Tareador: a tool to unveil parallelization strategies at undergraduate level

Eduard Ayguadé, Rosa M. Badia, Daniel Jiménez, José R. Herrero, Jesús Labarta, Vladimir Subotic and Gladys Utrera
Computer Sciences Department, Barcelona Supercomputing Center (BSC-CNS)
Computer Architecture Department, UPC–BarcelonaTech
Barcelona, Spain
Contact author: eduard.ayguade@bsc.es

***Abstract*—This paper presents a methodology and framework designed to assist students in the process of finding appropriate task decomposition strategies for their sequential program, as well as identifying bottlenecks in the later execution of the parallel program. One of the main components of this framework is *Tareador*, which provides a simple API to specify potential task decomposition strategies for a sequential program. Once the student proposes how to break the sequential code into tasks, *Tareador* 1) provides information about the dependences between tasks that should be honored when implementing that task decomposition using a parallel programming model; and 2) estimates the potential parallelism that could be achieved in an ideal parallel architecture with infinite processors; and 3) simulates the parallel execution on an ideal architecture estimating the potential speed–up that could be achieved on a number of processors. The pedagogical style of the methodology is currently applied to teach parallelism in a third-year compulsory subject in the Bachelor Degree in Informatics Engineering at the Barcelona School of Informatics of the Universitat Politècnica de Catalunya (UPC) - BarcelonaTech.**

I. Introduction

For decades, single–core processors have been improving their performance doubling in speed every two years, without requiring major changes in the applications and relying on compilers to optimize the code for each target architecture. The move towards multicore architectures in less than a decade, has changed the scenario due to severe technological constraints and the difficulties to efficiently exploit the instruction level parallelism (ILP) on those single-core architectures. Multicores have introduced the need to re-design (parallelize) applications in order to utilize the increasing but still modest number of cores currently available [1]: neither the compiler nor the hardware can automatically detect and exploit the parallelism needed to feed them.

Parallel programming is not an invention of the multicore era. For decades, the high-performance community has been programming multiprocessor systems, without caring much about programming productivity. Tools to predict and analyze performance were also designed by and for the high-performance computing community. Also parallel computing was a subject just considered in courses at the advanced levels of computer science and engineering curricula. In the current scenario, in which current systems (from mobile to desktop/laptop to servers) are mostly based on parallel architectures, the lack of parallel programming expertise in the IT sector needs to be overcome in order to reach the efficiency and scalability that future generations of software will need to exploit those architectures.

Efforts are being done to introduce parallel programming to undergraduate students. However, both the analysis of potential concurrency (task and data decomposition, task ordering and data sharing constraints, ...) in applications and their parallelization using todays available programming models is still a challenge for those students. We believe that a methodology of study and tools to visually analyze and predict the potential parallelism in those applications are necessary; both should be designed to help these students in their first steps to understand the key issues in the parallelization process.

In this paper we first present the proposed methodology and framework that we propose for teaching parallelism to ”fresh” students and that has been successfully used to support teaching activities in *Parallelism*, a third-year compulsory subject in the Bachelor Degree in Informatics Engineering at the Barcelona School of Informatics (FIB) of the Universitat Politècnica de Catalunya (UPC) - BarcelonaTech. This subject is our first opportunity to teach parallelism at the undergraduate level. Then we present the main component in this framework, *Tareador*, which 1) offers an API to quickly define task decompositions; 2) provides information about the dependences that should be honored when implementing these task decompositions; and 3) computes the parallelism offered by the task decomposition and estimates the achievable speed–up on an ideal parallel architecture with a certain number of processors. The information provided by *Tareador* allows the student to specify task decompositions using a parallel programming model; in this introductory course we use *OpenMP* [2] version 4.0, which provides support for tasks (including task dependences) in addition to the traditional loop–level parallleism which is considered as a particular case of the generic tasking model. The framework also offers a dynamic instrumentation and tracing package (*Extrae* [3]) and a trace visualization tool (*Paraver* [3]) which allow the programmer to understand the actual behavior of the parallel decomposition when executed on a real parallel architecture. The complete framework motivates the learning process, improves the understanding of the proposed task decompositions and significantly reduces the time to reach parallel implementations of the original sequential codes.

The paper is organized as follows. Section II describes the context of the course where the methodology and framework are used. Then, Section III describes the main component in this framework (*Tareador*) and Section IV presents an example of use. Finally, section V presents some related work and section VI concludes the paper.

