Third NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar-13), held in conjunction with the *27th IEEE International Parallel & Distributed Procession Symposium (IPDPS 2013)*, Boston, MA, May 20, 2013.

Toward using higher-level abstractions to teach Parallel Computing

Clayton Ferner
University of North Carolina Wilmington
601 S. College Rd.
Wilmington, NC 28403, USA
cferner@uncw.edu

Barry Wilkinson
University of North Carolina Charlotte
9201 University City Blvd.
Charlotte, NC 28223 USA
abw@uncc.edu

Barbara Heath
East Main Evaluation & Consulting, LLC
P.O. Box 12343
Wilmington, NC 28405 USA
bheath@emeconline.com

Abstract— **We have developed two new approaches to teaching parallel computing to undergraduates using higher level tools that lead to ease of programming, good software design, and scalable programs. The first approach uses a new software environment that creates a higher level of abstraction for parallel and distributed programming based upon a pattern programming approach. The second approach uses compiler directives to describe how a program should be parallelized. We have studied whether using the above tools better helps the students grasp the concepts of parallel computing across the two campuses of the University of North Carolina Wilmington and the University of North Carolina Charlotte using a televideo network. We also taught MPI and OpenMP in the traditional fashion with which we could ask the students to compare and contrast the approaches. An external evaluator conducted three surveys during the semester and analyzed the data. In this paper, we discuss the techniques we used, the assignments we gave the students, and the results of what we learned.**

***Keywords-** pattern programming; compiler directives; parallel computing; distributed computing.*

I. INTRODUCTION

General-purpose computers now have multiple cores. It is common to see processors with 4 cores; although, soon we can expect desktop computers to have 10s and even 100s of cores. It is no longer sufficient for computer science students to be trained solely in the programming of single processor systems. It is now imperative that all computer science students, graduate as well as undergraduate, develop the skills to successfully program for systems with multiple processors. Although the multiple cores on a single die use a shared-memory model, as the number of cores increase, it will become necessary to use distributed-memory and hierarchical-memory models. Furthermore, GPUs are becoming more common as an effective way to increase performance.

Parallel computing has typically been treated as an advanced topic in computer science. It must now be treated as a core topic. However, most parallel computing classes use low-level tools such as MPI for distributed-memory sytems, OpenMP for shared-memory systems and CUDA/OpenCL for high performance GPU computing. Such a course also uses fairly simple problems. Unfortunately, this approach does not give the student programmer the skills to tackle larger problems nor skills in computational thinking for parallel applications. In addition, programmers have to deal with issues such as deadlock and mutual exclusion. A programming approach is needed that raises the level of abstraction to make parallel programming easier and also more scalable.

To address the above problems, we have developed two new approaches to create a higher level of abstraction for parallel and distributed computing. The first approach uses a new software environment we have developed that creates a higher level of abstraction for parallel and distributed programming based upon a pattern programming approach. The second approach is uses compiler directives to describe how a program should be parallelized for a distributed-memory system.

We developed new educational materials based on these two approaches to teach parallel computing. In Fall 2012, we taught a course in parallel computing jointly at the University of North Carolina Wilmington and the University of North Carolina Charlotte broadcast to both campuses using the North Carolina Research and Education televideo network (NCREN) connecting universities across North Carolina. Our goal was to determine if using the above tools would better help the students grasp the concepts of parallel computing. We also taught MPI and OpenMP in the traditional fashion with which we could ask the students to compare and contrast the approaches. An external evaluator conducted three surveys during the semester and analyzed the data. In this paper, we discuss the techniques we used, the assignments we gave the students, and the results of what we learned.

Teaching effectiveness data was collected by co-author Heath completely independent of the two instructors Ferner and Wilkinson and was not released to the instructors until after the course had finished and graded. Proper Institutional Review Board (IRB) protocols were followed throughout including using consent forms and maintaining complete confidentiality of individual students. Student participation was completely voluntary. The class size was around 58 students with about a third participating in the surveys. The class was a mix of undergraduate and graduate students, all studying computer science, with approximately half being undergraduate students. The prerequisite for the course is two semesters of programming plus a course on data structures. The data collected and presented herein is from the first offering of this new approach, and the results will be used to improve our materials.

