solver-omp.c` to implement the geometric data decomposition used in this code.

   ```
   #define lowerb(id, p, n)  ( id * (n/p) + (id < (n%p) ? id : n%p) )
   #define numElem(id, p, n) ( (n/p) + (id < (n%p)) )
   #define upperb(id, p, n)  ( lowerb(id, p, n) + numElem(id, p, n) - 1 )
   #define min(a, b) ( (a < b) ? a : b )
   #define max(a, b) ( (a > b) ? a : b )
   ```

   Draw the geometric data decomposition that is generated when these macros are used in the *Jacobi* solver for `howmany=4`, clearly indicating the part of the matrix that corresponds to each value of `blockid`.

2. Next parallelize the code in function `relax_jacobi`, compile using `make heat-omp` and submit its execution to the queue using the `submit-omp.sh` script (using 8 threads). Validate the parallelization by visually inspecting the image generated and making a `diff` with the file generated with the original sequential version.

3. Instrument the execution of the binary with *Extrae* by submitting the `submit-omp-i.sh` script. Notice we are using `instrum.dat` as the configuration file for the *Extrae* instrumented executions (which just performs four iterations on a problem with the same resolution as the one defined in `test.dat`). Reason about the behavior observed, trying to answer the following questions: is the parallel execution appropriate for 8 threads? Why the *Jacobi* solver does not benefit from using 8 threads? Modify the source code to avoid this bottleneck and execute again without and with instrumentation. Is the parallel efficiency improving?. Is there a load balancing problem?

4. Is there any serious serialization in your parallel execution? Parallelize other parts of the code, or simply rewrite them in a different way, in order to improve even more the efficiency of your parallel code. Execute again without and with instrumentation in order to see the new behavior. Do the load balancing and/or code serialization problems persist?

5. Once you are satisfied with the parallel behaviour observed, use the `submit-strong-omp.sh` script to queue the execution of `heat-omp` and analyze the scalability of the parallelization for different number of processors (1 to 12). Reason about the scalability that is observed.

4

Parallelization of *Gauss-Seidel* with OpenMP `ordered`

Finally in this section you will parallelize the *Gauss-Seidel* solver using `#pragma omp for` and its `ordered` clause. For this solver the important part is to decide how you will synchronize the parallel execution of the rows assigned to each processor in order to guarantee the dependences that you detected with *Tareador*.

1. Parallelize the *Gauss-Seidel* solver. Compile using `make heat-omp` and submit the execution of the binary using the `submit-omp.sh` script to validate the parallelization (by visually inspecting the image generated and making a `diff` with the file generated with the original sequential version).
2. Instrument with *Extrae* by submitting the `submit-omp-i.sh` script and visualize the traces generated for the parallel execution. Does the parallel behaviour match your expectations?
3. Once the parallelization is correct, use the `submit-strong-omp.sh` script to queue the execution and analyze the scalability of the parallelization.
4. How can you control in your code the trade-off between computation and synchronization? Is there an optimum value for the ratio between computation and synchronization? For the execution with 8 threads, explore possible ratios and plot how the execution time varies.

Optional 1: Implement an alternative parallel version for *Gauss-Seidel* using `#pragma omp task` and task dependences to ensure their correct execution. Compare the performance against the `#pragma omp for` version and reason about the better or worse scalability observed.

5

Deliverable

Important:

- Deliver a document that describes the results and conclusions that you have obtained (only PDF format will be accepted). In the following subsections we highlight the main aspects/points that should be included in your document. Only PDF format will be accepted.
- The document should have an appropriate structure, including, at least, the following sections: Introduction, Parallelization strategies, Performance evaluation and Conclusions. The document should also include a front cover (assignment title, course, semester, students names, the identifier of the group, date, ...) and, If necessary, include references to other documents and/or sources of information.
- Include in the document, at the appropriate sections, relevant fragments of the C source codes that are necessary to understand the parallellization strategies and their implementation (i.e. for Tareador instrumentation and for all the OpenMP parallelization strategies).
- You also have to deliver the complete C source codes for Tareador instrumentation and all the OpenMP parallelization strategies that you have done. Include both the PDF and source codes in a single compressed tar file (`GZ` or `ZIP`). Only one file has to be submitted per group through the Raco website.

As you know, this course contributes to the **transversal competence "Tercera llengua"**. Deliver your material in English if you want this competence to be evaluated. Please refer to the "Rubrics for the third language competence evaluation" document to know the *R*ubric that will be used.

IMPORTANT: due to the COVID-19 exceptionality that caused a change in the evaluation methodology, this report will contribute to the *Continuous Assessment* mark that is used to decide if a student has to do the final exam. We encourage you to spend effort in writing this document, doing clear and complete explanations about the code changes that were introduced and the results (by means of tables, graphs, plots, ...) that are reported, as well as writing relevant concluding remarks about the outcomes of the laboratory assignment.

Analysis of task granularities and dependences

Starting from the initial coarse-grain task decomposition, explain where the calls to the *Tareador* API have been placed for the fine-grain task decomposition applied to each one of the two solvers: *Jacobi* and *Gauss-Seidel*. Explain the task graphs that you obtained and reason about the causes of the data dependences that appear and how will you protect them in your parallel `OpenMP` code.

OpenMP parallelization and execution analysis: *Jacobi*

Describe the data decomposition strategy that is applied to solve the problem, including a picture with the part of the data structure that is assigned to each processor and the relevant `OpenMP` constructs that you included in order to parallelise the execution of the heat equation using the *Jacobi* solver, commenting how did you address the serialisation and load balancing problems observed. You should include captures of *Paraver* windows to justify your explanations and the differences observed in the execution. Finally you should analyse the speed–up (strong scalability) plots that have been obtained for the different numbers of processors, reasoning about the performance that is observed.

OpenMP parallelization and execution analysis: *Gauss-Seidel*

Describe how did you implement the parallelisation strategy for the *Gauss-Seidel* solver and how did you guarantee the proper synchronization between threads. Analyse the speed–up (strong scalability) plot that has been obtained for the different numbers of processors, reasoning about the performance that is observed and including captures of Paraver windows to justify your explanations. Finally explain how did you obtain the optimum value for the ratio computation/synchronization in the parallelization of this solver for 8 threads.

Optional

If you have done the optional part in this laboratory assignment, please include and comment in your report what have you done, the relevant portions of the code, performance plots, or *Paraver* windows that have been obtained.