II. Context, course description and methodology

Parallelism is a third-year (fifth term) compulsory subject in the Bachelor Degree in Informatics Engineering. The subject concerns parallel programming models and parallel computer architectures; it comes after a series of courses on computer organization and architecture, programming and data structures. The term effectively lasts for 15 weeks. There are 4 contact hours per week: 2 hours dedicated to theory/problems (60 students per class) and 2 hours dedicated to laboratory (15 students per class). Students are expected to invest about 5-6 additional hours per week to do homework and personal study (over these 15 weeks). Thus, the total effort devoted to the subject is 6 ECTS credits[1]

There are eight guided laboratory assignments. The first four are to make students get familiar with the experimental environment: compilation, execution, performance measurement and analysis with *Paraver*, analysis of potential parallelism using *Tareador* when going from coarse- to fine-grain task decomposition strategies using a simple example, and a very practical *OpenMP* tutorial. The last four assignments are devoted to parallelizing four different applications (the last one is optional):

- Maldelbrot set (embarrassing parallel iterative task decomposition). Two different task granularities are evaluated (each row or point in the two–dimensional set) to observe the influence of task generation overheads. The program has load balancing issues due to differences in the cost of computing each point in the Mandelbrot set, suggesting the need of dynamic task/loop scheduling in *OpenMP*.
- Multisort (divide–and–conquer recursive task decomposition strategy). The divide–and–conquer strategy recursively splits the vector in four parts which are sorted with four totally independent invocations of sort. After these sort tasks end, two merge tasks follow, each one joining the results of two sort tasks. Their results are merged again with a final merge call. Task dependencies are the main concept in this code that can be satisfied with task barriers or task dependencies in *OpenMP*.
- Heat equation (geometric iterative task decomposition). Three different solvers are used: Jacobi, Red-Black and Gauss-Seidel. The program makes use of a two–dimensional data structure iteratively traversed using loop nests. Different synchronization constructs to support both loop and task parallel constructs are required in each solver, including reductions, barrier and point–to–point synchronization.
- Sudoku (branch–and–bound recursive task decomposition). The code is useful to show the need of data replication to enable exploratory parallelization strategies and the need to control task generation based on recursion depth or number of tasks to avoid excessive overheads.

After compiling and executing the sequential version, the students follow a similar approach in each of these four parallel programming assignments: a) analysis of potential task decompositions using *Tareador*; b) parallelization using task and work–sharing constructs using *OpenMP*; and c) understanding performance by analyzing weak and strong scalability of the implementations using *Extrae* and *Paraver* [3]. Table I shows the parallelization strategies that the students explore to express the different task decompositions in the applications listed above.

TABLE I
OpenMP parallelization strategy used in each application

	tasks	**dependent tasks**	**loop parallelism**
Mandelbrot set	X		X
Heat equation	X	X	X
Multisort	X	X	initialization
Sudoku game	X		

The *Tareador* framework is also used at the master level in *Concurrency, Parallelism and Distributed Systems*, another subject part of the first term of the two-year Master in Innovation and Research in Informatics in the same school. In that subject, *Tareador* is used as the driver to explore task decompositions to be expressed in *MPI* [4] and *CUDA* [5]. Finally, *Tareador* was also successfully exercised in a handson HPC Educators session at Supercomputing SC'12 and SC'13 [6].

III. Analyzing potential parallelism

Starting from a sequential specification of the program (in C/C++ or Fortran), the objective is to find a decomposition of the problem into pieces of work (tasks) that can be executed concurrently ensuring that the same results are produced. *Tareador* has been designed as a framework to help programmers in deciding the best decomposition strategy by 1) offering an API to quickly define potential task decompositions in the original sequential program; 2) providing information about the dependences that should be honored when implementing them using a parallel programming model; 3) estimating the potential parallelism that could be achieved in an ideal parallel architecture with infinite processors; and 4) simulating the parallel execution and estimating the potential speed–up that could be achieved on a number of processors. All that is done prior to actually implementing the task decomposition using a specific parallel programming model (e.g. *OpenMP* or *MPI*).

The annotated code is executed sequentially – all annotated tasks are executed in the order of their instantiation. *Tareador* dynamically instruments the sequential execution and collects

[1]The European Credit Transfer System (ECTS) is a unit of appraisal of the academic activity of the student. It takes into account student attendance at lectures, the time of personal study, exercises, labs and assignments, together with the time needed to do examinations. One ECTS credit is equivalent to 25-30 hours of student work.

a log with the data regions accessed by each potential task. Once the log is generated, *Tareador* calculates inter-task dependencies and evaluates the potential parallelism of the decomposition providing the results to the user in the form of:

- task dependency graph with all task instances represented in a hierarchical way (task nesting is allowed in the API);
- visualization of the data accessed by each task;
- simulation of the parallel execution when using a number of processors.

Through a refinement process, students can easily explore different task decompositions, from very coarse- to fine-grained ones, and analyze the influence of some algorithmic parameters.

