Teaching Parallel Programming in Interdisciplinary Studies

Eduardo Cesar⋆, Ana Cortés, Antonio Espinosa⋆⋆, Tomàs Margalef, Juan Carlos Moure, Anna Sikora, and Remo Suppi⋆⋆⋆

Computer Architecture and Operating Systems department,
Universitat Autònoma de Barcelona, 08193 Cerdanyola del Vallés, Spain
`{eduardo.cesar,ana.cortes,antoniomiguel.espinosa,tomas.margalef,`
`juancarlos.moure,anna.sikora,remo.suppi}@uab.cat`

Abstract. Nowadays many fields of science and engineering are evolving by the joint contribution of complementary fields. Computer science, and especially high performance computing, has become a key factor in the development of many research fields, establishing a new paradigm called computational science. Researchers and professionals from many different fields require a knowledge of high performance computing, including parallel programming, to develop a fruitful work in their particular field. So, at Universitat Autònoma of Barcelona, an interdisciplinary master on Modeling for science and engineering was started 5 years ago to provide a deep knowledge on the application of modeling and simulation to graduate students on different fields (Mathematics, Physics, Chemistry, Engineering, Geology, etc.). In this master, parallel programming appears as a compulsory subject, because it is a key topic for them. The concepts learnt in parallel programming must be applied to real applications. Therefore, a subject on Applied Modelling and Simulation has also been included. In this paper, the experience on teaching parallel programming in such interdisciplinary master is shown.

Keywords: Parallel programming, Message passing, Shared Memory, GPUs, MPI, OpenMP, CUDA

1 Introduction

Many fields of science and engineering are applying the techniques and recent advances in complementary fields. In this interdisciplinary context, researchers and professionals with a wider knowledge are highly demanded by companies

and research centres. Universitat Autònoma of Barcelona started a Master degree on Modeling for Science and Engineering to provide such kind of professionals to those companies. The students enroll from different degrees: Mathematics, Physics, Chemistry, Engineering, Computer Science, and others. The master studies has been very successful, attracting students from many countries of origin. Also, companies are actively hiring finishing students and providing internships for students while attending the master.

A key point in the success of the master is the strong collaboration with companies applying such techniques and methods in their everyday business processes, so that the students can realize the significant impact of such techniques and methods and economical savings provided in a specific company.

The master itself involves an interdisciplinary collaboration among professors from different departments; mainly Physics, Mathematics, and Computer Architecture and Operating Systems. Its objective is to provide the students the basic knowledge to be able to model a physical system, to represent it in a mathematical way, to solve it applying different methods such as differential partial equations, optimisation, time series, and related methods. Finally, they program some implementations as high performance computing applications or simulations. The main three pillars of the training of the students are:

- Definition of complex systems
- Mathematical representation and resolution of such systems
- High performance computing implementation of designed solutions, typically by means of a software simulation tool

In this context, we strive for a good training in parallel programming and a successful experience with the efficiency analysis of the implementation of some real small-size applications.

This paper focuses on the description of the training on parallel programming and on applied modelling and simulation that is offered to the students. In section 2 we present the basic concepts on parallel programming shown to the students to establish a common framework. Section 3 describes the parallel programming approaches taught to the students and the lab exercises proposed to them. Then, Section 4 presents some examples of real applications that are shown and proposed to the students for their development. Finally, Section 5 presents the main conclusions of this teaching experience.

2   Basic concepts for interdisciplinary students

Training parallel programming is a core subject in this interdisciplinary master, but the first challenge is to set a common background for the students. The students of the master typically have some programming knowledge of high level languages such as Java or Python, but usually they have a limited knowledge of C programming language. Since C is in the core of high performance computing, the very first part of the course is devoted to introduce to the students its main concepts and to work on several programming exercises. Usually, students

succeed in this initial training part due to their high interest and their previous programming experience. Our previous experiences have showed us that devote some time for setting this basic C knowledge must become a hard requirement before introducing OpenMP or MPI programming.

