Performance and Usability Evaluation of a Pattern-Oriented Parallel Programming Interface for Multi-Core Architectures

Dalvan Griebler, Daniel Adornes, Luiz Gustavo Fernandes
Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS),
School of Informatics (FACIN), Graduate Program in Computer Science (PPGCC),
Parallel Application Modeling Group (GMAP).
Av. Ipiranga, 6681 - Building 32 - Porto Alegre - Brazil
dalvan.griebler@acad.pucrs.br, daniel.adornes@acad.pucrs.br, luiz.fernandes@pucrs.br

***Abstract*—Multi-core architectures have increased the power of parallelism by coupling many cores in a single chip. This becomes even more complex for developers to exploit the available parallelism in order to provide high performance scalable programs. To address these challenges, we propose the DSL-POPP (Domain-Specific Language for Pattern-Oriented Parallel Programming), which links the pattern-based approach in the programming interface as an alternative to reduce the effort of parallel software development, and achieve good performance in some applications. In this paper, the objective is to evaluate the usability and performance of the master/slave pattern and compare it to the Pthreads library. Moreover, experiments have shown that the master/slave interface of the DSL-POPP reduces up to 50% of the programming effort, without significantly affecting the performance.**

Keywords: Parallel Programming, Pattern-Oriented, Performance Evaluation, Usability Evaluation.

I. Introduction

Parallel programming has become inserted in the daily activities of software developers due to easy access to the multi-core architectures. However, its higher computational power achieved by coupling multiple processing elements on a single chip, increases complexity in parallelism exploitation. Thus requiring higher programming efforts to develop programs with good scalability and high performance.

Initially, the main interest of software designers was to provide flexible libraries. In recent years, the concern has also been to create high-level abstractions including flexibility issues without compromising the performance of the application. Therefore, the challenge is to evaluate the programming interface's usability. This is not a simple task because it requires volunteers to perform the experiments, and becomes laborious for everyone involved [16].

Such challenges have already been targeted in many studies that propose some level of abstraction [17]. On the other hand, [14] proposed an environment to automatically evaluate the usability of parallel language constructions, which does not consider the programming effort. In this paper, we present the master/slave pattern-oriented interface of DSL-POPP for multi-core architectures in order to evaluate its performance and usability. The contributions are the following:

- We present the master/slave pattern interface of the DSL-POPP;
- We perform a usability experiment comparing the programming effort between DSL-POPP and Pthreads;
- We conduct a performance experiment using four different applications.

This paper is organized as follows: Section II discusses related work; Section III presents the DSL-POPP in a nutshell; Section IV demonstrates the usability experiment; Section V describes the performance experiment; and finally, Section VI presents the conclusions.

II. Related Work

We consider Delite the first project to face the challenge of using DSLs as a programming environment for simplifying parallel programming. It generates parallel embedded DSLs for an application, whereas programmers implement domain-specific operations by extending the DSL framework [4]. The final result is a high performance DSL for a wider diversity of parallel architectures. In its interface implementation, Delite uses the Scala language. However, one disadvantage is that the lack of some features in the host language may restrict performance optimizations.

The concept of the algorithm skeleton was introduced by Murray Cole [6], who later proposed a skeleton-based programming environment called eSkel. Many libraries and frameworks emerged [8], allowing skeleton constructions in message passing and thread model scenarios. Other related approaches have emerged like the CO2P3S, which is a pattern-based parallel programming framework that creates pre-defined templates through a high-level interface [3], and the TBB (Thread Building Blocks) library that allows skeleton implementation in low-level operations [10]. Recently, the FastFlow template library [1] introduced a precise and optimized implementation of patterns for streaming applications and has proven to be faster than TBB, due to its advanced

parallelism implementation through techniques such as lock-free synchronization [2].

Yet, there are also programming models that are not pattern-based such as OpenMP [5], which have a set of pragma primitives designated to exploit loop parallelism in sequential programs. Charm++ is a parallel language based on the C++ syntax, parallelizing the execution through parallel objects and channel mechanisms [12]. Lastly, Cilk++ is a C language extension that provides keywords for spawn and synchronize threads, and loop parallelism [13]. The drawback of these specialized systems is that pattern-based implementations may require substantial work.

