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

Ll. Àlvarez, E. Ayguadé, R. M. Badia, J. R. Herrero,
P. Martínez-Ferrer, J. Morillo, J. Tubella and G. Utrera

Fall 2021-22

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Index

Note:

- All files necessary to do this laboratory assignment are available in a compressed tar file available from the following location: `/scratch/nas/1/par0/sessions/lab5.tar.gz`. Copy it to your home directory in `boada.ac.upc.edu` and uncompress it with this command line: `"tar -zxvf lab5.tar.gz"`.

1

Sequential heat diffusion program and analysis with Tareador

In this laboratory assignment you will work on the parallelisation of a sequential code that simulates the diffusion of heat 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 behaviours. In any case, you should be familiar with the two solvers since we have been using them quite extensively in the course.

Take a look at the the source code of `heat.c` (where the solver is invoked) and `solver.c` (where the solvers are coded). You will soon realise that both solvers use the same function `solve`. The difference is that *Jacobi* uses a temporary matrix to store the new computed matrix (`param.uhelp`) while *Gauss-Seidel* directly updates the same matrix (`param.u`). Notice that function `solve` is iteratively invoked inside a `while` loop that iterates while two different conditions are met: 1) the maximum number of iterations `param.maxiter` is not reached; and 2) the value returned by the solver is larger than `param.residual`. And also that at each iteration of the `while` loop *Jacobi* needs to copy the newly computed matrix into the original one in order to repeat the process, by simply invoking funtion `copy_mat` also defined inside `solver.c`.

The picture in Figure 1.1 shows the resulting heat diffusion 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 number of heat sources, their position, size and temperature. The program also accepts several execution arguments: `-n`: maximum number of simulation steps or iterations; `-s`: the size of the body (resolution); `-r`: the residual value that stops the algorithm; `-a`: the solver to be used; and `-o` the output file (an image similar to the one shown in Figure 1.1 in portable pixmap file format, showing a gradient from red (hot) to dark blue (cold)). The execution of the program reports the execution time and performance measurements.

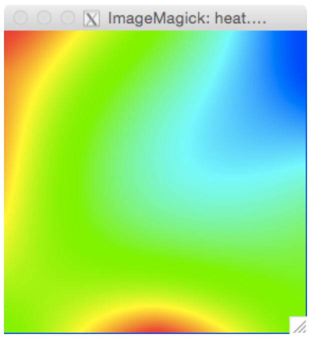

Figure 1.1: Image representing the temperature in each point of the 2D solid body

1. Compile the sequential version of the program using `"make heat"` and execute the binary generated using the *Jacobi* solver: `"./heat test.dat -a 0 -o heat-jacobi.ppm"`. 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. Visualise the image file generated with an image viewer (e.g. `"display heat-jacobi.ppm"`); keep this file in your directory to check later the correctness of the parallel versions you will program.

2. Change the solver from *Jacobi* to *Gauss-Seidel* by simply re-executing with `"./heat test.dat -a 1 -o heat-gauss.ppm"`. 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, keep the `.ppm` file generated to check later the correctness of the parallel versions you will program.

Once you understand the code, you will use *Tareador* to analyse 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"`) that you will refine later.

1. Take a look at the instrumentation performed inside `heat-tareador.c` in order to identify the parallel tasks that are initially proposed. The two solvers and an auxiliary function `copy_mat` are identified as tasks in *Tareador*. Compile with the appropriate `make` target and execute with `./run-tareador.sh`; the script has to receive the name of the executable and the solver to be used (0 for *Jacobi* or 1 for *Gauss-Seidel*). The script internally specifies a very small test case which performs a few iterations on a very small body. Is there any parallelism that can be exploited at this granularity level?

2. We assume that the answer to the previous question was not affirmative. Let's explore a finer granularity for both solvers. Open the `solver-tareador.c` file and take a closer look at the implementation of function `solve`. Notice that the function divides the computation of the 2-dimensional matrix `unew` using `u` in blocks, each block computing a subset of rows and columns. the lower and upper bounds in each dimension are computed (`i_start`, `i_end`, `j_start` and `j_end`) based on the size of the matrices that are used. **Important:** This is the granularity level we want you to explore for the tasks: **one task per block**. Make sure you understand the macros that are defined in the same file and how they are used to implement the blocking transformation.

3. Change the original *Tareador* instrumentation to reflect the new proposed task granularity. Compile again, execute and analyze the task graphs that are generated when using both *Jacobi* and *Gauss-Seidel*.