A. Tareador API

The input to *Tareador* is a sequential application with simple annotations (in this paper we just present the API for C/C++) that specify a possible task decomposition for the sequential code. The API provides two functions to initialize and finalize *Tareador*:

```
tareador_ON();
      ...
tareador_OFF();
```

and two functions to specify task boundaries:

```
tareador_start_task(name_string);
      ...
tareador_end_task(name_string);
```

These functions allow the specification of any arbitrary task decomposition (nesting of tasks is supported). `name_string` is used to provide a name to each task and can be dynamically built (e.g. using `sprintf` from the `libc` library) in order to use different labels. Initially this process does not require any refactoring of the sequential code; however, some code refactoring (e.g. privatization of data structures) could be useful in order to remove some dependences, easing the later parallel coding step. The API provides a couple of functions to filter objects, removing them from the *Tareador* dependence analysis:

```
tareador_disable_object(address of object);
      ...
tareador_enable_object(address of object);
```

The programmer can filter a data object to discover the parallelism of the application if he/she already knows how to protect the dependences (data sharing) that are generated by that object.

B. Tareador implementation

The *Tareador* environment integrates various internally and externally developed tools. The framework (Figure 1) takes the annotated sequential code and compiles it with an LLVM-based [7] compiler. The execution of the binary generates the information that is processed, once the execution is finished, by the *Tareador* backend embedded into the *Tareador* GUI.

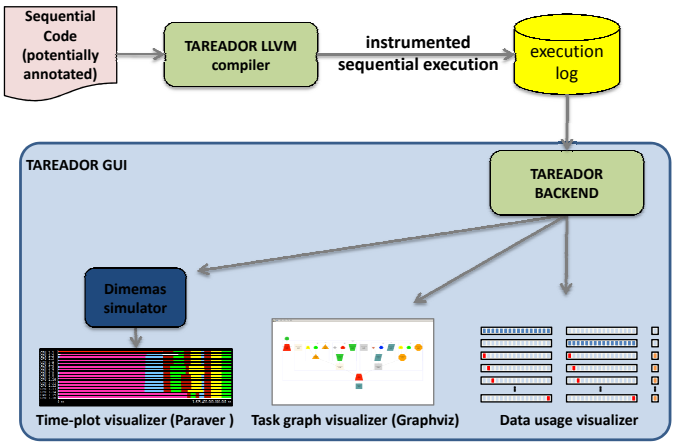


Fig. 1. Tareador framework

This *Tareador* backend produces the task dependency graph, which can be queried to find out the cause of each data dependence in the graph (dataview). The *Tareador* backend also generates an execution trace that feeds the *Dimemas* [8] simulator which generates *Paraver* [9] time-plots showing the potential parallel execution.

The *LLVM*-based compiler dynamically instruments the sequential application in order to collect information about all annotated tasks and the memory locations accessed. The compiler allows the user to manually annotate tasks by wrapping arbitrary code sections using the described *Tareador* API. Furthermore, by intercepting all allocations and releases of the memory, the tool maintains the pool of all alive memory objects. The instrumentation tracks all memory objects, intercepting and recording accesses to them at the granularity of one byte. The instrumentation introduces a non-negligible overhead (1000x slowdown) which suggests the use of small input datasets.

The *Tareador* backend is a *Python* program that consumes the logs generated by the instrumentation and produces inputs for the different modules in the *Tareador* GUI. Given the result of the instrumentation, *Tareador* identifies all task instances and read-after-write dependencies among them. As a result, the backend generates the trace with all the information necessary for *Dimemas* to simulate the parallel temporal behavior on a configurable ideal target architecture. The trace contains bursts that correspond to instances of the programmer-annotated tasks and the inter-task synchronizations based on the identified inter-task dependences.

C. Tareador example: dot product

This subsection illustrates the main features and views provided by *Tareador* using a very simple dot product example. The annotated source code is shown in Figure 2, in bold font showing the annotations introduced to specify tasks at different task granularities. The code to initialize the two vectors `A` and `B`, as well as the function computing the `dot_product` are initially marked as potential tasks. For a finer-grain task decomposition, the programmer could also specify as a task

```
void dot_product (long N, double A[N], double B[N],
                  double *acc) {
  double prod;

  *acc=0.0;
  for (int i=0; i<N; i++) {
    // tareador_start_task("dot_product_loop");
    prod = my_dot(A[i], B[i]);
    // tareador_disable_object(acc);
    *acc += prod;
    // tareador_enable_object(acc);
    // tareador_end_task("dot_product_loop");
  }
}

int main(int argc, char **argv) {
    double result;

    tareador_ON ();
    tareador_start_task("init_A");
    for (int i=0; i< size; i++) A[i]=i;
    tareador_end_task("init_A");

    tareador_start_task("init_B");
    for (int i=0; i< size; i++) B[i]=2*i;
    tareador_end_task("init_B");

    tareador_start_task("dot_product");
    dot_product (size, A, B, &result);
    tareador_end_task("dot_product");

    tareador_OFF ();
}
```

Fig. 2. Annotated sequential code for a simple dot product

each iteration of the loop inside the `dot_product` function, as shown with the commented lines.