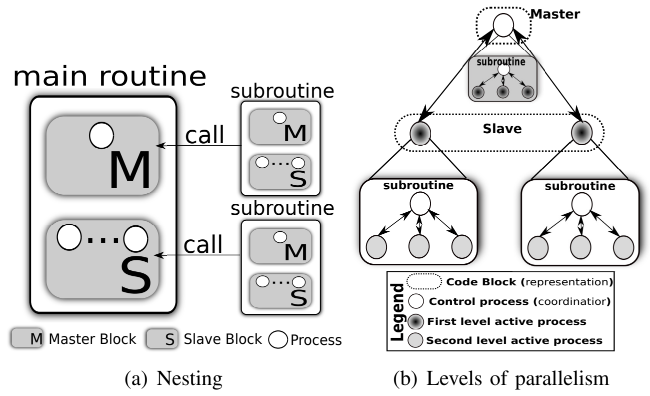
This research differs from these previous related works in it focuses on usability through a pattern-oriented interface, although it continues to focus on high performance and scalable programming. Also, this paper will evaluate the performance of parallel applications in multi-core architectures using statistical method, which differs of studies for performance prediction of parallel patterns [15].

III. DSL-POPP in a Nutshell

The POPP model is a standard way to design algorithms capable of combining and nesting parallel patterns while remaining compliant with different high performance architectures. It is framed on top of pattern-based approaches, where these patterns are presented as routine, and the skeleton templates are code blocks. The nesting of patterns occurs when a pattern subroutine is called inside a code block, producing a hierarchical composition. Additionally, it is possible to have different pattern combinations, resulting in different variations from the source application and implying different parallelism behaviors [9]. However, in this paper we focus on the master/slave, demonstrating how it can be nested in order to achieve levels of parallelism. Therefore, we exemplify in Figure 1(a) its behavior using process illustrations to represent abstract parallelism.

The master is responsible for sending the computational tasks to all slaves. Then, once all tasks have been computed, results are sent back to the master to finalize the whole computation. In this pattern, both the POPP routine and the subroutines can implement their own master and slaves code blocks. For instance, a slave block may have as many slave processes as necessary running parallel, and nesting occurs when a slave or master code block calls a Master/Slave subroutine.

We named active processes those that are used to measure the performance, and control processes those that are only used to represent the skeleton behavior pattern. Through the processes labeled second level, we see the point from which levels of parallelism are achieved. In general, the control process coordinates the computation flow and does not perform significant computations (being CPU idle) while active processes are computing their tasks. For this reason, those processes are not used to measure the speed-up. The levels of parallelism can be achieved by calling subroutines from inside the slave block. Even though the execution sequence



(a) Nesting (b) Levels of parallelism

Fig. 1. How to address nested patterns and levels of parallelism through the POPP model for the master/slave pattern.

of the routines is not clearly defined in the example, the idea is that each one executes after the other has finished. In other words, using the example in Figure 1(b), slave block processes do not run parallel with the subroutine called on the master block, but they are on the same parallelism level.

The POPP model is built over the C language and standardized as follows: the specification of routines begins with “$” and code blocks with “@”. The pattern routine should be declared in a function followed by the return data type and its name. Code blocks should be used inside of the pattern routine, consisting of pure C code, and the skeleton communication occurs through predefined arguments. Also, the code declared inside each block is private and will not be accessible to other code blocks. In the current master/slave interface, in the master block only the buffer and its size have to be given, whereas in the slave block it is necessary to specify the number of threads, buffer, buffer size, and the load balancing policy (Listing 1).

```
1 $MasterSlavePattern int func_name(){
2   @Master(void *buffer, const int size){
3     //full C code
4     @Slave(const int num_threads, void *
         buffer,const int size, const policy){
5       //full C code
6     }
7     //full C code
8   }
9 }
```

Listing 1. Master/Slave programming interface.

In the runtime system, the master block creates as many threads as defined in the slave block and waits until all slave threads have finished their work. This is hidden from programmers, since thread creation occurs when the slave block starts and the synchronization is performed automatically at the end of the slave block. Moreover, at the end of a slave block, all slave threads send their work back to the master (using the buffer parametrized to the slave block) in order to allow it to merge all results. This communication procedure, and the load balancing are also hidden from developers.

