SC12 HPC Educators session:
Unveiling parallelization strategies at undergraduate level

E. Ayguadé, R. M. Badia, D. Jiménez, J. Labarta and V. Subotic

August 31, 2012
Index

Index

1 The infrastructure: Tareador 2
  1.1 Inputs to Tareador ................................................. 2
  1.2 Implementation of Tareador ........................................... 2
    1.2.1 Tracer .......................................................... 3
    1.2.2 Replay simulator ................................................. 4
  1.3 Outputs of Tareador .................................................. 4

2 A complete example: Cholesky 6
  2.1 Decomposition D1 .................................................... 6
  2.2 Decomposition D2 .................................................... 8
  2.3 Decomposition D3 .................................................... 9
  2.4 Decomposition D4 ................................................... 11
  2.5 Decomposition D5 ................................................... 13
  2.6 Decomposition D6 ................................................... 15
  2.7 Decomposition D7 ................................................... 17

3 Hands-on: analyzing the potential parallelism in different solvers for the heat diffusion kernel 19

4 Additional material 21
  4.1 Embarrassingly parallel code: Mandelbrot set ........................ 21
  4.2 Divide and conquer: Multisort ...................................... 22
The infrastructure: Tareador

In this HPC Educators session we present the environment, named Tareador, we have designed to assist the programmer in the process of finding appropriate task decomposition strategies for their sequential applications. With Tareador the programmer proposes how he intends to break the sequential code into tasks and the environment automatically estimates how much parallelism can be achieved, in an ideal parallel architecture, with the selected task decomposition. The environment has been successfully used to support an undergraduate course on Parallelism at the Computer Science School (Technical University of Catalunya, Barcelona–Spain). The environment allows students to estimate the potential parallelism that could be obtained with a given task decomposition and the variables that cause task interactions, prior to implementing that task decomposition using a parallel programming model (e.g. OpenMP or MPI).

Next we further explain how the proposed environment works, the information that is provided by the programmer and the information reported to the programmer.

1.1 Inputs to Tareador

The input to Tareador is a sequential application with simple annotations that specify a possible task decomposition for the sequential code. To do that, the programmer manually inserts two function calls:

```c
tareador_start_task("name of task");
/* Code region / task */
tareador_end_task();
```

A simple example is shown in Figure 1.1. These functions allow the specification of any arbitrary task decomposition (nesting of tasks is supported), without any refactoring of the sequential code. In addition, the programmer needs to insert two additional function calls to initialize and finalize Tareador (start_tareador() and end_tareador(), respectively).

1.2 Implementation of Tareador

The proposed framework does the following steps:

1. it runs the annotated sequential code annotated with a possible task decomposition;
2. it dynamically analyzes the memory usage of each annotated task;
3. it identifies possible data dependencies among all task instances; and
4. it simulates how those tasks would ideally execute in parallel with a specific number of processors.

The framework (shown in Figure 1.2) takes the input code and passes it through the tool chain that consists of a Valgrind-based tracer, the Dimemas replay simulator and the Paraver visualization tool.
```c
void tareador_start_task(char* task);
for (i=0; i<N; i++)
    A[i] = randomize (i);
reset_buf (B);
tareador_end_task();

tareador_start_task("compute_A");
compute(A);
tareador_end_task();

tareador_start_task("compute_B");
compute(B);
tareador_end_task();

tareador_start_task("sum_both");
res = 0
res += sum (A);
res += sum (B);
tareador_end_task();
```

Figure 1.1: Simple example with annotations to specify a possible task decomposition

The Valgrind-based tracer instruments the sequential execution of the annotated input code. The tracer dynamically tracks task instances and identifies their memory usage. With this information the tracer derives all inter–task dependences and generates a trace of the potential parallel execution of these tasks. From the generated trace, Dimemas reconstructs the potential temporal behavior on an ideal parallel architecture. Paraver visualizes the simulated temporal behavior and allows a deep study of the potential parallel execution. Moreover, the trace generated can be further processed in order to derive the task dependency graph, which can be visualized within the Graphviz environment.