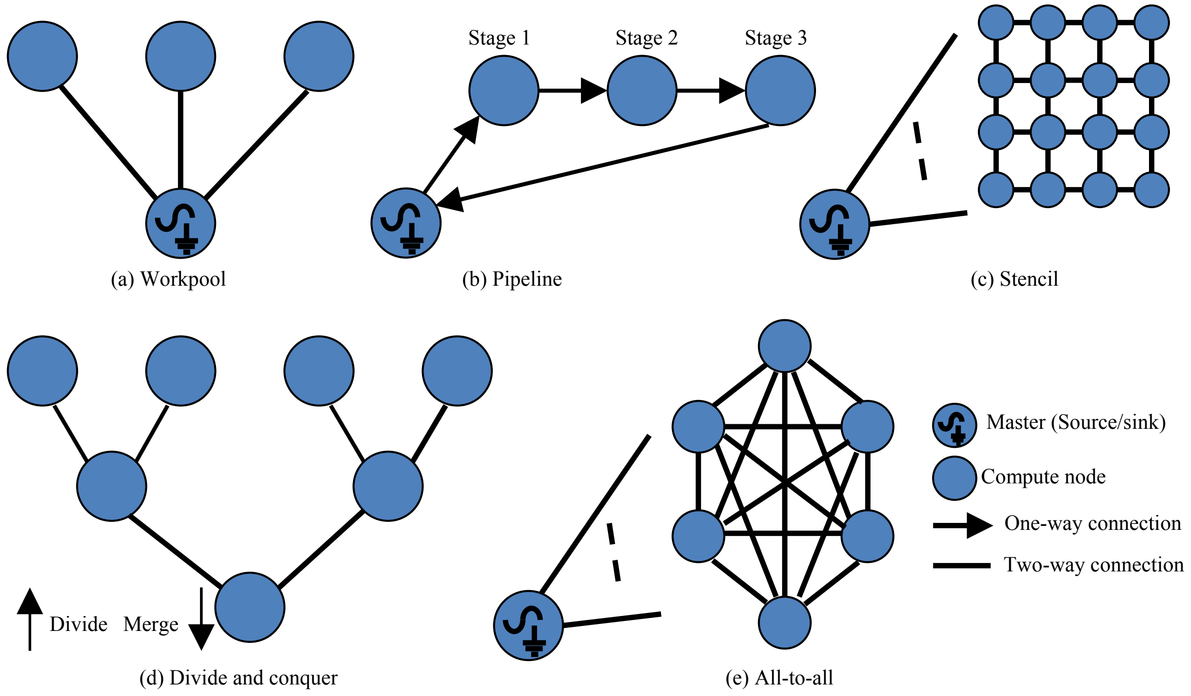

Figure 1: Parallel Patterns

The rest of this paper is organized as follows. Existing work is briefly reviewed in Section II. In Section III, we describe the two new approaches. In Section IV, we describe the survey instruments. In Section V, we present and discuss the results. Finally, Section VI concludes.

II. EXISTING WORK

The idea of design patterns in software engineering has been around for many years [5] and applied to undergraduate teaching [1]. The pattern approach to parallel programming has been explored in the influential textbook by Mattson et al. [7] written for software developers, and also in several research projects including at University of Illinois at Urbana-Champaign, University of California, Berkeley [6], and University of Torino/Università di Pisa Italy [4]. Industrial efforts in this direction include Intel [8] and Microsoft [11]. These projects do not focus on teaching parallel programming at the undergraduate level; rather they promote higher-level tools for programming style, scalability, and productivity. We extend such goals to undergraduate parallel programming teaching.