In fact, slave threads receive their workloads according to the policy argument in the slave block. The implementation of

these policies is automatically generated by the pre-compiler system. Currently, we have only implemented an optimized version of the static load balancing (POPP_STATIC), where the workload is uniformly divided by the number of threads, and the resulting chunks are then statically assigned to slave threads as they start their computations.

To use the programming interface, developers have to include the DSL-POPP library (*poppLinux.h*) in the source code and compile the program with the DSL-POPP compiler (*popp*). This library includes all routine definitions and code blocks. The compilation process of the source code is depicted in Figure 2. The source-to-source code transformation between DSL-POPP interface and C code is automatically done by the pre-compiler, which is also responsible for checking syntax and semantic errors. Then, the pre-compiler generates the C parallel code using the Pthreads library based on the parallel pattern used. Finally, we use the GNU C compiler to generate binary code for the target platform.

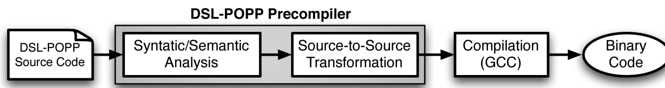


Fig. 2. Compiler overview.

A. Matrix Multiplication Example

DSL-POPP can be used to design different parallel algorithms [9]. However, we chose the classic matrix multiplication algorithm to illustrate how programmers have to parallelize it to perform the usability experiment. This application consists of some auxiliary functions to load and print the matrix as well as measure the execution time. We intend to present in details only the main parts of the application, since the focus relies on Matrix Multiplication (MM), which has to be implemented with Pthreads (Listing 2) and DSL-POPP (Listing 3).

```
1#include <stdio.h>
2#include <pthread.h>
3#define MX 1000   // matrix size
4long int num_threads=2;
5long int matrix1[MX][MX], matrix2[MX][MX],
     matrix[MX][MX];
6double timer(){/* return the time in seconds */
     }
7void attr_val(long int **matrix, long int **
     matrix1, long int **matrix2{/* values
     attribution */}
8void printMatrix(long int **matrix){/* prints
     a matrix */}
9void *Thread(void *th_id){
10   long int id= (long int*) th_id;
11   long int i, j, k, end;
12   end=(id*(MX/num_threads))+(MX/num_threads);
13   if(id==num_threads-1)
14       end=MX;
15   for(i=id*(MX/num_threads); i<end; i++)
16     for(j=0; j<MX; j++)
17       for(k=0; k<MX; k++)
18           matrix[i][j] += (matrix1[i][k] *
                 matrix2[k][j]);
19   pthread_exit(NULL);
20}
21int main(){
22   double t_start, t_end;
23   pthread_t th[num_threads];
24   void *status;
25   t_start = timer();
26   attr_val(matrix,matrix1,matrix2);
27   int i;
28   for(i=0;i<num_threads; i++)
29       pthread_create(&th[i], NULL, &Thread, (
             void *) i);
30   for(i=0; i<num_threads; i++)
31       pthread_join(&th[i], &status);
32   t_end = timer();
33   printf("EXECUTION TIME: %lf seconds\n",
         t_end-t_start);
34   printMatrix(matrix);
35}
```

Listing 2. MM implemented with Pthreads .