For the coarsest-grain decomposition, Figure 3.a shows the task graph obtained by *Tareador*, a directed acyclic graph where each node represents a task instance and each edge represents a data dependence between two task instances. In the task graph, the size of the node is proportional to the number of instructions executed in the task. In this graph, the green and red nodes represent tasks `init_A` and `init_B`, respectively, while the yellow node represents the single instance of `dot_product`. The graph shows that `dot_product` depends on both `init_A` and `init_B`, with no much parallelism to be exploited. If tasks are refined, so that a task corresponds to one iteration of the loop inside function `dot_product`, the we obtain the task graph in Figure 3.b. By clicking on each of the edges between a yellow node and the others we obtain *dataview* windows similar to the ones shown in Figure 3.c, which reveal the variables (or portion of them) that cause the dependences. Observe that the serialization between two consecutive iterations of the loop (yellow tasks) is caused by the accumulative operation on variable `result` (which in the code is accessed through pointer `acc`). If we disable the analysis of this variable in the excerpt of code between the `tareador_disable_object` and `tareador_enable_object` calls (initially commented in the code in Figure 2) then we obtain the task dependence graph in Figure 3.d. By filtering this object the programmer is assuming that he/she will guarantee the dependence (data sharing without race condition) in some way (for example using the `reduction` clause in *OpenMP*).

The second *Tareador* output is the timeline of the simulated potential parallel execution when using a specified number of processors (4 in Figure 3.e). The timeline shows, for each of the 4 cores in the parallel machine (y-axis), which task is executed at any time (x-axis). The colors representing task types match the colors used in the task dependency graph.

I

IV. Heat diffusion example

This section presents one of the codes used in the undergraduate course mentioned before. The code simulates heat diffusion in a solid body using three different 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 the example, and present different parallel behaviors.

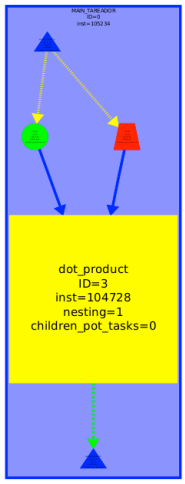
The code at the top in Figure 4 shows the iterative loop executing one of the three solvers on a 2D data array `u` of size `np` by `np`. When Jacobi is used, the solver returns the result in an auxiliary data array `uhelp`, which is copied to the original `u`. The other two solvers return their result in place. Solvers also compute a `residual` that is used to determine convergence and the termination of the iterative loop. The iterative loop also finishes if convergence is not reached after a certain number of iterations `maxiter`.

The listing at the bottom in Figure 4 shows the annotated code for the *Gauss-Seidel* solver. In this solver, matrix `u` is updated in place, causing the dependences among the computation of blocks shown in the task graph in Figure 5 (for `NB=8` in this case). As we did for `dot_product`, we have filtered the analysis for object `sum` (in *OpenMP*, that can be done, for example, using a reduction clause). The student can observe how the computation of each block depends on its neighbors and the data regions that cause the dependences.

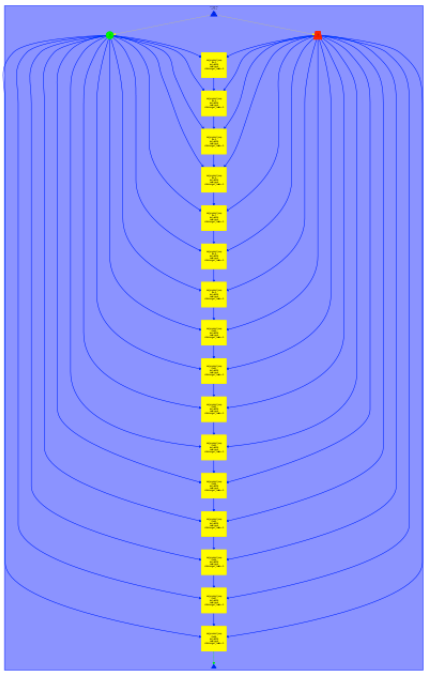
For the *Gauss-Seidel* solver, Figure 6 shows the timelines obtained by *Tareador* when simulating the parallel execution using 2, 4 and 8 processors. All three timelines are in the same temporal scale, so the student can observe good scalability when going from 1 to 2 and from 2 to 4 processors. However, using more than 4 processors is not scaling well in this case due to having an insufficient number of blocks.

A. Other examples

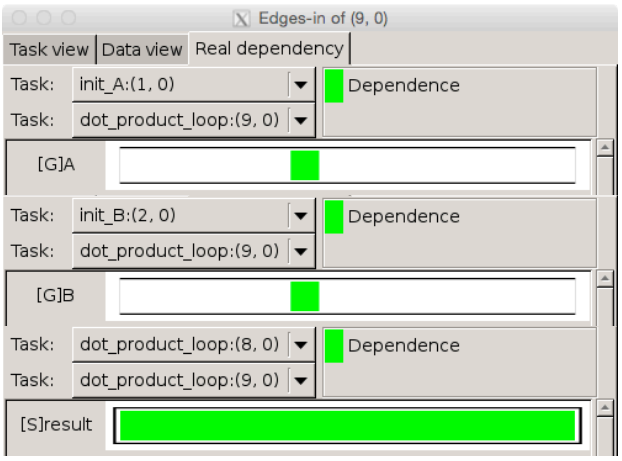
To conclude this section about the heat diffusion example, Figure 7 shows the task graphs generated by *Tareador* for two of the other codes used in the undergraduate course. Specifically, Figure 7.a shows the embarrassingly parallel nature of the *Mandelbrot* program as is observed in the corresponding task graph obtained by *Tareador*. However, notice that nodes have different sizes, which should suggest to the students the existence of a load unbalance problem and the need to use some dynamic loop schedule that tries to fight against that. The task graph in Figure 7.b shows how the divide and conquer nature in the *Multisort* program results in four totally independent invocations of sort (rectangular boxes), each one