   (a) Which variable is causing the serialisation of all the tasks? Use the *Dataview* option in *Tareador* to identify it.

   (b) In order to emulate the effect of protecting the dependences caused by this variable, you can use the `tareador_disable_object` and `tareador_enable_object` calls, already introduced in the code as comments. With these calls you are telling to *Tareador* to filter the dependences caused by the variable indicated as object. Uncomment them, recompile and execute. Are you obtaining more parallelism? How will you protect the access to this variable in your OpenMP implementation? Simulate the execution when using 4 processors and extract your conclusions. Is there any other part of the code that can also be parallelised?. If so, modify again the instrumentation to parallelise it.

2

Parallelisation of the heat equation solvers

2.1 Jacobi solver

In this section you will first parallelise the sequential code for the heat equation code considering the use of the *Jacobi* solver, using the **implicit tasks** generated in `#pragma omp parallel`[1], following a *geometric block data decomposition by rows*, as shown in Figure 2.1 for 4 threads running on 4 processors.

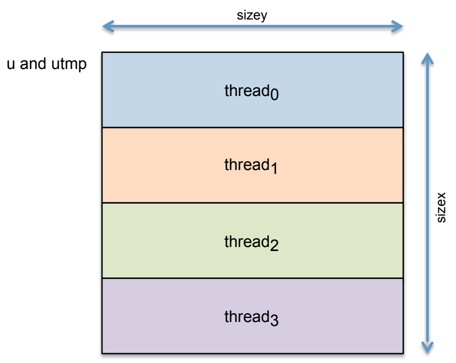

Figure 2.1: Geometric (data) decomposition for matrix `u` (and `utmp`) by rows, for 4 threads.

With this strategy the computation that the implicit task executed each thread has to perform is determined by the data it has to access, either read or write. This may have clear benefits in terms of data locality exploitation because each thread will always execute implicit tasks that access to the same data, whenever possible. How could we parallelise the sequential code for function `solve` to ensure that each processor computes its assigned elements?

1. To start with, take a look at the initial parallelisation proposed in `solver-omp.c` for function `solve`. Try to understand how the proposed parallelisation follows the data decomposition strategy mentioned above. **Complete the parallel code so that it honors the dependences that you have discovered when applying the *Jacobi* solver**. Compile using `make heat-omp` and submit its execution to the queue using the `submit-omp.sh` script, specifying the binary file, the use of the Jacobi solver and 8 threads (`sbatch submit-omp.sh heat-omp 0 8`). Validate the parallelisation by visually inspecting the image generated and making a `diff` with the file generated with the original sequential version. Is the execution time reduced? (if not, you should not continue with the next steps).

2. Once validated, submit the execution of the `submit-strong-omp.sh` script to obtain the scalability plot for different number of processors (1 to 12), specifying with an argument the solver to be used

[1]**Important:** You can not make use of other pragmas to create explicit tasks or distribute the work among the implicit tasks. You can use OpenMP intrinsic functions.

(0). In order to do a better analysis of scalability, this script is already modified to run with a larger problem size (-s 1022) but less simulation steps (-n 1000). **Is the scalability that is obtained with this initial parallelisation appropriate?**

3. In order to understand what is happening, instrument the execution of the binary with *Extrae* by submitting the `submit-extrae.sh` script with the same arguments as `submit-omp.sh`. If you look at the script you will see that the instrumented execution just performs 100 iterations on a problem with the original resolution (hot surface size) as the one used before. Reason about the behaviour of the parallel execution.

4. Is there any serious serialisation in your parallel execution? Parallelise other parts of the code in order to improve the efficiency of your parallel code. Compile the new version and submit its execution to the queue using the `submit-omp.sh` script, specifying the binary file, the use of the Jacobi solver and 8 threads. Is the execution time reduced? Make sure the new parallel version still generates correct results.

5. Once validated, instrument its parallel execution with *Extrae* in order to see the new parallel behaviour. Has the execution time for the invocations to function `solve` changed? **Why the new code that you have parallelised makes the difference in the performance results?**

6. Finally, use the `submit-strong-omp.sh` script to queue the execution of `heat-omp` and analyse the scalability of the parallelisation for different number of processors (1 to 12); do not forget to specify with an argument the solver to be used (0). Reason about the scalability that is obtained.