1.2.1 Tracer

Leveraging Valgrind functionalities, the tracer instruments accesses to all memory objects and derives data dependencies among tasks. By intercepting all dynamic allocations and releases of the memory (allocs and frees), the tool maintains the pool of all dynamic memory objects. Similarly, by intercepting all static allocations and releases of the memory (mmaps and munmaps), and reading the debugging information of the executable, the tool maintains the pool of all the static memory objects. The tracer tracks all memory objects, intercepting and recording accesses to them at the granularity of one byte. Based on these records, and knowing in which task the execution is at every moment, the tracer detects all read-after-write dependencies and interpret them as dependences among tasks.

Given the result of the instrumentation, the tracer generates the trace of the potential parallel execution of the task decomposition for the sequential code. The trace of the original (actually executed)
sequential code consists of only one computation burst. On the other hand, while generating the trace of the potential parallel execution, the tracer breaks the original computation burst into smaller bursts that correspond to instances of the programmer-annotated tasks. Furthermore, based on the identified inter-task dependences, the tracer inserts in the trace inter-task synchronization, which is then used by Dimemas to simulate the parallel temporal behavior on a configurable ideal target architecture.

1.2.2 Replay simulator

Dimemas is a trace-driven simulator that simulates MPI processes and message-passing among them. Dimemas traces consist of two types of events: computation burst that are replayed during the simulation; and communication requests that are reevaluated during the simulation. Reevaluating communication requests means computing communication durations for the configured target machine. Paraver can then visualize temporal behavior simulated by Dimemas, allowing to visualize all communications among different MPI processes.

In Tareador we have extended Dimemas to support synchronization of tasks in a way that allows Paraver to visualize data dependencies between tasks. We defined semantics that allow that each MPI process consists of multiple shared memory tasks. Also, we implemented a task synchronization as a lightweight intra-node MPI transfer. This way, Paraver can visualize the simulated temporal behavior showing also data dependencies among tasks. Using this feature, the programmer can visually detect each execution bottleneck and further inspect its causes. New scheduling policies have been included in Dimemas to optimize task-based dataflow parallelism.

1.3 Outputs of Tareador

Figure 1.3(a) shows the dependency graph generated by Tareador for the code presented in Figure 1.1. Tareador generates graphs in Graphviz format, allowing xdot to visualize graphs in vector form with basic zooming functionality. Each node in the graph represents one task instance, while each vertex represents a data dependence. A node in the graph also includes additional information about the corresponding task instance (name of the task type, unique id of the instance, number of instructions inside the instance, nesting level of task instance, etc.). Also, the size of a node is proportional to the number of instructions in the execution of the task instance.

The second result of Tareador is the simulated parallel execution. Figure 1.3(b) illustrates how code in Figure 1.1 could execute in parallel on a machine with 2 cores. In both execution timelines, the x-axis represents the time through the complete execution of the program. Different Paraver configurations
allow different views of the interesting execution properties. In this figure, the lower Paraver view shows the number of cores that are active during the execution. From this plot, the programmer can see that most of the time in the parallel execution, only one core is active. In the same figure, the upper Paraver window shows, for each core in the system, which task is being executed in every moment. A white burst represents that the core is idle while a colored burst represents that a specific task is executing. Task colors in this Paraver view match the colors in the dependency graph. For example Figure 1.3(b) shows that tasks init_buffers and sum both never run in parallel with any other task. Finally, comparing these two views on the same time scale, the programmer can see that the region in which there are two cores active in the machine (lower figure) corresponds to the region in which compute_B and compute_A run in parallel (upper figure).

The Tareador outputs can be very useful in identifying parallelization bottlenecks and deciding how to refine a task decomposition in order to achieve higher parallelism. Therefore, Tareador can be used to explore possible task decompositions of a sequential application. In the following Chapter, we illustrate an iterative approach that uses Tareador to find the optimal task decomposition for a Cholesky sequential kernel.
2

A complete example: Cholesky

In this section we will demonstrate the use of the infrastructure described in the previous section to analyze possible parallelization strategies, and their potential parallelism, for the well-known Cholesky factorization kernel. The initial sequential code is the following:

```c
void compute(struct timeval *start, struct timeval *stop, long NB, long DIM,
float *A[DIM][DIM])
{
for (long j = 0; j < DIM; j++)
{
for (long k = 0; k < j; k++)
for (long i = j+1; i < DIM; i++)
sgemm_tile( &A[i][k][0], &A[j][k][0], &A[i][j][0], NB);

for (long i = 0; i < j; i++)
ssyrk_tile( A[j][i], A[j][j], NB);

spotrf_tile( A[j][j], NB);

for (long i = j+1; i < DIM; i++)
strsm_tile( A[j][j], A[i][j], NB);
}
}
```