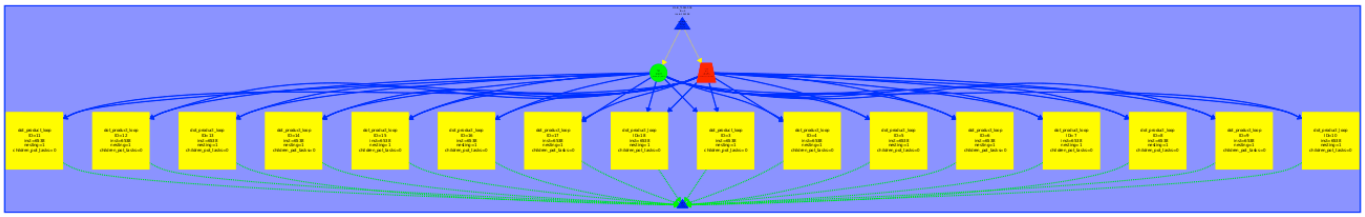
(a) Coarse-grain task dependence graph



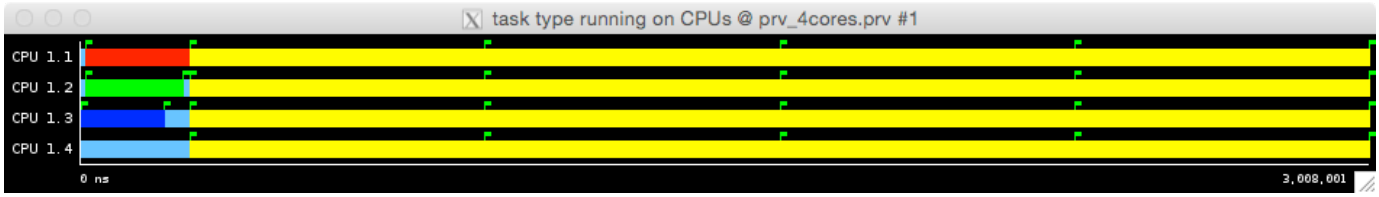
(b) Fine-grain task dependence graph



(c) Data dependences for a node



(d) Fine-grain task dependence graph (if the dependences caused by `result` are filtered)



(e) Simulated parallel execution with 4 processors

Fig. 3. Applying Tareador on the dot product kernel.

dividing again in the same 4 independent calls (red circle, yellow circle, green square and red trapezoid). After the sort tasks the merge tasks follow, joining two sort tasks (pink and green boxes) or two previous join tasks (magenta hexagon).

V. Related work

There is a wide discussion on the need of introducing parallel programming concepts to undergraduate education and the best way to expose them. Such concepts are often not easy to be understood by early students, specially if they come with a strong sequential programming background. There is much effort dedicated to analyzing whether parallel programming concepts have to be taught during several courses on programming, or there should be a dedicated course introducing all the concepts at once [10], [11], to cite a few. Experiences, methods, pedagogical approaches, tools and techniques for teaching parallel and distributed computing topics in the Computer Science and Engineering curriculum are collected in the *EduPar* workshop series [12], with special emphasis on undergraduate education in the last couple of years.

```
iter = 0;
while(1) {
   switch( param.algorithm ) {
      case 0: // JACOBI
         residual = relax_jacobi(param.u, param.uhelp,
                                 np, np);
         // Copy uhelp into u
         for (i=0; i<np; i++)
             for (j=0; j<np; j++)
                param.u[i*np+j] = param.uhelp[ i*np+j ];
         break;
     case 1: // GAUSS
         residual = relax_gauss(param.u, np, np);
         break;
     case 2: // RED-BLACK
         residual = relax_redblack(param.u, np, np);
         break;
 }
 iter++;
 // solution good enough ?
 if (residual < 0.00005) break;

 // max. iteration reached ? (no limit with maxiter=0)
 if (param.maxiter>0 && iter>=param.maxiter) break;
}
```

```
double relax_gauss (double *u, unsigned sizex,
        unsigned sizey) {
double unew, diff, sum=0.0;
int nbx, bx, nby, by;
int ii, jj, i, j;

nbx = NB; nby = NB;
bx = sizex/nbx; by = sizey/nby;
for (ii=0; ii<nbx; ii++)
    for (jj=0; jj<nby; jj++) {
       sprintf(stringMessage,"Gauss (%d-%d)",ii, jj);
       tareador_start_task(stringMessage);
        for (i=1+ii*bx; i<=min((ii+1)*bx, sizex-2); i++)
           for (j=1+jj*by; j<=min((jj+1)*by,sizey-2);j++){
              unew= 0.25 * (u[ i*sizey     + (j-1) ]+
                            u[ i*sizey     + (j+1) ]+
                            u[ (i-1)*sizey + j     ]+
                            u[ (i+1)*sizey + j     ]);
              diff = unew - u[i*sizey+ j];
              tareador_disable_object(&sum);
              sum += diff * diff;
              tareador_enable_object(&sum);
              u[i*sizey+j]=unew;
           }
       tareador_end_task();
   }
return sum;
}
```