Renault and Parrot [9] have created a pre-processors that can automatically generated MPI derived datatypes from the C data types. The goal is to assist the programmer with creating the complex MPI datatypes needed for the transmission of user-defined datatypes. Their work, however, does not generate the code to parallelize an algorithm. The closest work to our work on compiler directives is the llCoMP compiler for the llc language [10]. The llc language allows the programmer to specify parallel constructs for both MPI and OpenMP using llc and OpenMP pragma statements. In their work, Reyes, Dorta, Almeida, and Sande discuss the use of "static" and "dynamic patterns". However, it appears that these patterns are compiler optimizations designed to improve the efficiency of the communication rather than user defined patterns related to the algorithm structure. Our compiler also has directives to specify parallel constructs to use both MPI and OpenMP. We plan to introduce new directives specifically to describe an algorithm through a pattern.

III. TWO APPROACHES TO HIGHER LEVEL ABSTRACTIONS

A. *Patterns*

In this approach, the programmer first identifies an appropriate parallel computational pattern or patterns to solve the problem rather than immediately starting with a low-level API such as MPI or OpenMP. We focus on higher level computational patterns such as workpool, pipeline, stencil, divide and conquer, and synchronous all-to-all as illustrated in Figure 1 rather than lower level constructs such as fork-join and loop. Our pattern programming framework, called Seeds, was developed as part of a PhD project exploring distributed computing by Jeremy Villalobos and is Java-based [13]. A C++ version is in development.

The framework will automatically distribute tasks across distributed computers and processors, once the programmer has selected the pattern, specified the data to be sent to and from the processors/processes, and specified the computation to be performed by the processors/processes. The framework has built-in patterns including workpool, pipeline, stencil, and synchronous all-to-all. Other patterns can be implemented by the programmer using more advanced knowledge of the framework. The framework will self-deploy on distributed computers, clusters, and multicore processors, or a combination of distributed- and shared-memory computers. The key aspect is the programmer does not program using low level message passing APIs such as MPI or OpenMP. Instead, the patterns are implemented automatically, relieving the programmer of concerns for message-passing deadlock. The programmer has a very simple programming interface avoiding the complexities of putting message-passing statements in the actual code. Figure 2 shows an example of the workpool pattern for the Monte Carlo estimation of π. The user must provide details of the diffuse, compute, and gather operations.

Our pattern programming framework is used for problems that traditionally would be coded in MPI for a message-passing computing platform or OpenMP for shared memory multicore platform. The first implementation used JXTA P2P networking and will run on both multicore or distributed platforms, or a combination (Windows, mac or Linux). A later thread-based version of the framework was implemented for running more efficiently just on multicore platforms. In either case, students use the Eclipse IDE. Just two classes are needed, one for the master and slave methods and one to deploy the framework and run the code. More details of the programmer's interface with tutorials can be found [14]. Its use in an educational undergraduate teaching setting was first described in [12].

In the course, we also teach HPC GPU programming with CUDA as in most complete parallel programming courses. However, we first start this section with introducing the data parallel pattern and then use CUDA as one (low-level) implementation of the pattern. Our strategy to work from computational strategies down to the implementation rather than start with the implementation that can change. For example, although CUDA is the prevalent programming environment for HPC GPU programming, higher level tools are becoming available, such as OpenACC (a compiler directive approach).

B. Compiler Directives

In teaching parallel computing in the past, we have found that students more easily learn to use effectively OpenMP on a shared-memory system than MPI on a distributed-memory system. OpenMP is a slightly higher level of abstraction than MPI, although they are not comparable. We wanted to create a similar abstraction as OpenMP but for distributed-memory systems. This was the catalyst for the creation of the Paraguin compiler [2][3].