2.1 Decomposition D1

In our top-to-bottom approach we start from the most coarse-grain task-decomposition, in which we put the whole execution into one task, as shown in the following code.
void compute(struct timeval *start, struct timeval *stop, long NB, long DIM, float *A[DIM][DIM])
{
    tareador_start_task("whole_compute"); // blue task
    for (long j = 0; j < DIM; j++)
    {
        for (long k = 0; k < j; k++)
            for (long i = j+1; i < DIM; i++)
                sgemm_tile( &A[i][k][0], &A[j][k][0], &A[i][j][0], NB);
        for (long i = 0; i < j; i++)
            ssyrk_tile( A[j][i], A[j][j], NB);
        spotrf_tile( A[j][j], NB);
        for (long i = j+1; i < DIM; i++)
            strsm_tile( A[j][j], A[i][j], NB);
    }
    tareador_end_task();
}

For this initial task decomposition, a single node is visualized in the task graph, as shown in Fig. 2.1.
The simulated execution timeline in Fig. 2.2 shows a single task that takes the whole execution time.

![Figure 2.1: Dependency graph of decomposition D1](image1)

![Figure 2.2: Simulation of the potential parallel execution of decomposition D1](image2)
2.2 Decomposition D2

In the next proposed decomposition, we refine the current decomposition by breaking the main loop \( j \) into its iterations.

```c
void compute(struct timeval *start, struct timeval *stop, long NB, long DIM,
             float *A[DIM][DIM])
{
    for (long j = 0; j < DIM; j++)
    {
        tareador_start_task("main_loop_one_iter"); // red task
        for (long k = 0; k < j; k++)
            for (long i = j+1; i < DIM; i++)
                sgemm_tile( &A[i][k][0], &A[j][k][0], &A[i][j][0], NB);
        for (long i = 0; i < j; i++)
            ssyrk_tile( A[j][i], A[j][j], NB);
        spotrf_tile( A[j][j], NB);
        for (long i = j+1; i < DIM; i++)
            strsm_tile( A[j][j], A[i][j], NB);
        tareador_end_task();
    }
}
```

Figure 2.3 shows the resulting task graph, in which all tasks (one per iteration of the \( j \) loop) are serialized. The simulated execution timeline in Fig. 2.4 reports the same execution time that we obtained for decomposition D1.

![Dependency graph of decomposition D2](image-url)
2.3 Decomposition D3

In the next decomposition, we break task main_loop_one_iter into its logical parts.

```c
void compute(struct timeval *start, struct timeval *stop, long NB, long DIM,
             float *A[DIM][DIM])
{
    for (long j = 0; j < DIM; j++)
    {
        tareador_start_task("first_loop_complete"); // light green task
        for (long k = 0; k < j; k++)
            for (long i = j+1; i < DIM; i++)
                sgemm_tile( &A[i][k][0], &A[j][k][0], &A[i][j][0], NB);
        tareador_end_task();

        tareador_start_task("second_loop_complete"); // yellow task
        for (long i = 0; i < j; i++)
            ssyrk_tile( A[j][i], A[j][j], NB);
        tareador_end_task();

        tareador_start_task("spotrf_tile"); // red task
        spotrf_tile( A[j][j], NB);
        tareador_end_task();

        tareador_start_task("third_loop_complete"); // dark green task
        for (long i = j+1; i < DIM; i++)
            strsm_tile( A[j][j], A[i][j], NB);
        tareador_end_task();
    }
}
```

Now the task graph in Fig. 2.5 shows the potential parallelism that exists between first_loop_complete and the sequence second_loop_complete-spotrf_tile. This parallelism results in a reduction of the execution time, as shown in the simulated execution timeline in Fig. 2.6.
Figure 2.5: Dependency graph of decomposition D3

Figure 2.6: Simulation of the potential parallel execution of decomposition D3
2.4 Decomposition D4