2.2 Gauss–Seidel solver

Once the parallelization of the solver for *Jacobi* is appropriate, you should continue the parallelization process considering the dependences that appear when the *Gauss-Seidel* is used (that you discovered using *Tareador* in the previous chapter). The parallelisation should follow the same *geometric block by rows data decomposition* that is shown in Figure 2.1, and again ONLY making use of the implicit tasks in parallel regions. **Note: before continuing, we suggest you take a look at the explanation in *Annex 1* to make sure you understand how to express ordering constraints among (implicit) tasks using shared variables and the *memory consistency* problem that may occur.**

1. Parallelise the *Gauss-Seidel* solver, introducing the necessary synchronisation among implicit tasks and data sharing constraints. In case you need to know inside function *solve* which solver has to be applied (i.e. *Jacobi* or *Gauss-Seidel*), you just need to check if `u==unew`, which will return true for *Gauss-Seidel.* The number of blocks in the `i` dimension should be determined by the geometric data decomposition (i.e. the number of threads or implicit tasks in the parallel region); for your first implementation we suggest to use the same number of blocks in the `j` dimension (i.e. `nblocksj = nblocksi`). Later you will explore different numbers of blocks in the `j` dimension.

2. Compile using `make heat-omp` and submit the execution of the binary using the `submit-omp.sh` script to validate the parallelisation (by visually inspecting the image generated and making a `diff` with the file generated with the original sequential version). Don't forget to specify the *Gauss-Seidel* solver when submitting the script (1) as well as the number of processors to use.

3. Instrument with *Extrae* by submitting the `submit-extrae.sh` script and visualize the traces generated for the parallel execution. Does the parallel behaviour match your expectations? Once the parallelisation is correct, use the `submit-strong-omp.sh` script to queue the execution and analyse the scalability of the parallelisation; do not forget to specify with an argument the solver to be used (1).

4. In order to exploit more parallelism in the execution of the solver, you should change the number of blocks in the `j` dimension. Why? Changing the number of blocks in the `j` dimension changes the ratio between computation and synchronisation, why? In order to do this change in your code, `main` is already prepared to receive an argument in the command-line execution (`-u value`) that is stored in global variable `userparam`. Modify your code accordingly to make use of this value and change

the number of blocks in the `j` dimension without recompiling the code for each new value to test. You could either use the value in `userparam` as a multiplicative factor (i.e. `nblocksj=userparam * nblocksi`) or directly as the number of blocks (i.e. `nblocksj=userparam`). Compile and execute with different values and check that it has the expected effect.

5. Use the provided `submit-userparam-omp.sh` script to explore a range of values for this argument (please set the range of values as appropriate depending on the use of `userparam`). The only argument for this script is the number of threads to be used for the exploration. For the execution with 4 and 8 threads plot how the execution time varies and explain the plot that is obtained. You can also execute with 2 and 12 threads to verify your conclusions.

2.3 Optional exercises

Several optional exercises are proposed with the objective of comparing the task and data decompositions in terms of the effects data locality has on performance.

Optional 1: Implement the *Jacobi* solver using explicit tasks and following an *iterative task decomposition*. Compare the performance results that are obtained with the ones obtained with the *data decomposition strategy*. Instrument the executions with *Extrae* and take a look at the execution time of `solve` instances.

Optional 2: Implement the *Gauss-Seidel* solver using explicit tasks and task dependences, following an *iterative task decomposition*. Compare the performance results that are obtained with the ones obtained with the *data decomposition strategy*.

3

Deliverable

Important:

- Deliver a document that describes the results and conclusions that you have obtained when doing the assignment. In the following subsections we highlight the main aspects that should be included in your document. Only PDF format for this document will be accepted.
- The document should have an appropriate structure, including, at least, the following sections: Introduction, Parallelisation 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, ...), index or table of contents, 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 parallelisation strategies and their implementation (i.e. for *Tareador* instrumentation and for all the OpenMP parallelisation strategies).
- You also have to deliver the complete C source codes for *Tareador* instrumentation and all the OpenMP parallelisation strategies that you have done. Your professor should be able to re-execute the parallel codes based on the files you deliver.

Also very important: Your professor will open the assignment in *Atenea* and set the appropriate dates for the delivery. You will have to deliver TWO files, one with the report in PDF format and one file compressed file (`tgz`, `.gz` or `.zip`) with the requested source codes.

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.

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 original task graph as well as the new graphs that you obtained, reasoning about the causes of the data dependences that appear and how will you protect them in your parallel `OpenMP` code. Include the timelines with the simulated execution for 4 processors to support your explanations.