The Paraguin compiler is a compiler we built at the University of North Carolina Wilmington using the SUIF compiler system created at Stanford University. The Paraguin compiler generates a parallel solution using MPI that is suitable for execution on a distributed-memory system based on programmer directives. Similar to OpenMP in which the programmer specifies parallelization techniques through the use of #pragma statements, the Paraguin compiler also takes direction from the programmer though the use of #pragma statements. The goal is to provide a similar level of abstraction as OpenMP but for distributed-memory systems. Figure 3 shows an example of using the Paraguin compiler directives for matrix multiplication.

```
public Data DiffuseData (int segment) {
    DataMap<String, Object> d=
                 new DataMap<String, Object>();
    d.put("seed", R.nextLong());
    // returns a random seed for each job unit
    return d;
}

public Data Compute (Data data) {
    DataMap<String, Object> input =
                  (DataMap<String,Object>)data;
    DataMap<String, Object> output = new
                      DataMap<String, Object>();
    // get random seed
    Long seed = (Long) input.get("seed");
    Random r = new Random();
    r.setSeed(seed);
    Long inside = 0L;
    for (int i = 0; i < DoubleDataSize ; i++) {
        double x = r.nextDouble();
        double y = r.nextDouble();
        double dist = x * x + y * y;
        if (dist <= 1.0)
            ++inside;
    }
    // to return to GatherData()
    output.put("inside", inside);
    return output;
}

public void GatherData (int segment, Data dat)
{
    DataMap<String,Object> out =
                  (DataMap<String,Object>) dat;
    Long inside = (Long) out.get("inside");
    // aggregate answer from all worker nodes.
    total += inside;
}
```

Figure 2: Example Workpool Pattern in the Seeds Framework for the Monte Carlo Estimation of π

One advantage of such an approach is that only compilers that recognize a particularly-named pragma will process it. All others will ignore it. For example, one can created a sequentially-executed program from the same source using gcc as the compiler. The pragma statements do not need to be removed or commented out. Furthermore, this makes it possible to create a hybrid program, which takes advantage of both shared-memory and distributed-memory, by using multiple types of pragmas. At present, the programmer can implement algorithms matching a particular pattern only through the use of templates. In the future, our compiler will have directives specifically to describe patterns, similar to the way described in Section III.A.

IV. SURVEY INSTRUMENTS

During the pilot offering of our new course in Fall 2012, students were invited to provide feedback that would assist with the development of the future course offerings. Feedback was collected by the external evaluator via three

```
#pragma paraguin begin_parallel
#pragma paraguin bcast a b
#pragma paraguin forall  C   p   i   j   k \
                        0x0  -1   1 0x0 0x0 \
                        0x0   1  -1 0x0 0x0

#pragma paraguin gather 1   C   i   j   k \
                           0x0 0x0 0x0   1 \
                           0x0 0x0 0x0  -1

for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
      for (k = 0; k < N; k++) {
         c[i][j] = c[i][j] + a[i][k] * b[k][j];
      }
   }
}
#pragma paraguin end_parallel
```

Figure 3: Example Paraguin Compiler Directives for Monte Carlo Estimation of π

surveys: a pre-, mid-, and post-course survey. Students who provided consent and completed each of the three surveys were entered in a drawing for one of eight $25 Amazon gift cards. For each survey, 58 invitations were sent to students at both campuses. The response rates for the three surveys were: 36%, 29%, and 28%, respectively.

The purpose of the pre- and post-semester surveys was to assess the degree to which the students learned the material taught during this offering. A set of seven pre-course items were developed for this purpose. The items were presented with a six-point Likert scale from "strongly disagree" (1) through "strongly agree" (6). Table I shows the questions that were on these surveys. There were additional questions on the post-semester survey related to students' feedback of the course, but the results of those questions are not discussed in this paper, because they are focused upon the delivery, helpfulness of the instructors, required learning outside the classroom, and suggestions for improvement. The information garnered from that data did not focus on the effectiveness of our approaches.

The questions from the mid-semester survey included some open-ended questions related to the assignments. Table II provides the questions that were asked. In addition to these questions, students were also asked to rate the relative difficulty of using Pattern Programming, MPI, and the Paraguin compiler directives using a six-point Likert scale from "very easy" (1) through "very difficult" (6). The purpose of this mid-semester survey was to compare and contrast our new approaches to parallel programming with just using MPI. The goal of this survey was to help us revise our materials.