Figure 2.6 shows that the light green task (first_loop_complete) is the parallelization bottleneck. That task is by far the longest one, so it is a good candidate to be broken into smaller ones, as shown below:

```c
void compute(struct timeval *start, struct timeval *stop, long NB, long DIM,
               float *A[DIM][DIM])
{
    for (long j = 0; j < DIM; j++)
    {
        for (long k = 0; k < j; k++)
        {
            tareador_start_task("first_loop_iter_1"); // blue violet task
            for (long i = j+1; i < DIM; i++)
                sgemm_tile( &A[i][k][0], &A[j][k][0], &A[i][j][0], NB);
            tareador_end_task();
        }
        tareador_start_task("second_loop_complete");  // yellow task
        for (long i = 0; i < j; i++)
            ssyrk_tile( A[j][i], A[j][j], NB);
        tareador_end_task();
        tareador_start_task("spotrf_tile");          // red task
        spotrf_tile( A[j][j], NB);
        tareador_end_task();
        tareador_start_task("third_loop_complete");  // dark green task
        for (long i = j+1; i < DIM; i++)
            strsm_tile( A[j][i], A[i][j], NB);
        tareador_end_task();
    }
}
```

Now the task graph in Fig. 2.7 shows the parallelism that exists among all instances of these new tasks coming from the same initial task. The simulated execution timeline in Fig. 2.8 reports the noticeable reduction in the execution time that is achieved. This timeline shows that a parallelism of 4 is achieved during some fragments of the execution.
Figure 2.7: Dependency graph of decomposition D4

Figure 2.8: Simulation of the potential parallel execution of decomposition D4
2.5 Decomposition D5

Figure 2.8 shows that the dark-green task (third_loop_complete) is the parallelization bottleneck. The instances of first_loop_iter_1 (blue-purple task) parallelize well, but cannot start before third_loop_complete finishes. Therefore, we will break third_loop_complete, hoping that will allow instances of first_loop_iter_1 to start earlier.

```c
void compute(struct timeval *start, struct timeval *stop, long NB, long DIM,
float *A[DIM][DIM])
{
    for (long j = 0; j < DIM; j++)
    {
        for (long k = 0; k < j; k++)
        {
            tareador_start_task("first_loop_iter_1"); // blue violet task
            for (long i = j+1; i < DIM; i++)
                sgemm_tile( &A[i][k][0], &A[j][k][0], &A[i][j][0], NB);
            tareador_end_task();
        }
        tareador_start_task("second_loop_complete"); // yellow task
        for (long i = 0; i < j; i++)
            ssyrk_tile( A[j][i], A[j][j], NB);
        tareador_end_task();
        tareador_start_task("spotrf_tile"); // red task
        spotrf_tile( A[j][j], NB);
        tareador_end_task();
        for (long i = j+1; i < DIM; i++)
        {
            tareador_start_task("strsm_tile"); // magenta task
            strsm_tile( A[j][j], A[i][j], NB);
            tareador_end_task();
        }
    }
}
```

The task graph and simulated execution timeline are shown in Fig. 2.9 and 2.10, respectively. As expected, this decomposition leads to an increase in the potential parallelism and reduction of the total execution time.
Figure 2.9: Dependency graph of decomposition D5

Figure 2.10: Simulation of the potential parallel execution of decomposition D5
2.6 Decomposition D6

Now the parallelization bottleneck is yellow task (second_loop_complete) – the long task appearing close to the end of the execution. We need to break this task into smaller ones and try to execute this computation concurrently with blue-purple tasks (because they carry most of the total computation time).

```c
void compute(struct timeval *start, struct timeval *stop, long NB, long DIM,
float *A[DIM][DIM])
{
    for (long j = 0; j < DIM; j++)
    {
        for (long k = 0; k < j; k++)
        {
            tareador_start_task("first_loop_iter_1"); // blue violet task
            for (long i = j+1; i < DIM; i++)
                sgemm_tile( &A[i][k][0], &A[j][k][0], &A[i][j][0], NB);
            tareador_end_task();
        }
        for (long i = 0; i < j; i++)
        {
            tareador_start_task("ssyrk_tile"); // brown orange task
            ssyrk_tile( A[j][i], A[j][j], NB);
            tareador_end_task();
        }
        tareador_start_task("spotrf_tile"); // red task
        spotrf_tile( A[j][j], NB);
        tareador_end_task();
        for (long i = j+1; i < DIM; i++)
        {
            tareador_start_task("strsm_tile"); // magenta task
            strsm_tile( A[j][j], A[i][j], NB);
            tareador_end_task();
        }
    }
}
```
Figure 2.11: Dependency graph of decomposition D6

