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

Mario Acosta, Eduard Ayguadé, Rosa M. Badia (Q1),
Josep Ramon Herrero (Q1), Daniel Jiménez-González, Pedro Martínez-Ferrer, Adrian Munera,
Jordi Tubella and Gladys Utrera

Spring 2022-23

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

Before starting this laboratory assignment ...

Before going to the labroom to start this laboratory assignment, we strongly recommend that you take a look at this section and try to solve the simple questions we propose to you. This will help to better face your last programming assignment in OpenMP: data decomposition for solving the heat diffusion equation.

1.1 Some data decomposition strategies

The tasking model in OpenMP making use of explicit tasks that we have used in the two previous laboratory assignments is very versatile, allowing to express a wide range of dynamic task decompositions, both iterative (Lab 3) and recursive (Lab 4); however the programmer has no control on data locality. In this laboratory assignment you will explore the last decomposition strategy we study in this course: *data decomposition* making use of **implicit tasks**. With this strategy the computation that each implicit task 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. *Data decompositions* can be *Geometric* or *Recursive*. In this lab we will focus our attention on the Geometric ones that are applied to n-dimensional matrices (including vectors).

Assume the code shown in Figure 1.1 for which we want to write a parallel code ensuring that each thread will compute the elements of matrix `C` that are stored in its own memory.

```
void vectoradd(int *A, int *B, int *C, int N) {
   int i, j;

   for (i=0; i< N; i++)
       for (j=0; j< N; j++)
           C[i*N+j] = A[i*N+j] + B[i*N+j];

}
```

Figure 1.1: Simple example performing matrix sum.

Assuming that P is the number of processors and that $thread_k$ is executed on processor P_k, the left part of Figure 1.2 shows a possible distribution for the matrices by rows, so that the memory associated to each processor has a block of consecutive $N \div P$ rows of matrices `A`, `B` and `C`. On the right we show a parallel implementation using implicit tasks for the loop that follows the so called *Owner-computes Rule*: each processor is responsible for the computations on the elements that are allocated in its main memory. Observe that, based on the value returned by intrinsic `omp_get_thread_num()`, each thread can determine the subset of iterations of the `i` loop to execute (range between `i_start` and `i_end`); since

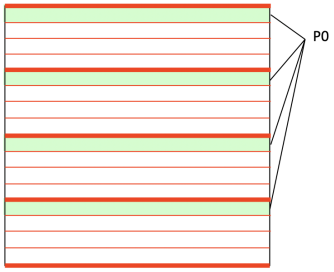
columns are not distributed, each thread executes all iterations of the `j` loop. For simplicity, the code in Figure 1.2 assumes that the number of elements N is a multiple of the number of threads executing the parallel region; think how it would change if this condition does not hold (think about it, trying to maximise load balancing).


```
#pragma omp parallel private(i, j)
{
    int my_id = omp_get_thread_num();
    int BS = N/omp_get_num_threads();
    int i_start = my_id * BS;
    int i_end = i_start + BS;

    for (i=i_start; i<i_end; i++)
        for (j=0; j<N; j++)
            C[i*N+j] = A[i*N+j] + B[i*N+j];
}
```

Figure 1.2: Left: geometric block data decomposition by rows. Right: parallelisation using implicit tasks.

Similarly, Figures 1.3 and 1.4 show two different data decomposition strategies and the code associated, always using implicit tasks, that follows the owner computes rule. In Figure 1.3 each processor also has $N \div P$ rows but now distributed in a cyclic way, starting with row 0 assigned to thread 0. Observe that the loop `i` now traverses the iteration space starting from the identifier of the thread executing the implicit task, jumping as many iterations as threads in the parallel region until reaching N.