V. Results

At the outset of the course, students responded in the "disagree" to "mildly disagree" range for all items presented indicating that students were not familiar with the topics or methods. However, by the conclusion of the course, students responded in the "mildly agree" to "agree" range to the same survey items, indicating that they learned the topics and how to use the methods (Table III).

TABLE I. Pre- and Post-semester survey questions

Item
I am familiar with the topic of parallel patterns for structured parallel programming.
I am able to use the pattern programming framework to create a parallel implementation of an algorithm.
I am familiar with the CUDA parallel computing architecture.
I am able to use CUDA parallel computing architecture.
I am able to use MPI to create a parallel implementation of an algorithm.
I am able to use OpenMP to create a parallel implementation of an algorithm.
I am able to use the Paraguin compiler (with compiler directives) to create a parallel implementation of an algorithm.

The students indicated that they mostly did not feel able to use the tools to implement algorithms in parallel in the beginning of the semester. By the end of the semester, the students were mostly confident in their ability to implement parallel algorithms. Naturally, this is expected. What is not expected is that the students indicated greater confidence in using the lower level parallel tools (MPI, OpenMP, and CUDA) than in using our new approaches (patterns and the Paraguin compiler). There are two possible explanations for this lower confidence in the higher level tools: 1) the tools need improvement to be easier to use; and 2) students preferred the flexibility and control of the lower level tools. Based upon the next set of data, both explanations are true.

The students were asked to provide open-ended comments comparing and contrasting the Seeds tool (Pattern Programming) with MPI. Table IV shows the comments made by students. There were more comments, but they were left out to conserve space.

The second comment describes well the students' sentiments. Computer Science students are accustomed to having control. It is true that the Seeds framework takes control from the user when the user is working within the "basic" layer of the framework. The framework is constructed in three layers, the "basic" layer, the "advanced" layer, and the "expert" layer. The basic layer provides standard well-established patterns, and the programmer need only implement a few simple Java interface methods. The advanced layer exposes some of the internal routines to

TABLE II. Open-ended questions comparing the methods used for parallel programming.

Item
Describe the benefits and drawbacks between the following methods: Pattern Programming (Assignment 1) and MPI (Assignment 2).
Describe the benefits and drawbacks between the following methods: Pattern Programming (Assignment 1) and Paraguin Compiler Directives (Assignment 3).
Describe the benefits and drawbacks between the following methods: MPI (Assignment 2) and Paraguin Compiler Directives (Assignment 3).

TABLE III. PRE- AND POST-SURVEY RESULTS OF FAMILIARITY WITH TOPICS AND METHODS

Item	Pre	Post
	Mean (sd) N=21	**Mean (sd) N=16**
I am familiar with the topic of parallel patterns for structured parallel programming.	2.74 (1.59)	4.44 (1.09)
I am able to use the pattern programming framework to create a parallel implementation of an algorithm.	2.38 (1.60)	4.25 (0.86)
I am familiar with the CUDA parallel computing architecture.	2.29 (1.55)	4.63 (0.72)
I am able to use CUDA parallel computing architecture.	1.95 (1.43)	4.44 (0.89)
I am able to use MPI to create a parallel implementation of an algorithm.	2.24 (1.26)	4.88 (0.81)
I am able to use OpenMP to create a parallel implementation of an algorithm.	2.19 (1.12)	5.06 (1.24)
I am able to use the Paraguin compiler (with compiler directives) to create a parallel implementation of an algorithm.	1.76 (0.89)	4.13 (1.15)

enable new patterns to be created or existing patterns to be optimized. The expert layer exposes the deployment and security services for the programmer who wants to increase the performance. We use the basic layer in our undergraduate parallel programming class. We feel that the advanced layer would have provided the students with a comparable feeling of "control" over their programs. This is an easy modification although it will take time away from other aspects of the course. Nonetheless, we are encouraged by the comments indicated the relative ease with using the Seeds framework.