Once the students have learned the C programming principles, it is necessary to introduce them the basic concepts of parallel programming. The first point is the general idea of parallelism itself and how HPC computing platforms are designed. So, a general idea about parallel and distributed systems, multi-core processors, memory hierarchy and GPUs, are presented to the students. These objectives become a challenge because it is necessary to provide the students useful real architecture concepts while avoiding too deep architectural details that are complex to relate to programming issues and may bore them and become a serious problem. For this reason, we provide a gentle introduction with selected further readings for those students particularly interested in these architectural aspects.

The following point in the subject is an introduction to parallel algorithms. Some of the students have some experience in sequential programming or even in object oriented programming, but the computational aspects of parallel algorithm design must be introduced to the students, showing them different current paradigms and related tools.

We provide details on several parallel algorithms for different problems. The first problem considered is matrix multiplication, which most of them know very well and have already programmed in a sequential way. We start by showing them how the problem is inherently parallel. Several matrix multiplication parallel algorithms are shown and analysed considering different aspects such as computational complexity, communication requirements, data size and memory requirements. These different algorithms are analysed considering the previously mentioned architectural aspects, showing the implications of computing capabilities, communication network and memory limitations.

Along the course we identify several important parallel computation patterns [17], which are used in many examples. The *map* pattern is exemplified by the vector addition algorithm (and the outer loops of matrix multiplication). It becomes an appropriate pattern to introduce parallelism as it does not involve any dependence nor communication among threads. The *reduce* pattern is studied in the inner loop of matrix multiplication, we use it to introduce the problem of synchronization and the idea of re-associating arithmetic operations to increase parallelism. The *stencil* pattern is used to simulate the movement of a string, and requires synchronization and to share and communicate boundary data.

Two additional parallel computation patterns are studied by means of exercises proposed to the students. The parallel prefix algorithm (*scan* pattern) and the convex hull problem (*divide and conquer* or *recursive* pattern) are proposed so that students analyse the problem and find out the potential parallelism in the algorithm. The students compare their proposals considering aspects such as algorithm complexity, memory and communication requirements, and so on.

When the students have understood all this basic concepts, it is possible to tackle the parallel programming challenge. This topic is presented in next section.

3   Parallel programming

Once the basic concepts of programming and parallelism have been presented to the students, it is possible to enter in the core part of parallel programming. In this part three paradigms are presented: Shared memory, Message passing and Accelerator-oriented kernel programming (GPUs). The rationale for this organisation is that developing programs with a shared memory model, such as OpenMp, requires a simple modification of a C sequential program by including just some directives. So, the students can parallelise their sequential C programs in just one lab session. After OpenMP, MPI is introduced. In this case, it is necessary to think how to paralelise the algorithm, which processes must be defined, how such processes must communicate, and so on. This implies a larger effort from the students. The last approach introduced is CUDA, as a programming model for GPUs (accelerators), that requires a more detailed understanding of memory hierarchy and the coordinated use of thousands of threads to reach relevant performance gains. The programming sessions are complemented with the introduction of performance analysis tools to understand the benefits of parallel programming and to detect and correct performance bottlenecks. The development of this topics is covered in the following subsections.

3.1   Shared memory: OpenMP

As mentioned above, once students are familiarized with C and basic concepts about parallel algorithms, the most natural way for introducing parallel applications development is using OpenMP [6].

OpenMP is a portable and flexible directive-based API for shared-memory parallel programming which, for some basic code constructions, allows to express parallelism in an extremely simple way. Given these characteristics, it has become the de-facto standard for multicore share-memory architectures. In addition, current laptops and desktop computers have multicore processors and, consequently, students can test all the examples given in class and develop new ideas in their own computer.