Fig. 4. Excerpts of code from the Heat diffusion example

We believe that *Tareador* is a tool that can help students in this gradual transition from sequential computing to parallel computing, visualizing if different sections in their code can be simultaneously executed and which are the reasons that preclude their parallel execution. We also believe their learning experience is further enhanced through the examples that we have selected covering different kind of task decompositions. We also promote the use of a directive-based parallel programming model (e.g. *OpenMP*) as the way to teach parallelism when coming from sequential programming courses. For example [13] states that with *OpenMP* students find it easier to exploit concurrency.

Some tools are available, both from academia and industry, to assist programmers in the parallelization process. None of them is specially designed for undergraduate students so we

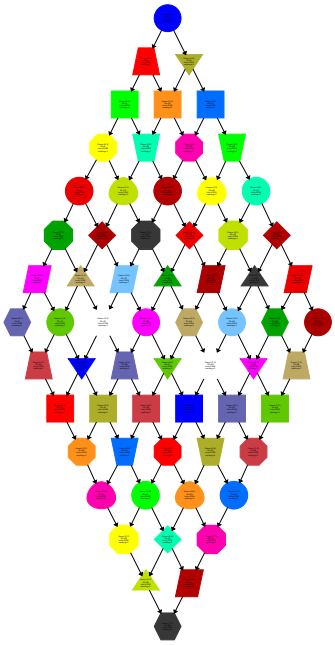


Fig. 5. Task graph for the Gauss-Seidel solver

believe that they don't really help to bridge the gap between sequential and parallel programming mentioned before. For example, Embla [14] is a Valgrind-based tool that estimates the potential speed-up for Cilk programs. On the other hand, Kremlin [15] identifies regions of a serial program that can be parallelized with OpenMP and proposes a parallelization planner for the user to parallelize the target program. The major drawback of both tools is that they are limited to fork-join loop-level parallelism.

Companies have also been recently developing solutions for assisted parallelization. For example, Intel's Parallel Advisor [16] assists parallelization with Thread Building Blocks (TBB) [17]. Parallel Advisor provides timing profile that suggests to the programmer which loops should be parallelized. Critical Blue provides Prism [18], a tool to do "what-if" analysis that anticipates the potential benefits of parallelizing certain parts of the code. Vector Fabrics provides Pareon [19], another tool for "what-if" analysis to estimate the benefits of parallelizing loop iterations. All the three mentioned tools provide rich GUI and visualization of the potential parallelization, although not tailored to early adopters.

Although not offered to early adopters, *Tareador* is also able to automatically explore parallelization strategies and find the one that exposes the highest potential parallelism [20]. Two new steps are added in this version of *Tareador*: one that dynamically instruments the sequential application identifying all potential regions for parallel execution (loops and function invocations) and another one that performs the automatic exploration of parallelization strategies driven by some heuristics and cost metrics (task length, number of task dependencies and task concurrency).