The students were also asked to provide open-ended comments comparing and contrasting using the Paraguin compiler with compiler directives to generate MPI code with programming directly with MPI. Table V shows the comments made by students.

From the comments made by student in Table V, we see that some students found using the compiler directives to be easier and some did not. On the other hand, students did not seem to have the same "control" issue they did with the Seeds framework. The Paraguin compiler generated the source MPI code. The user is free to inspect the resulting code and even modify and recompile it. Therefore, the students could see how the abstraction was being implemented.

Table VI shows the results of comparing the relative difficulty of the three methods used to program parallel algorithms. The results show that students felt that Pattern Programming was the easiest of the three while the Paraguin compiler directives approach was the hardest. This agrees with the students' responses to the open-ended questions

TABLE IV. OPEN-ENDED STUDENT COMMENTS COMPARING PATTERNED PROGRAMMING WITH MPI

"MPI is more flexible but pattern programming is easy to use."
"Using the seeds framework for pattern programming made it easy implement patterns for workflow. However, seeds works at such a high level that I do not understand how it implements the patterns. MPI gave me much more control over how program divides the workflow, but it can often be difficult to write a complex program that requires a lot of message passing between systems."
"mpi is easier to understand than seeds"
"Since our implimentation [sic] of MPI was in c, it allowed far more control and a higher level of efficiency than Seed's allowed for."
"Pattern programming is more high-level and easier to implement, while MPI is more lower-level and much more difficult to implement."
"Assignment 2 was more straight forward and easier to understand than Pattern Programming. I feel that because the Pattern Programming was a new topic it was assumed it would be easier than what it was."
"Pattern Programming offers a higher level of abstraction when designing parallel programs, but [MPI] gives the programmer greater control over the internals, which aids in solving particular problems. Each method's strengths could, to certain people, also be its weakness"
"Pattern programming is pre set up and easier to implement then mpi. However mpi seems to permit you to split things up more as you want. Also the pattern programming framework we used wont' [sic] work in c."
"Pattern programing provides various forms of message passing methods. For programing with MPI is a bit difficult."
"Pattern Programming:- Liked using java, easier to understand what was going on.- Seeds was not stable, sometime would work and sometimes would not work MPI:- Easier to see the parrellzation- Much more difficult to use in C when I had never used the language before. Not enough time was dedicated to going over MPI"

comparing the three methods. Interestingly, the standard deviation went up as the mean went down.

We feel that the concerns of the Paraguin being complicated are legitimate. The compiler was designed to provide significant flexibility over partitioning nested loops. This flexibility overly complicated the partitioning directive. We have already implemented a simpler version of the directive to parallelize a loop, which will be introduced in the next offering of this course.

VI. CONCLUSIONS

In this paper, we present preliminary results from a course on Parallel Programming where we tried two new methods to raise the level of abstraction for parallel computing. The first method used a framework for specifying patterns. The second method used compiler directives to generation MPI code. We used survey instruments to measure students' understanding of the concepts as well as have them compare the various methods

TABLE V. OPEN-ENDED STUDENT COMMENTS COMPARING COMPILER DIRECTIVES USING THE PARAGUIN COMPILER WITH MPI

"I found Paraguin to be useful because it eliminated the need for me to write the more complex message passing routines in MPI. However, I found it difficult to debug errors in the code and also determine the correct loop dependencies."
"Paraguin is incomplete, confusing, and has no debugging tools."
"Paraguin Compiler was much easier to use because it would generate the MPI code automatically."
"The Paraguin assignment was simpler and easier to understand how it worked as you could see the code which the compiler actually made for you."
"Paraguin might be beneficial for some parallelizing problems that have a lot of layers of interior loops."
"Paraguin overcomplicates something simple. Using paraguin takes more time to formulate the required input than the time spent typing straightforward for loops."
"MPI is the underlying code that gets generated by Paraguin, so the comparison is slim. MPI can give the programmer more control over internal actions, but you lose the abstraction Paraguin offers"
"Programming is lengthy for mpi compare to paraguin for the same reasons as parallel."
"I did like how we did MPI before paraguin. Understanding MPI was much more helpful when it came time to do paraguin. MPI was more difficult because of trying to visualize the scatter, broadcast, and gather methods while trying to keep the partitioning to the number of processors. Paraguin takes care of that for you with the parallel pragma statements."