As can be observed, we used a very simple strategy to parallelize the MM with Pthreads. The *main* function is responsible for creating auxiliary threads (lines 28 - 31), which will perform the MM in parallel (lines 15 - 18). Therefore, each thread performs the MM on a sub-matrix. The sub-matrix is obtained by dividing the number of rows of the resulting matrix by the number of threads. We used the thread ID to select a different sub-matrix for each thread, avoiding the use of synchronization mechanisms (lines 12 - 14).

```
1#include <stdio.h>
2#include "poppLinux.h"
3#define MX 1000   // matrix size
4long int num_threads=2;
5long int matrix1[MX][MX], matrix2[MX][MX],
     matrix[MX][MX];
6double timer(){/* return the time in seconds */
     }
7void attr_val(long int **matrix, long int **
     matrix1, long int **matrix2{/* values
     attribution */}
8void printMatrix(long int **matrix){/* prints
     a matrix */}
9$MasterSlavePattern int main(){
10  @Master(matrix, MX){
11    double t_start, t_end;
12    t_start = timer();
13    attr_val(matrix,matrix1,matrix2);
14    @Slave(num_threads, matrix, MX,
        POPP_STATIC){
15      long int i, j, k;
16      for(i=0; i<MX; i++)
17        for(j=0; j<MX; j++)
18          for(k=0; k<MX; k++)
19            matrix[i][j] += (matrix1[i][k]*
                matrix2[k][j]);
20    }
21    t_end = timer();
22    printf("EXECUTION TIME: %lf seconds\n",
          t_end-t_start);
23    printMatrix(matrix);
24  }
25}
```

Listing 3. MM implemented with DSL-POPP .

The corresponding DSL-POPP version of the MM is presented in Listing 3. The main routine is composed of a master block (line 9) and a slave block (line 14). The master block does not contribute to the whole parallel computation, since it starts the computation, joins the results of each slave threads, and creates the slaves and waits for them to finish so that it is automatically performed by the pre-compiler. The slave block contains the sequential code of the MM which will be split among independent slave workers (threads).

It is possible to notice that DSL-POPP considerably reduces the amount of code necessary to parallelize the application. Additionally, developers do not have to worry about how to split the work. To evaluate how this will impact the programming effort, we performed a usability experiment using this application in the next section.

IV. Usability Evaluation

In this experiment, we only considered the time spent (in minutes) to implement the parallel solution, using it as the metric to evaluate the programming effort when using DSLPOPP compared to Pthreads. In order to do so, we invited M.Sc. and Ph.D. students in Computer Science and asked them to parallelize the matrix multiplication with these approaches. Then, we performed an analysis using the hypothesis statistical test, making the assumption of a null hypothesis (H_0) that will be rejected or not. We used 95% reliability and the significance level (max probability to reject the H_0) adopted was 5%. This means that to reject the H_0 result has to be less than 0.05 (this value is called p-value in the literature) [7]. The hypotheses are formalized as follows:

1) **Null hypothesis** (H_0): the effort in parallel programming with Pthreads is equal to the effort using DSL-POPP (Pthreads = DSL-POPP).
2) **Alternative hypothesis** (H_1): the effort in parallel programming with Pthreads is greater than DSL-POPP approach (Pthreads > DSL-POPP).

We invited students to answer a questionnaire to evaluate their previous knowledge (the options were: none, low, medium, and high). Based on this questionnaire, we removed the students that did not have the required skills (knowledge in C language, parallel programming and Linux platform). Thus, we split the remaining 20 participants into two groups with equal knowledge. Group 1 parallelized the problem with the DSL-POPP first and then with Pthreads. Group 2 parallelized the same problem but in the reverse order.

In order to avoid external influences, the experiment was carried out in a controlled environment (laboratory), where participants only had access to a user manual (previously reviewed by another researcher) and the original sequential code. Each participant used a desktop machine with Ubuntu Linux installed and no Internet access was allowed. All students had to achieve correct parallel implementations with performance above a minimum threshold (of at least 90% of the linear speedup with two threads) for the experiment to be considered completed.

Table I presents the results obtained from the effort evaluation. For each participant, we present his/her previous experience with Pthreads (taken from the questionnaire) and the time spent to implement the problem using the DSL-POPP and Pthreads. Therefore, it is important to highlight that all participants had never developed applications with the DSL-POPP and most of them had already developed parallel applications with Pthreads. The results demonstrate that the DSL-POPP demands less work from the developers. An exception was found in Group 1 (ID=1), which parallelized the problem faster with Pthreads than with DSL-POPP. This is due to two factors: (i) the participant had already implemented a matrix multiplication with Pthreads and (ii) he had significant experience in developing applications with Pthreads.