After a few motivating examples, such as the one shown in Listing 1.1, the contents of the OpenMP lecture are structured as follows:

- **Introduction**. Concept of thread, shared and private variables, and need for synchronization.
- **Fork-join model**. The `#pragma omp parallel` clause. Introducing parallel regions.
- **Data parallelism: parallelizing loops**. The `#pragma omp for` clause. Data management clauses (`private`, `shared`, `firstprivate`).

- **Task parallelism: sections**. The `#pragma omp sections` and `#pragma omp section` clauses.
- **OpenMP runtime environment function calls**. Getting the number of threads of a parallel region, getting the thread id, and other functions.
- **Synchronization**. Implicit synchronization, `nowait` clause. Controlling executing threads, `master`, `single`, and `barrier` clauses. Controlling data dependencies, `atomic` and `reduction` clauses.
- **Performance considerations**. Balancing threads' load, `schedule` clause. Eliminating barriers and critical regions.

Listing 1.1. OpenMP simple example: adding two vectors.

```
#pragma omp parallel for
for ( i = 0; i < N; i++ )
          c[i] = a[i] + b[i];
```

The concepts introduced in this lecture are reinforced in a lab session, where students must use OpenMP for parallelizing the code for simulating the movement of a string developed in the C labs (see Listing 1.2). In this way, students continue their work and can experience the advantages of using the 4 cores available in each lab equipment.

Listing 1.2. String simulation main computation loop.

```
for  (t=1; t<=T;  t++) {
    for  (x=1; x<X;  x++)
        U3[x] = L2*U2[x] + L*(U2[x+1]+U2[x−1]) − U1[x];
    double *TMP =U3;
    // rotate usage of vectors
    U3=U1;  U1=U2;  U2=TMP;
}
```

Parallelizing this code with OpenMP is straightforward as can be seen in Listing 1.3, its only complexity is that the clause `firstprivate(T,U1,U2,U3)` must be used to ensure that each thread does the same vector rotation using its private copies. This parallelization is specially designed to be done in a short time, leaving students plenty of opportunities to test the code and analyze its behavior.

Listing 1.3. Parallelized string simulation main computation loop.

```
#pragma omp parallel firstprivate(T,U1,U2,U3)
for  (t=1; t<=T;  t++) {
    #pragma omp for
    for  (x=1; x<X;  x++)
        U3[x] = L2*U2[x] + L*(U2[x+1]+U2[x−1]) − U1[x];
    double *TMP =U3;
    // rotate usage of vectors
    U3=U1;  U1=U2;  U2=TMP;
}
```

3.2 Message passing: MPI

After explaining parallelism at multi-core level using shared memory, the next step is to introduce cluster parallelism (distribute memory) using message passing. With this objective, the course includes two lectures on Message Passing Interface (MPI) [4] and two lab sessions for developing related exercises.

MPI is by far the most used interface for developing distribute memory parallel programs, mainly because many libraries have been implemented based on the MPI consortium specification (OpenMPI, MPICH, Intel MPI, etc.). MPI includes plenty of features but this course focuses on presenting the basic MPI program structure and the functions for point-to-point as well as collective communication.

The contents of the MPI lectures are structured as follows:

- **Message passing paradigm**. Distributed memory parallel computing, the need for a mechanism for interchanging information. Introducing MPI history.
- **MPI program structure**. Initializing and finalizing the environment `MPI_Init` and `MPI_Finalize`. Communicator's definition (`MPI_COMM_WORLD`), getting the number of processes in the application (`MPI_Comm_size`) and the process rank (`MPI_Comm_rank`). General structure of an MPI call.
- **Point-to-point communication**. Sending and receiving messages (`MPI_Send` and `MPI_Recv`). Sending modes: standard, synchronous, buffered and ready send.
- **Blocking and non-blocking communications**. Waiting for an operation completion (`MPI_Wait` and `MPI_Test`).
- **Collective communication**. Barrier, broadcast, scatter, gather and reduce operations.
- **Performance considerations**. Overlapping communication and computation. Measuring time (`MPI_Time`). Discussion on the communication overhead. Load balancing.