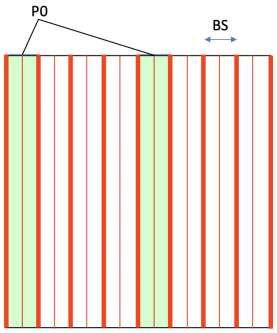



```
#pragma omp parallel private(i, j)
{
    int my_id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for(i=my_id; i<N; i+=howmany)
        for (j=0; j<N; j++)
            C[i*N+j] = A[i*N+j] + B[i*N+j];
}
```

Figure 1.3: Left: geometric cyclic data decomposition by rows. Right: parallelisation using implicit tasks.

In Figure 1.4 each processor has blocks of $BS = 2$ consecutive columns assigned in a cyclic way, starting with the first block assigned to thread 0. Observe that now the loop that is decomposed is the `j` loop since it is the one used to access the columns. Do you understand how the loop is transformed in order to follow the owner-computes rule?




```
#pragma omp parallel private(i, j)
{
    int my_id = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    for(i=0; i<N; i++)
        for (jj=my_id*BS; jj<N; jj+=howmany*BS)
            for (j=jj; j<jj+BS; j++)
                C[i*N+j] = A[i*N+j] + B[i*N+j];
}
```

Figure 1.4: Left: geometric block-cyclic data decomposition by columns. Right: parallelisation using implicit tasks, assuming $N\%(BS \times howmany) == 0$.

Considering that two–dimensional matrices are stored in memory by rows, which should be the value (or values) for BS to avoid false sharing when writing elements `C[i*N+j]`?

2

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 2.1 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 2.1 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.

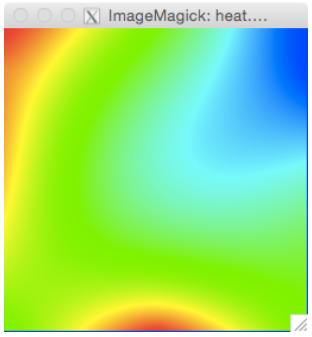


Figure 2.1: Image representing the temperature in each point of the 2D solid body

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.

   For the deliverable: Include the task dependency graph shown by *Tareador*. 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*.

   For the deliverable: Include the excerpt of the code that you have modified in order to specify **one task per block**.

   (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.

   (c) Simulate the execution of both solvers when using 4 threads. Is there any other part of the code that can also be parallelised (take into account this question for the parallel version!)?. If so, modify again the instrumentation to parallelise it.

   For the deliverable: Include the task dependency graph shown by *Tareador* after adding the dependences filter and the new tasks per block in other parts of the code to increase parallelism. Which variable was causing the serialisation of all the tasks? How will you protect the access to this variable in your OpenMP implementation? Are you obtaining more parallelism than in the previous version? Is the parallelism achieved the same for *Jacobi* and *Gauss-Seidel*?

3

Parallelisation of the heat equation solvers

3.1 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 3.1 for 4 threads running on 4 processors.

3.1.1 First Implementation

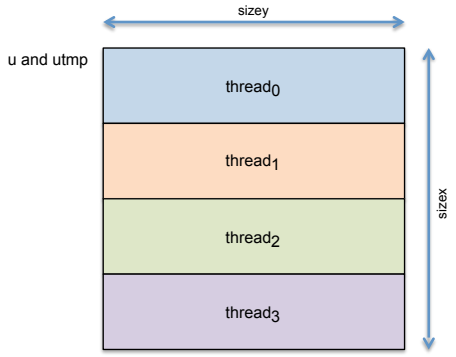


Figure 3.1: Geometric (data) decomposition for matrix `u` (and `utmp`) by rows, for 4 threads.

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?

To start with, analyse 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 1 (`sbatch submit-omp.sh heat-omp 0 1`) 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.

[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.

Is the execution time of the `OpenMP` version of heat using 8 threads reduced compare to the sequential execution or execution using 1 thread? if not, you should reconsider your implementaton. For instance, what kind of synchronization are you using? Review different strategies to avoid and/or reduce the amount of sychronization overheads per interation.