We used the Statistical Package for the Social Sciences (SPSS) to analyze the obtained results. The analysis showed a significantly higher level of effort to program with Pthreads than with DSL-POPP. The average time spent to implement the parallel version of the matrix multiplication was 51.45 minutes with Pthreads (the 95% confidence interval was 42.98 to 59.92 minutes) whereas it was 20.70 minutes for the average with DSL-POPP (the 95% confidence interval was 17.59 to 23.81).

For the hypothesis test, there are basically two statistical tests that can be applied: parametric and nonparametric. The parametric test normally requires distributed data and homogeneity of variance. However, the time spent to implement the solution with Pthreads did not follow a normal distribution, thus indicating a non-parametric test. Due to that, we used the Wilcoxon approach for paired sample tests [7]. This hypothesis test is based on the differences between the scores of the two approaches, ranking them positively and negatively. The results are shown in Table II.

TABLE II
Statistical hypothesis test.

Ranks	N	Mean Rank	Sum of Ranks
Negative Ranks	1 *(a)*	4.0	4.0
Positive Ranks	19 *(b)*	10.8	206.0
Ties	0 *(c)*	–	–
Total	**20**		

(a) Pthreads ¡ DSL_POPP
(b) Pthreads ¿ DSL_POPP
(c) Pthreads = DSL_POPP

Wilcoxon test	Pthreads vs. DSL-POPP
Z	-3.771*
Asymp. Sig. (2-tailed)	0.0

**Based on negative ranks.*

We noticed that only one negative rank was found in the statistical test. This means that Pthreads require more effort than DSL-POPP (**(b)** Pthreads > DSL_POPP). In order to confirm whether the effort is significantly different between these approaches, we used the significance level (Sig.), which must be less than 0.05 [7]. The SPSS returned the result shown at the bottom (Wilcoxon test) of Table II, concluding that the effort is significantly different. Thus, we can reject the null hypothesis (H_0) and based on the mean results, we can accept the alternative hypothesis H_1, which states that the effort in

TABLE I
EFFORT EVALUATION RESULTS.

Group 1: DSL-POPP → Pthreads				Group 2: Pthreads → DSL-POPP			
ID	Experience with Pthreads	DSL-POPP Time (min)	Pthreads Time (min)	ID	Experience with Pthreads	DSL-POPP Time (min)	Pthreads Time (min)
1	Medium	29	18	11	High	15	33
2	Medium	23	32	12	Medium	17	54
3	Low	13	30	13	Medium	12	65
4	Medium	25	29	14	Low	15	70
5	Medium	19	27	15	Low	17	60
6	Low	33	65	16	Low	15	71
7	Low	15	39	17	Low	12	61
8	Low	31	81	18	Zero	21	63
9	Zero	28	59	19	Zero	24	61
10	Low	28	45	20	Low	22	66

parallel programming with Pthreads is greater than with DSL-POPP.

V. PERFORMANCE EXPERIMENTS

This experiment seek to evaluate the performance in order to identify whether there are significant performance differences between DSL-POPP and Pthreads. The metric associated with this experiment is the execution time, which is used to calculate the speed-up and compare the performance results. The hypotheses for this experiment are the follows:

1) **Null hypothesis** (H_0): the performance of the algorithms implemented with Pthreads is equal to the implemented using DSL-POPP (Pthreads = DSL-POPP).
2) **Alternative hypothesis** (H_1): the performance of the algorithms implemented with Pthreads is significantly different than implemented with DSL-POPP (Pthreads ! = DSL-POPP).

The performance tests were carried out on a machine running Ubuntu-Linux-12.04-server-64bits and the architecture was composed of Intel Xeon X3470 (2.93GHz), 8GB of main memory, and 2TB of disk. We chose four well-known algorithms from the literature to be parallelized by one volunteer from the usability experiment. This programmer implemented these algorithms with Pthreads and DSL-POPP, and we certified that the output result was the same as the sequential version in order to guarantee the parallelization correctness. Also, for each sample we performed 40 random executions to measure/compare the performance and efficiency.