Students work around these concepts in the lab sessions by developing a simple program for computing π approximation using the dartboard approach. This approach simulates throwing darts to a dartboard on a square backing. As each dart is thrown randomly the ratio of darts hitting the board to those landing on the square is equal to the ratio between the two areas, as shown in Figure 1, which is $\pi/4$.

A parallel implementation of this algorithm can consists of a certain number of processes throwing a fixed number of darts and calculating its own approximation of π, then, one of the processes (the master) receives all approximations and calculates the average value. In this solution workers send their results to the master (process with rank 0) using point-to-point communication.

A second approach consists in distributing the total number of throws among all processes, each of them will calculate its number of hits (darts in the circle) and send it to the master process, which will compute π approximation. In this case, the master sends the number of throws that must be done by each process and receives the number of hits using always collective communication functions.
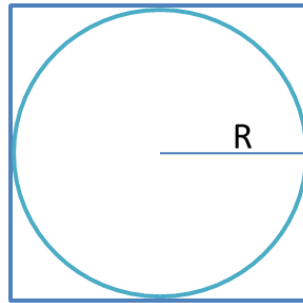
Fig. 1. Dartboard π approximation.

3.3   GPUs: CUDA

CUDA is an extension for massively parallel programming of GPUs (or accelerators). We choose CUDA instead of OpenCL because of the existence of efficient and mature compiling, debugging and profiling tools, and because of the extensive information available. The contents of the CUDA lectures are structured as follows:

- **Introduction**. Hierarchy of threads: warp, CTA (Cooperating Thread Array) and grid. 3-dimensional thread identifiers.
- **Model of an accelerator: host and device**. Moving data among host and device. Allocating memory on the device and synchronizing the execution.
- **Architectural restrictions**. Warp size. Maximum CTA and grid dimensions.
- **Memory space**. Global, local and shared memory.
- **Synchronization**. Warp-level and CTA-level synchronization.
- **Performance considerations**. Excess of threads to tolerate the latencies of data dependencies. Increasing work per thread to improve instruction-level parallelism.

The lecture uses vector addition as an example to introduce the CUDA syntax. Four implementations are provided and evaluated using: (a) one single thread, (b) one CTA, (c) a grid of CTAs where each thread performs a single addition, and (d) a grid of CTAs with more work per thread. We show the performance results (deceiving, for the first implementations) to motivate the different solutions and the need for developing good performance engineering skills.

We present Thrust [11], a high-level parallel algorithms library written in C++, to show the students the benefit of learning object-oriented programming and software engineering concepts. However, due to the limited background of our students and obvious time limitations, it is out of the scope of our course.

Students must use CUDA in a lab session for parallelizing the code that simulates the movement of a string. They explore, step by step, the different obstacles they must save to exploit the full potential of GPUs and increase performance 10x with respect to the multicore CPU code.

3.4   Performance analysis: tools

It is not only important to be able to develop applications using the different approaches taught during the course. Parallel programming main goal is to improve applications performance and, consequently, performance analysis should be introduced to students.

During the course labs, students use basic tools, such as nvprof [1], perf Linux command [8], jumpshot [10] and likwid [3] for visualizing and analyzing the behavior of their applications. These tools are enough for the simple applications developed and the small cluster used in this course.