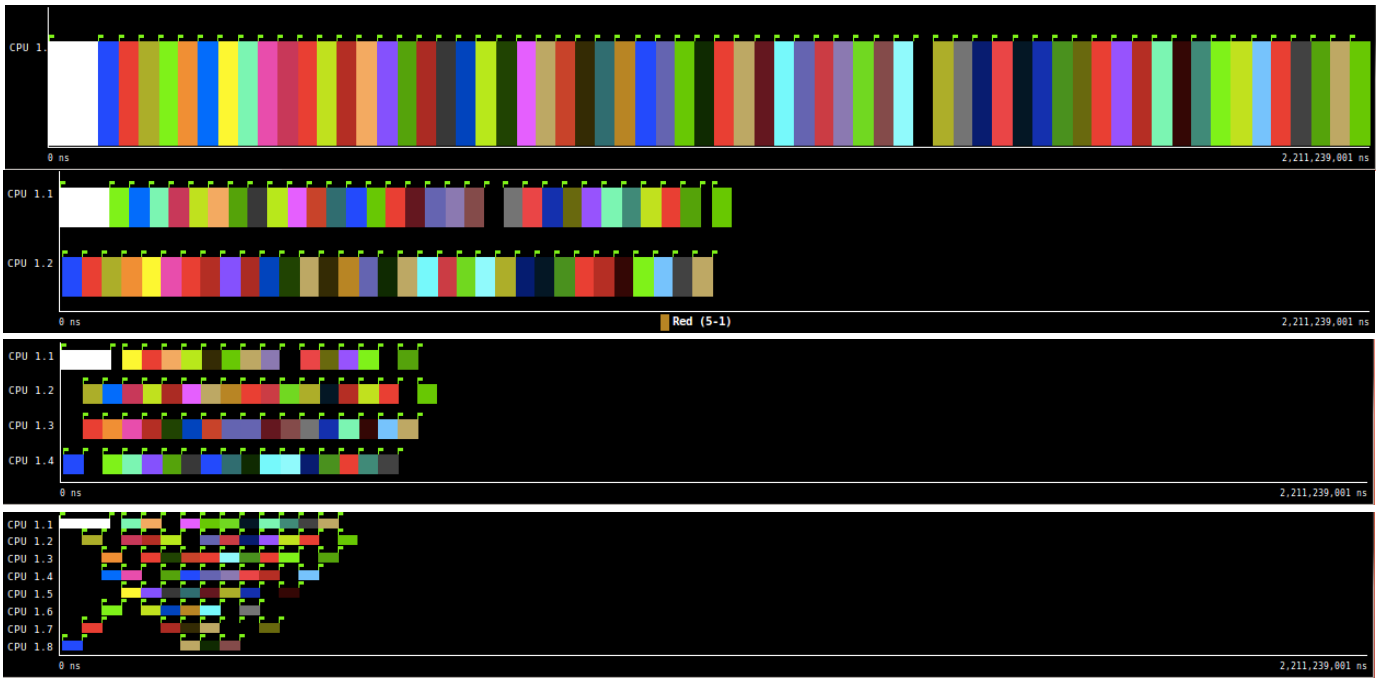
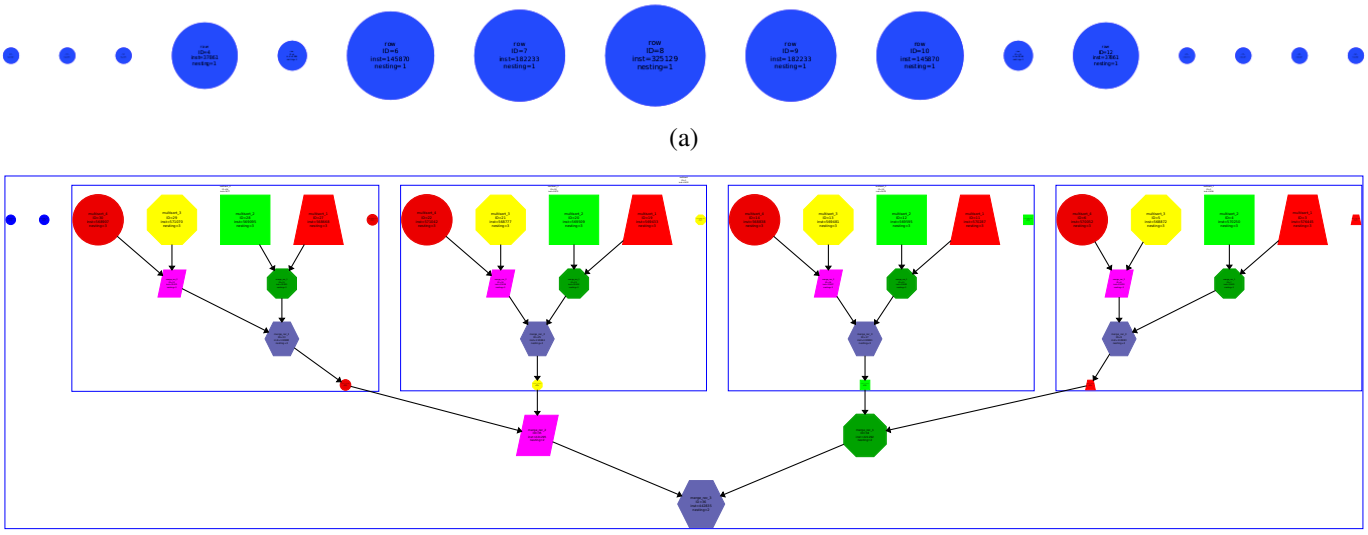


Fig. 6. Visualization of the simulated parallel execution for 1, 2, 4 and 8 processors for the *Gauss-Seidel* solver



(a)

(b)

Fig. 7. Tasks graphs for mandelbrot set (upper) and multisort (lower)

VI. Conclusions

In this paper we have presented a methodology with a complete framework, that significantly reduces the effort required by students to understand different task decomposition strategies and their constraints in terms of task orderings and data sharing. The framework offers an API to quickly define task decompositions, provides information about the dependences that should be honored when implementing them using a parallel programming model, and performs estimates of the potential parallelism and speed–up that could be achieved in an ideal parallel architecture with a certain number of processors.