3.1.2 Overall Analysis

Once the parallelisation is validated with some speedup when using more than 1 thread, submit the execution of the `submit-strong-extrae.sh` script to trace the execution for different number of threads (1, 4, 8 and 16). This script will also execute `modelfactors.py` that will help you to perform the overall analysis. When submitting the script you should specify the solver to be used (argument with value 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).

For the deliverable: Include the *Modelfactor* tables. Is the scalability that is obtained with this initial parallelisation appropriate? Which is the metric reported by `modelfactors.py` that you should address first?

3.1.3 Detailed Analysis

We recommend you to open with `paraver` the trace that has been generated for 16 threads and observe what is causing the low value for that metric. Use `paraver` Hints Instantaneous Parallelism and Implicit Tasks in Parallel Constructions, synchronize them and zoom in one of the trace to see the detail of the execution.

For the deliverable: Include some captures of the window timelines to show the problem. What is the region of the code that is provoking the low value for the *parallel fraction* in your parallelisation? Hint: Remember the *Tareador* analysis you did.

3.1.4 Optimization

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 16 threads. Make sure the new parallel version still generates correct results.

For the deliverable: Include an excerpt of the code to show the OpenMP annotations you have added to the code.

Overall Analysis of the Optimized Code

Once validated, submit again the `submit-strong-extrae.sh` script and analyse the scalability of the new parallelisation for different number of threads (1, 4, 8 and 16); do not forget to specify with an argument the solver to be used (0). For completeness, use the `submit-strong-omp.sh` script to queue the execution of `heat-omp` and obtain the scalability plot to be included in the deliverable; do not forget to specify with an argument the solver to be used (0).

For the deliverable: Include the *Modelfactor* tables and the plot of scalability. Is the execution time reduced?. Have you increased the scalability? Is the *parallel fraction* larger than before? What is the speedup that you have achieved compared to your first implementation for 16 threads?

Detailed Analysis of the Optimized Code

We recommend you to open with `paraver` the trace that has been generated for 16 threads and observe the behaviour for your parallel code. Use `paraver` Hints Instantaneous Parallelism and Implicit Tasks in Parallel Constructions, synchronize them and zoom in one of the trace to see the detail of the execution.

For the deliverable: Include some captures of the window timelines.

3.2 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).

3.2.1 First Implementation

The parallelisation should follow the same *geometric block by rows data decomposition* that is shown in Figure 3.1, 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 4 blocks in the `j` dimension. 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 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 threads to use. **For the deliverable:** Include an excerpt of the code to show the modifications done.

3.2.2 Overall Analysis

Once validated, submit the execution of the `submit-strong-omp.sh` script to queue the execution of `heat-omp` and obtain the scalability plot; do not forget to specify with an argument the solver to be used (1).

For the deliverable: Include the plots obtained with `submit-strong-omp.sh` and explain the plots that are obtained. Do you observe a linear speedup? Note: Remember the analysis done with *tareador* with a simulation with 4 threads.

3.2.3 Detailed Analysis and Optimization

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, and may increase the parallelism, 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. Use the value in `userparam` directly as the number of blocks (i.e. `nblocksj=userparam`). Compile and execute with different values and check that it has the expected results and effect.

For the deliverable: Include an excerpt of the code to show the modifications done.

Number of blocks tune

Use the provided `submit-userparam-omp.sh` script to explore a range of values of the number of blocks. The only argument for this script is the number of threads to be used for the exploration. Submit this script for 16 threads. Select a number of blocks and submit `submit-strong-omp.sh` script to queue the execution of `heat-omp` and obtain the scalability plot; do not forget to specify with an argument the solver to be used (1).

For the deliverable: Include the plots obtained with `submit-userparam-omp.sh` and explain the plot that is obtained. Also, include the scalability plot and explain the performance obtained. Compare it with the version with `nblocksj=4`. Reason *why* changing the number of blocks in the `j` dimension changes the ratio between computation and synchronisation.

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. Notice 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?