However, our students will likely participate in the development of real parallel applications during their professional life. Consequently, a lecture is used to describe the performance analysis cycle shown in Figure 2 and introduce the main tools currently available for supporting each of these steps.
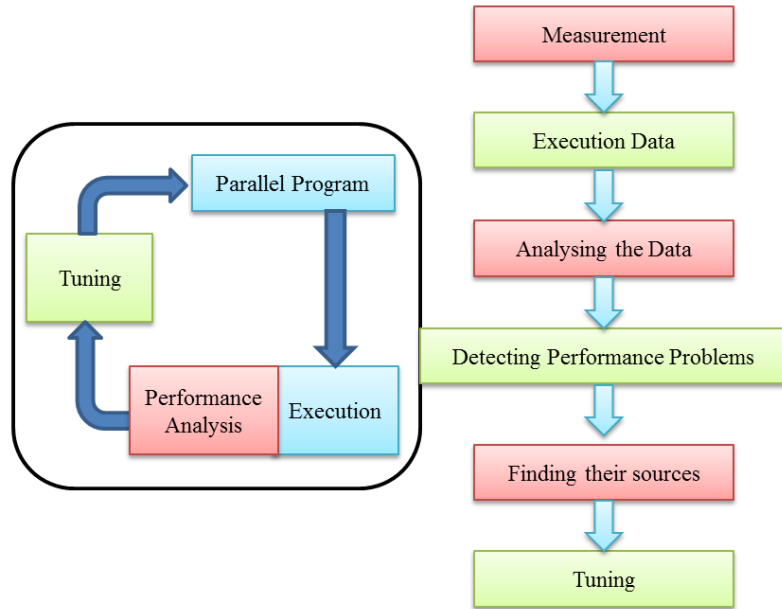

Fig. 2. Performance analysis in the application development cycle.

For example, Performance API (PAPI) [9] and Dyninst [2] are mentioned as supporting tools for getting execution measurements; Tuning and Analysis Utilities (TAU) [20], Scalasca [21] and Paraver [7] are presented as analysis and visualization tools; and Periscope Tuning Framework (PTF) [18], MATE [19] and Elastic [16] are introduced as automatic analysis and tuning tools.

It is worth mentioning that the Computer Architecture and Operating Systems department of Universitat Autònoma of Barcelona have received support

from computation industry leaders for the design and development of computation labs. We have been appointed by Intel as an Academic Partner with the use of Intel Parallel Studio as one of the programming environments for the practical laboratories and we have also been awarded as a GPU Teaching Center by Nvidia Corporation for introducing CUDA and GPU technology into the computer architecture studies. Our materials and systems are well updated to the latest versions released by Nvidia.

4   Applied Modelling and Simulation

The main goal of the Applied Modelling and Simulation subject is to introduce to the students real applications that use modelling and simulation and apply parallel programming. It is very significant to show the need to use high performance computing to make these real applications operational.

So, the proof of concept for this subject are developed by two different parts:

 − simulation model development and its performance analysis,
 − analysis of cases of use in collaboration with industry and research laboratories that use modeling and simulation activities every day.

In the first case study, a model of emergency evacuation using Agent Based Modeling is presented to the students [15]. In this model there are different aspects of analysis: the environment and the information (doors and exit signals), policies and procedures for evacuation, and social characteristics of individuals that affect the response during the evacuation. Moreover, the model includes the following hypothesis as a starting point to define it:

1. In emergency evacuation situations, people are generally nervous or even in panic, so they tend to perform irrational actions.
2. Individuals try to move as quickly as possible (more than normal).
3. Individuals try to achieve its objective and may try to push each other in their attempt to exit through a specific door causing physical injury to other individuals.

Students receive a partial model that includes management of the evacuation of an enclosed area that present a certain building structure (walls, access, etc.), obstacles, with a particular signaling and the corresponding safe zones and exits. The model also includes individuals who should be evacuated to safe areas. The model has been developed to support different parameters such as: individuals with different ages, total number of people in the area, number of exits, number of chained signals and safe areas, speed of each individual, probability of exchanging information with other individuals. The model is implemented in NetLogo [5] and the figure 3 is a representation of its main characteristics.