OpenMP parallelization and execution analysis: *Jacobi*

Describe how did you implement in OpenMP the proposed data decomposition strategy for the heat equation when applying the *Jacobi* solver, commenting how did you address any detected performance bottleneck (serialisation, load balancing, ...). 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 obtained.

OpenMP parallelization and execution analysis: *Gauss-Seidel*

Describe how did you complete the previous parallelisation strategy in order to consider the dependences that appear when using the *Gauss-Seidel* solver, focussing on how did you guarantee the proper synchronization between implicit tasks. Analyse the speed–up (strong scalability) plot that has been obtained for the different numbers of processors, reasoning about the performance that is obtained and including captures of Paraver windows to justify your explanations. Finally include your conclusions about the optimum value for the ratio computation/synchronization in the parallelization of this solver for 4 and 8 threads.

Optionals

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.

4

Annex 1: Creating your own synchronisation objects

The implicit tasks used to express parallelisation strategies based on data decomposition can not synchronise themselves using task dependences (i.e. `depend` clauses can not be applied to implicit tasks). For this reason in this laboratory assignment you will also implement your own synchronisation objects to implement some sort of task ordering constraints. These synchronisation objects will be implemented using shared variables for which one has to ensure that all accesses (reads and writes) to them always access to memory. This is what is called the *memory consistency problem*: usually the compiler tries to optimise memory accesses by placing variables in registers, and then only read/write from/to memory at certain points in the parallel program, usually at synchronisation points.

For example consider the simple producer-consumer code shown in Figure 4.1, using implicit tasks. Notica that all implicit tasks can do their *computation A* part in parallel; however, the instance of the implicit task executed by `myid` cannot execute *computation B* until the previous thread has executed it (in other words, the execution of *computation B* has to be done in an ordered way). Only the instance of the implicit task executed by thread 0 can do the execution of *computation B* initially. This is controlled by vector `next` with as many positions as threads (i.e. instances of the implicit task): if position `myid` is 0, then the implicit task will wait in the `while` loop; once it is set to 1 by the implicit task `myid-1`, the implicit task will be allowed to continue.

```
int next[P];
...
next[0] = 1;
for (int i = 1; i < P; i++) next[i] = 0;
...
#pragma omp parallel num_threads(P)
{
    int myid = omp_get_thread_num();
    // computation A

    while (next[myid] == 0); // wait to advance

    // computation B

    if (myid < P-1) next[myid+1]++;
}
```

Figure 4.1: Simple producer-consumer code.

Although the code seems to be correct, it has memory consistency problems. To ensure that each task always accesses to the last value of the shared variable `next`, the programmer has to introduce some sort of data sharing/synchronisation construct. The preferred one is `atomic`, which can have three different clauses: `read`, `write` and `update`. Clauses `read` and `write` are used to express that memory accesses are always served by main memory. The resulting code is shown in Figure 4.2. Inside the `do` while construct we are ensuring that the element of vector next is read from memory by adding the atomic read; in order to ensure that the element of vector next is updated in memory we need to add atomic write.

```
int next[P];
...
next[0] = 1;
for (int i = 1; i < P; i++) next[i] = 0;
...
#pragma omp parallel num_threads(P)
{
    int myid = omp_get_thread_num();
    // computation A

    do {
        #pragma omp atomic read
        tmp = next[myid];
    } while (tmp == 0); // wait to advance

    // computation B

    if (myid < P-1) {
        #pragma omp atomic write
        next[myid+1] = 1;
    }
}
```

Figure 4.2: Simple producer-consumer code making use of atomic pragmas to guarantee memory consistency.

How would you change the code above if we introduce an iterative loop that repeats the execution of *computation A* and *computation B* several times, always ensuring the same execution order constraints? Figure 4.3 shows the code to be completed in order to ensure the appropriate execution ordering.

```
int next[P];
...
next[0] = ...;
for (int i = 1; i < P; i++) next[i] = ...;
...
#pragma omp parallel num_threads(P)
{
    int myid = omp_get_thread_num();
    for (int iters = 0; iters < num_iters; iters++) {
        // computation A

        do { ... } while ( ... ); // wait to advance

        // computation B

        if (myid < P-1) next[myid+1] = ...;
    }
}
```

Figure 4.3: Second version of simple producer-consumer code.

Question: Can the access to vector next cause false sharing? If you answered yes, how would you solve the problem?