We have described the main components of the system (with major focus on *Tareador*) and provided a complete example of use. Our experience in a third-year undergraduate subject shows that students are able to perform the analysis and the parallelization in the hours assigned to each of the assignments, plus the time required to finish the analysis and writing of the deliverable at home. In general, students usually spend less than 2 hours to analyze the different task decompositions,

their potential parallelism and dependences, task ordering and data sharing analysis. The rest of the time is devoted to defining the task decomposition and task ordering/data sharing constraints using *OpenMP* and understanding the performance that is achieved.

We believe that *Tareador* is a tool that helps students in this gradual transition from sequential computing to parallel computing, visualizing if different sections in their code can be simultaneously executed and which are the reasons that preclude their parallel execution. Based on the experience of use, we are in the process of designing an introductory parallel programming MOOC course based on the use of *Tareador*, using similar examples to the ones commented in this paper, through its web portal [21] for experimentation. The tool is also available for installation on Linux platforms through the same web portal, as well as *Extrae* and *Paraver* through the Barcelonal Supercomputin Center tools website [3].

ACKNOWLEDGMENTS

We thank the other professors in the *Parallelism* course (Nacho Navarro, Jordi Tubella and Jordi Garcia) for their feedback on the laboratory sessions where *Tareador* and associated methodology are used. We also thank the two developers (Arturo Campos and Alejandro Velasco) that have been working on different parts of *Tareador*. This work has been supported by the grant SEV-2011-00067 of the Severo Ochoa Program, awarded by the Spanish Government, by the Spanish Ministry of Science and Innovation (contract TIN2012-34557) and by Generalitat de Catalunya (contracts 2014-MOOC-00057 and 2014-SGR-1051).

REFERENCES

[1] K. Asanovic, R. Bodik, J. J. Catanzaro, Bryan Christopher an d Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Y elick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[2] OpenMP Architecture Review Board. OpenMP 4.0 Specification. [Online]. Available: http://www.openmp.org/

[3] Barcelona Supercomputing Center. BSC Performance Tools. [Online]. Available: https://www.bsc.es/computer-sciences/performance-tools

[4] R. L. Graham, "The MPI 2.2 Standard and the Emerging MPI 3 Standard," in *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 2–2.

[5] Nvidia. CUDA Parallel Computing Platform. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[6] E. Ayguadé and R. M. Badia, "Unveiling parallelization strategies at undergraduate level," *HPC Educator, Supercomputing 2012*. [Online]. Available: http://pm.bsc.es/SC12_training_session

[7] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Intl. Symp. on Code Generation & Optimization*, San Jose, CA, USA, Mar 2004, pp. 75–88.

[8] Barcelona Supercomputing Center. Dimemas: performance analysis tool for message-passing programs. [Online]. Available: http://www.bsc.es/computer-sciences/performance-tools/dimemas/general-overview

[9] ——. Paraver: a flexible performance analysis tool. [Online]. Available: http://www.bsc.es/computer-sciences/performance-tools/paraver/general-overview

[10] C. Brown, Y.-H. Lu, and S. Midkiff, "Introducing parallel programming in undergraduate curriculum," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 1269–1274.

[11] A. Minaie and R. Sanati-Mehrizy, "Incorporating parallel computing in undergraduate computer science curriculum," in *Proceedings of American Society for Engineering Education Annual Conference*, 2009.

[12] NSF/TCPP, "Workshop on Parallel and Distributed Computing Education," http://cs.gsu.edu/ tcpp/curriculum/?q=edupar.

[13] H. de Freitas, "Introducing parallel programming to traditional undergraduate courses," in *Frontiers in Education Conference (FIE)*, Oct 2012, pp. 1–6.

[14] J. Mak, K.-F. Faxén, S. Janson, and A. Mycroft, "Estimating and Exploiting Potential Parallelism by Source-Level Dependence Profiling," in *Euro-Par (1)*, 2010, pp. 26–37.

[15] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: rethinking and rebooting gprof for the multicore age," in *PLDI*, 2011, pp. 458–469.

[16] Intel Corporation., "Intel Parallel Advisor," http://software.intel.com/en-us/intel-advisor-xe, active on 10.11.2014.

[17] C. Pheatt, "Intel threading building blocks," *J. Comput. Sci. Coll.*, vol. 23, no. 4, pp. 298–298, apr 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1352079.1352134

[18] Critical Blue., "Prism," http://www.criticalblue.com/, active on 10.11.2014.

[19] Vector Fabrics., "Pareon," http://www.vectorfabrics.com/products, active on 10.11.2014.

[20] V. Subotic, A. Campos, A. Velasco, E. Ayguade, J. Labarta, and M. Valero, "The unbearable lightness of exploring parallelism," in *8th International Parallel Tools Workshop*, Stuttgart, Germany, October 2014.

[21] Barcelona Supercomputing Center. Tareador Portal. [Online]. Available: http://pm.bsc.es/tareador/service