The first practical work for the students is to use a single-core architecture to make a performance analysis of the model and then modify it to incorporate a new, not covered, policy: overcrowding in exit zones [14]. With this new model the students must complete a new performance analysis.
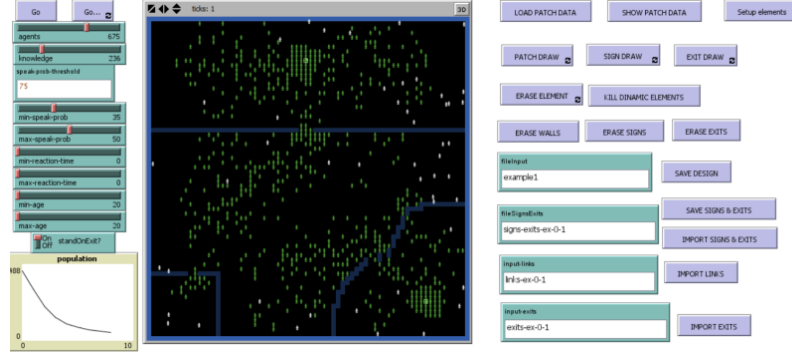
Fig. 3. Agent Based Modeling for emergency evacuations.

Considering the variability of each individual in the model a stability analysis is required. For this the Chebyshov Theorem (also spelled as Tchebysheff) will be used with confidence interval of 95% and $\alpha = 0.05, m = 6$. The result for this analysis indicates that 720 simulations must be made at least to obtain statistically reliable data. Taking into account the 720 executions on one core processor, the simulation time (average) is 7.34 hours for 1000 individuals and 27.44 hours for 1,500 individuals per scenario. In order to use this tool as DSS, the students are instructed of necessary HPC techniques and the embarrassingly parallel computing model is presented to reduce the execution time and the decision-making process time [13]. To perform this task, students must learn how to execute multiple parametric Netlogo model runs in a multi-core system and how to make a performance analysis to evaluate the efficiency and scalability of the proposed method.

The second case of analysis presented to the students is the paradigmatic example of meteorological services. Everybody watches the weather forecast on the TV news everyday and can imagine the complexity of the models involved, with huge meshes of points with hundreds of variables estimated for every point, and the computing requirements needed to provide a real prediction that must run much faster than real time. However, in this particular case, it is known that weather prediction models show chaotic behaviour. The way to keep this chaotic behaviour as limited as possible is to execute, not just a single simulation, but a complete set of scenarios (called ensemble) and apply statistical methods to conform the final prediction. This meteorological modelling and prediction part is presented by members of the Servei Meteorològic de Catalunya (Meteorological Service of Catalonia). Obviously, it is out of the scope of the subject to develop a meteorological model, but the students can use some small specific models such as wind field models (WindNinja [12]) for analysing its execution time, scalability and speedup. In this context, some students (one or two per year) may enroll an internship in this meteorological service developing code for some

particular model or applying parallel programming techniques to some of the existing models.

In a similar way, a collaboration has been established with the IC3-BSC (Institut Català de Ciències del Clima- Barcelona Supercomputing Center), but in this case the models and predictions are related to climatological models involving very large time scales. In this case, the real time aspect is not so critical, since the predictions are considered for decades or even centuries. However, the main point is to run hundreds or thousands of simulations with different parameters that make the total amount of computational requirements extremely large. Also in this case, some students carry out a internship in this centre, where they have access to very large computing resources and can make very interesting studies on speedup and scalability.

5   Conclusions

Many fields of science and engineering are evolving by the contribution of complementary fields. This implies that working teams in companies and research centres have significant interdisciplinary components and it is necessary that people from different fields are able to establish a common ground and understand the requirements and problems of all the sides. High performance computing, including parallel and distributed programming, becomes a central factor that is applied to many fields from science and engineering. So, it is necessary that the students from different fields receive a significant training in high performance computing. In this way, they are able to design and develop their own applications and, even more important, they understand the implications of using some particular architecture or programming paradigm. In this way they can establish a common language with computer scientist and work together in the development of more powerful and successful applications. In this interdisciplinary context, the experience of teaching parallel programming in interdisciplinary studies at master level has been presented. The main conclusion is that the experience is very successful and most students enjoy developing parallel programs, analysing their behaviour and trying to improve their performance. After this experience it would be very interesting to introduce similar subjects at the undergraduate level, so that students from different fields are able to apply high performance computing techniques to their computational problems from the very beginning.