Figure 2.12: Simulation of the potential parallel execution of decomposition D6
2.7 Decomposition D7

Now the parallelization bottleneck is blue-purple task (first_loop_iter_1). Figure 2.12 shows that task first_loop_iter_1 is late giving dependencies, thus most of the stalls appear after some instance of first_loop_iter_1. So finally, we will break first_loop_iter_1.

```c
void compute(struct timeval *start, struct timeval *stop, long NB, long DIM,
float *A[DIM][DIM])
{
    for (long j = 0; j < DIM; j++)
    {
        for (long k = 0; k < j; k++)
        {
            for (long i = j+1; i < DIM; i++)
            {
                tareador_start_task("sgemm_tile"); // olive green task
                sgemm_tile( &A[i][k][0], &A[j][k][0], &A[i][j][0], NB);
                tareador_end_task();
            }
        }
    }

    for (long i = 0; i < j; i++)
    {
        tareador_start_task("ssyrk_tile"); // brown orange task
        ssyrk_tile( A[j][i], A[j][j], NB);
        tareador_end_task();
    }

    tareador_start_task("spotrf_tile"); // red task
    spotrf_tile( A[j][j], NB);
    tareador_end_task();

    for (long i = j+1; i < DIM; i++)
    {
        tareador_start_task("strsm_tile"); // magenta task
        strsm_tile( A[j][j], A[i][j], NB);
        tareador_end_task();
    }
}
```

Observe the task graph obtained and the parallelism that is achieved in the simulated execution timeline. An speed-up close to 3.5 is obtained with 4 processors.
Figure 2.13: Dependency graph of decomposition D7

Figure 2.14: Simulation of the potential parallel execution of decomposition D7
Hands-on: analyzing the potential parallelism in different solvers for the heat diffusion kernel

Next you will work on analyzing the potential parallelism available in the sequential (heat.c) code. This code simulates heat diffusion in a solid body, using several solvers for the heat equation (Jacobi, Red-Black and Gauss-Seidel). Each solver has different numerical properties which are not relevant for the purposes of this hands-on session; we use them because they show different parallel behaviors. Once compiled, 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). The picture below shows the resulting heat distribution when a single heat source is placed in the lower right corner.

Figure 3.1: 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, 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"). Change the solver from Jacobi to Red-Black and 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 different solvers are slightly different.

2. Take a look at the instrumented version of the original sequential code (solver-tareador.c) that we provide you. In this initial version, the Jacobi solver is instrumented so that one task corresponds with the execution of one block of the matrix that represents the 2D solid body.
This version is ready to be compiled ("make heat-tareador"). Use the run_tareador.sh script to run the binary generated, changing the configuration file to use the Jacobi solver. For this purpose, we recommend that you use the testgrind.dat configuration file (which just performs one iteration on a very small image). From the task graph generated, try to guess the accesses to variables that are causing the serialization of all the tasks? How would you handle these dependences in the most appropriate way, for example in OpenMP? Is it possible to identify other tasks in the main time step loop?

3. Simulate the execution with Dimemas of the previous task decomposition with the Jacobi solver. To do that execute "./run_dimemas heat-tareador n" being n the number of processors to simulate. Compute the speed-up when when using 2, 4, 8 and 16 processors with respect to the execution with 1 processor.

4. Now it is your turn to instrument the Red-Black and the Gauss-Seidel solvers. Repeat steps 2 and 3 in order to visualize the task graphs generated and simulate the parallel execution. For the Red-Black solver, which accesses to variables are causing the dependences among Red tasks, among Black tasks, and among Red and Black tasks? How would you handle these dependences in the most appropriate way, for example in OpenMP? Identify the causes for the dependences that appear when using the Gauss-Seidel solver. How would you guarantee them, for example in OpenMP?
Additional material

In this part of the document we provide you with other examples that you will be able to use to analyze the potential of the environment we have been describing and using in this training session. Source code for the instrumented versions, as well as all makefile and scripts, are provided in the appropriate directory of the software distribution.