TABLE VI. RELATIVE DIFFICULTY OF THE THREE METHODS OF PARALLEL COMPUTING

	Mean (sd)
Pattern Programming	3.63 (0.89)
MPI	3.25 (1.13)
Paraguin Compiler Directives	2.56 (1.26)

and tools. We are using this information to revise the materials we've developed.

What we have discovered from the material presented here is that the students would benefit from using the Seeds framework at an advanced layer instead of simply the basic layer. Not only will this give the students' a sense of control over their programs, the will also benefit from seeing how their concepts of parallelization are being implemented. Second, we learned that the compiler directives were designed in a way that overly complicates matters. We have already started and partially completed the reworking of the compiler directives to make the simpler. We expect the new directives to be much easier for undergraduate students to learn and use.

The perceived advantage of our pattern programming framework was that it provided the programmer a higher level and simpler interface avoid low level implement details – this then leads to more structured scalable designs. However the students did not fully appreciate this model although many would have seen design patterns in software engineering courses. We feel that we can accomplish our goal of raising the level of abstraction for parallel computing by using more sophisticated engineering applications that can be implemented using patterns.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under the collaborative grant #1141005/1141006. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Astrachan, O. 1998. Design Patterns: An Essential Component of CS Curricula, *SIGCSE Bulletin and Proceedings*. 30, 1, 153-160.

[2] Ferner, C.S. 2006. Revisiting communication code generation algorithms for message-passing systems, *International Journal of Parallel, Emergent and Distributed Systems (JPEDS) 21(5)*, 323-344.

[3] Ferner, C. S. 2002. The Paraguin compiler---Message-passing code generation using SUIF, in *Proceedings of the IEEE SoutheastCon 2002*, Columbia, SC, 1-6.

[4] Fastflow. University of Torino, Italy /Università di Pisa. http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about

[5] Gamma, E., Helm., R., Johnson, R., and Vlissides, V. 1995. *Design Patterns.* Addison-Wesley, New York.

[6] Keutzer, K., and Mattson, T. Our Pattern Language (OPL): A Design Pattern Language for Engineering (Parallel) Software. http://parlab.eecs.berkeley.edu/wiki/_media/patterns/paraplop_g1_1.pdf.

[7] Mattson, T. G., Sanders, B. A., and Massingill, B. L. 2004. *Patterns for Parallel Programming.* Addison Wesley.

[8] McCool, M., Reinders, J., and Robison, A. 2012. Structured Parallel Programming: Patterns for Efficient Computation. Morgan Kaufmann.

[9] Renault, E., Parrot, C. MPI Pre-processor: Generating MPI Derived Datatypes from C Datatypes Automatically, in *Proceedings of the2006 International Conference on Parallel Processing Workshops (ICPPW'06)*, Columbus, OH, August 14-18, 2006.

[10] Reyes, R., Dorta, A.J., Almeida, F., Sande, F., Automatic hybrid MPI+OpenMP code generation with llc, in *Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7-10, 2009.

[11] Toub, S. Patterns of Parallel Programming Understanding and Applying Parallel Patterns with the .Net Framework 4 and Visual C#. Microsoft. http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=19222

[12] Wilkinson, B., Villalobos, J., and Ferner, C. 2013. Pattern Programming Approach for Teaching Parallel and Distributed Computing. *SIGCSE 2013 Technical Symposium on Computer Science Education*. Denver, Colorado. To appear.

[13] Villalobos, J. 2011. Running Parallel Applications on a Heterogeneous Environment with Accessible Development Practices and Automatic Scalability. PhD diss. University of North Carolina Charlotte.

[14] Villalobos, J. Parallel Grid Application Framework. http://coit-grid01.uncc.edu/seeds/