References

1. Cuda Visual Profiler. `http://docs.nvidia.com/cuda/profiler-users-guide/index.html#visual-profiler`. [Online; accessed 18-May-2015].
2. Dyninst API. `http://www.dyninst.org/`. [Online; accessed 18-May-2015].
3. Lightweight performance tools. `https://code.google.com/p/likwid/`. [Online; accessed 18-May-2015].
4. Message Passing Interface Forum. `http://www.mpi-forum.org/`. [Online; accessed 18-May-2015].

5. NetLogo. Wilensky, U. (1999). Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. `https://ccl.northwestern.edu/netlogo/index.shtml`. [Online; accessed 18-May-2015].

6. OpenMP. `http://openmp.org/`. [Online; accessed 18-May-2015].

7. Paraver. `http://www.bsc.es/computer-sciences/performance-tools/paraver`. [Online; accessed 18-May-2015].

8. perf: Linux profiling. `https://perf.wiki.kernel.org/index.php/Main_Page`. [Online; accessed 18-May-2015].

9. Performance API. `http://icl.cs.utk.edu/papi/`. [Online; accessed 18-May-2015].

10. Performance Visualization. `http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/`. [Online; accessed 18-May-2015].

11. Nathan Bell and Jared Hoberock. Thrust: a productivity-oriented library for cuda. *GPU Computing Gems: Jade Edition*, 2012.

12. J.M. Forthofer, K. Shannon, and B. W. Butler. Initialization of high resolution surface wind simulations using nws gridded data. In *Proceedings of 3rd Fire Behavior and Fuels Conference; 25-29 October*, 2010.

13. Ian T. Foster. *Designing and building parallel programs - concepts and tools for parallel software engineering*. Addison-Wesley, 1995.

14. Albert Gutierrez-Milla, Francisco Borges, Remo Suppi, and Emilio Luque. Individual-oriented model crowd evacuations distributed simulation. In *Proceedings of the International Conference on Computational Science, ICCS 2014, Cairns, Queensland, Australia, 10-12 June, 2014*, pages 1600–1609, 2014.

15. Dirk Helbing, Lubos Buzna, Anders Johansson, and Torsten Werner. Self-organized pedestrian crowd dynamics: Experiments, simulations, and design solutions. *Transportation Science*, 39(1):1–24, 2005.

16. Andrea Martínez, Anna Sikora, Eduardo César, and Joan Sorribes. ELASTIC: A large scale dynamic tuning environment. *Scientific Programming*, 22(4):261–271, 2014.

17. Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

18. Renato Miceli, Gilles Civario, Anna Sikora, Eduardo César, Michael Gerndt, Houssam Haitof, Carmen B. Navarrete, Siegfried Benkner, Martin Sandrieser, Laurent Morin, and François Bodin. AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications. In *Applied Parallel and Scientific Computing - 11th International Conference, PARA 2012, Helsinki, Finland, June 10-13, 2012, Revised Selected Papers*, pages 328–342, 2012.

19. Anna Morajko, Oleg Morajko, Tomàs Margalef, and Emilio Luque. MATE: dynamic performance tuning environment. In *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, Pisa, Italy, August 31-September 3, 2004, Proceedings*, pages 98–106, 2004.

20. Sameer Shende and Allen D. Malony. The Tau Parallel Performance System. *IJHPCA*, 20(2):287–311, 2006.

21. Felix Wolf. Scalasca. In *Encyclopedia of Parallel Computing*, pages 1775–1785. 2011.