Therefore, the first program Estimates an Integral (EI) over a domain of two dimensions using an averaging technique. The second is a Molecular Dynamic (MD) simulation algorithm. The other application sets up a dense Matrix Multiplication (MM), and the last application counts the Prime Numbers (PN) between 1 to N. We used the SLOCCount tool [18] to present an overview of the physical source lines of code and the development effort for this application. The results are presented in Table III. As can be seen, the estimate of this tool demonstrates that the programmer used more lines of code and the development effort was probably more expensive than with DSL-POPP.

TABLE III
CODE ANALYSIS.

	Source Lines of Code (SLOC)			*Development Effort Estimate (Min.)*		
App.	**Ori.**	**DSL-POPP**	**Pthreads**	**Ori.**	**DSL-POPP**	**Pthreads**
EI	91	93	135	8322	8760	12702
MD	249	259	352	24528	25404	35040
MM	99	105	209	9198	10074	20148
PN	78	100	142	7008	9198	13578

It is important to highlight that this tool is not equivalent to the usability experiment because it does not effectively evaluate human interaction. Additionally, it does not consider some parallel programming issues, for example, the programmers have to find the pieces of code that can/must be parallelized, study how to split the computation, how to communicate, and how to perform synchronization. Due to this, it only gives an overview of the development effort to show some trends. For example, applying this test over the matrix multiplication application used in the usability experiment, it estimates a development effort for the sequential version of (52 SLOC) 80.18 hours, for DSl-POPP (59 SLOC) 87.36 hours, and with Pthreads (73 SLOC) 109.30 hours.

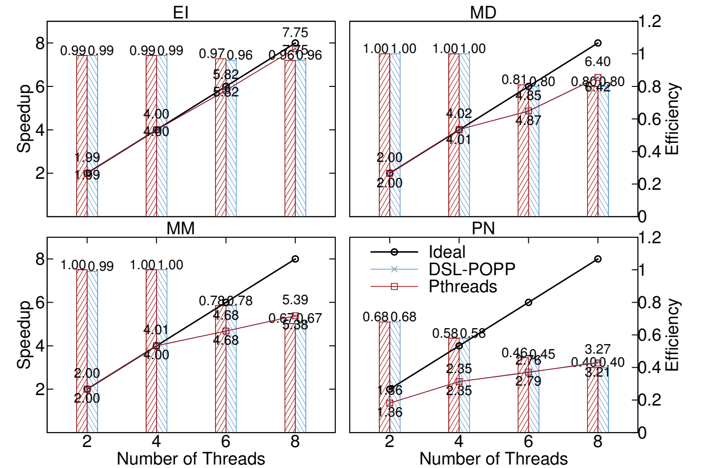


Fig. 3. Speed-up and Efficiency.

The performance and efficiency of these applications are presented in the graphs of the Figure 3. The EI, MD, and

MM achieved ideal speed-ups until 4 threads. However, with 6 and 8 threads they have performance losses due to the use of logical cores. As expected, the PN does not achieve good performance over the other applications, because the algorithm is not embarrassingly parallel. Finally, both speed-up and efficiency presents similar results between DSL-POPP and Pthreads implementations.

Even though it seems there are no significant performance differences according to the graphs of speed-up and efficiency, we performed a significance (Sig) test using the SPSS tool. It was performed with 95% reliability (the probability to reject the H_0 is 5%) over the 40 execution time of each sample so that the Sig. must be < 0.05 for reject the H_0. The place where the statistics test indicates significant performance differences are in bold (Table IV), thus in these particular cases H_0 is rejected. Overall, all performance tests of MM application presented results without significant differences admitting 95% reliability, which is considered one of the most accurate statistic evaluations for software usability [11].

TABLE IV
SPSS OUTPUT FOR THE TEST OF SIGNIFICANCE (SIG.)

Threads	EI	MD	MM	PN
2	0.455	0.135	0.059	0.281
4	0.740	0.090	0.174	**0.001**
6	**0.000**	**0.000**	0.837	0.199
8	**0.000**	0.269	0.381	**0.011**

VI. Conclusions

This paper presented a performance and usability evaluation of a pattern-oriented interface for multi-core architectures. It was performed using software experiments and statistical analysis, whose results demonstrated that the master/slave pattern interface of the DSL-POPP requires less programming effort than Pthreads in the parallelization of a matrix multiplication algorithm. Additionally, we demonstrated that the performance is not significantly affected in four applications, and we discuss the development effort estimated by the SLOCCount tool in order to demonstrate some programming effort trends in these applications.