4.1 Embarrassingly parallel code: Mandelbrot set

The Mandelbrot set is a particular set of points, in the complex domain, whose boundary generates a distinctive and easily recognizable two-dimensional fractal shape (Figure 4.1). To each point \( c \) the complex quadratic polynomial recurrence \( z_{n+1} = z_n^2 + c \) is iteratively applied. That is, a complex number \( c \) is part of the Mandelbrot set if, when starting with \( z_0 = 0 \) and applying the iteration repeatedly, the absolute value of \( z_n \) never exceeds a certain number however large \( n \) gets. A plot of the Mandelbrot set is created by coloring each point \( c \) in the complex plane with the number of steps \( \text{max} \) for which \( |z_{\text{max}}| \geq 2 \) (or simply \( |z_{\text{max}}|^2 \geq 2 \times 2 \) to avoid the computation of the square root in the modulus of a complex number). In order to make the problem doable, the maximum number of steps is also limited.

![Figure 4.1: Fractal Shape](image)

In this first part you should understand how the code `mandel-serial.c` implements the above mentioned mathematical formulation in the `mandelbrot` function and analyze, using the `Tareador` tool, the potential parallelism that is available and its main characteristics.

1. Compile the sequential version of the program using `make mandel` and `make mandeld`. The first one generates a binary that will be used for timing purposes and to check for the numerical validity of the output (`-o` option). The second one generates a binary that visualizes the Mandelbrot
set. Execute both binaries with no additional parameters and compare the execution time reported. When executing `mandeld` the program will wait for a key to be pressed inside the created X window; once the program finishes drawing the Mandelbrot set, waits again for a key to be pressed; in the meanwhile, you can find new coordinates in the complex space by just clicking with the mouse (these coordinates could be used to zoom the exploration of the Mandelbrot set). Execute `./mandel -h` and `mandeld -h` to figure out the options that are available to run the program. For example `./mandeld -c -0.737 0.207 -s 0.01 -i 100000`.

2. Think about two possible task granularities that you can exploit in this program. Complete the sequential `mandel-tareador.c` code partially instrumented to reflect the two task granularities that you decided. Once instrumented, compile the instrumented code using "make mandel-tareador" and "make mandeld-tareador" to generate the two executables to be run with `run_tareador.sh`. This script accepts two arguments: the name of the binary and the size of the display used to visualize the Mandelbrot set (option `-w`). Use a small size in order to generate a small task graph, for example `./run_tareador.sh mandel-tareador 16`. Which are the two most important characteristics of the task graph generated for `mandel-tareador`? Why the task graph generated for `mandeld-tareador` is so different? Which section of the code do you think is causing the serialization of all tasks? Change the Tareador instrumentation in order to isolate the part of the code that is causing the dependence.

3. Simulate the execution with Dimemas of the non-graphic version for the two task granularities decided in the previous point. To do that execute `./run_dimemas mandel-tareador n` being `n` the number of processors to simulate. Compute the speed-up when when using 2, 4, 8 and 16 processors with respect to the execution with 1 processor. Which task granularity scales better and which is the reason for the poor scalability?

4.2 Divide and conquer: Multisort

Multisort is a sort algorithm which combines a "divide and conquer" mergesort strategy that divides the initial list into multiple sublists recursively, a sequential quicksort that is applied when the size of these sublists is sufficiently small, and a "divide and conquer" merge of the sublists back into a single sorted list. First you should understand how the code `multisort.c` implements the "divide and conquer" strategy, recursively invoking functions `multisort` and `merge`. You will also analyze, using the Tareador tool, the potential parallelism that is available and its main characteristics.

1. Compile the sequential version of the program using "make multisort" and execute the binary. This will report you the three parameters that you need to provide to sort a list randomly initialized: size of the list in Kiloelements, size in Kiloelements that breaks the recursion to apply quicksort, and size in Kiloelements that breaks the recursion when doing the merge part. For example `./multisort 8192 512 512`. The execution generates an output file with the sorted vector that has been generated.

2. The `multisort.c` file is already prepared (ssgrind.h, start_tareador and end_tareador invocations using conditional compilation) to insert Tareador instrumentation. Complete the instrumentation to understand the potential parallelism that the "divide and conquer" strategies provide when applied to the sort and merge phases. To compile the code use the `multisort-tareador` target in the Makefile, which activates the conditional compilation guarded with 
"#if _TAREADOR_". Execute the binary generated using the `run-tareador.sh` script. This script accepts one argument (the name of the binary to execute) and uses a very small case to generate a reasonable task graph. Analyze the task graph generated and deduce the task synchronizations that are needed to enforce the dependences between the tasks that you have identified.