The comparative analysis with Pthreads library was performed because it is used to implement parallel code generation. Hence, we could see the efficience of the DSL-POPP for usability and performance issues. However in the future, we intend to evaluate the impact of the parallel programming efforts in applications that can exploit the nested pattern ability, and repeat these experiments for other pattern interfaces available in the DSL-POPP. Also, we plan to compare the performance and usability with other programming interfaces that are not pattern-oriented, such as OpenMP and Cilk.

Acknowledgments

Authors wish to thank the research support of FAPERGS (Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul) and CAPES (Coordenação de Aperfeiçoamento Pessoal de Nível Superior). Additionally, authors would like to thank the financial support of FACIN (Faculdade de Informática) and PPGCC (Programa de Pós-Graduação em Ciência da Computação).

References

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. Accelerating Code on Multi-cores with FastFlow. In *Euro-Par 2011 Parallel Processing*, volume 6853, pages 170–181, Berlin, Heidelberg, August 2011. Springer-Verlag.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. An Efficient Unbounded Lock-Free Queue for Multi-core Systems. In *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, volume 7484, pages 662–673, Rhodes Island, Greece, August 2012. Springer.

[3] S. Bromling, S. MacDonald, J. Anvik, J. Schaeffer, D. Szafron, and K. Tan. Pattern-Based Parallel Programming. In *Parallel Processing, 2002. Proceedings. International Conference on*, pages 257–265, British Columbia, Canada, 2002. IEEE Computer Society.

[4] H. Chafi, A. Sujeeth, K. Brown, H. Lee, A. Atreya, and K. Olukotun. A Domain-specific Approach to Heterogeneous Parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, volume 1, pages 35–46, New York, NY, USA, February 2011. ACM.

[5] B. Chapman, G. Jost, and R. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. Massachusetts Institute of Technology, London, England, 2008.

[6] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, USA, 1989.

[7] A. Field. *Discovering Statistics Using SPSS*. SAGE, Dubai, EAU, 2009.

[8] H. González-Vélez and M. Leyton. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, 2010.

[9] D. Griebler and L. G. Fernandes. Towards a Domain-Specific Language for Patterns-Oriented Parallel Programming. In *Programming Languages - 17th Brazilian Symposium - SBLP*, volume 8129 of *Lecture Notes in Computer Science*, pages 105–119, Brasilia, Brazil, October 2013. Springer Berlin Heidelberg.

[10] Intel. Thread Building Block (Intel TBB), Extracted from <https://www.threadingbuildingblocks.org>, 2014.

[11] N. Juristo and A. M. Moreno. *Basics of Software Engineering Experimentation*. Springer, Boston, USA, 2001.

[12] L. V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 91–108, New York, NY, USA, October 1993. ACM.

[13] C. E. Leiserson. The Cilk++ Concurrency Platform. In *Proceedings of the 46th Annual Design Automation Conference*, volume 1, pages 522–527, New York, NY, USA, July 2009. ACM.

[14] V. Pankratius. Automated Usability Evaluation of Parallel Programming Constructs (NIER Track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 936–939, New York, NY, USA, May 2011. ACM.

[15] M. Raeder, D. Griebler, L. Baldo, and L. G. Fernandes. Performance Prediction of Parallel Applications with Parallel Patterns Using Stochastic Methods. In *Sistemas Computacionais (WSCAD-SSC), XII Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 1–13, Espírito Santo, Brasil, October 2011. IEEE Computer Society.

[16] C. Sadowski and A. Shewmaker. The Last Mile: Parallel Programming and Usability. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, volume 1, pages 309–314, New York, NY, USA, November 2010. ACM.

[17] D. Szafron and J. Schaeffer. An Experiment to Measure the Usability of Parallel Programming Systems. *Concurrency - Practice and Experience*, 8(2):147–166, 1996.

[18] D. A. Wheeler. SLOCCount, Access from <http://www.cic.unb.br/~facp/cursos/ps/slides/doc/SLOCCount.html>, 2014.