2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Using Patterns 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—**In this paper, we describe the results of teaching a parallel programming course using a pattern programming approach in a course taught across the State of North Carolina on a televideo network in Fall 2013. Five universities participated in this study. The course begins with a higher-level tool called the Seeds framework that creates and executes high-level message passing patterns such as a workpool without writing low level MPI code. To avoid going directly to MPI next, we used another tool (Paraguin compiler) which uses compiler directives to create MPI code for patterns. Once students understand the pattern programming approach we then present low level MPI routines and their more complex parameters but now with the knowledge of parallel patterns. An independent professional evaluator is employed to deploy survey instruments and produce an analysis of the results. The lessons we learned from this data collected in Fall 2013 are: 1) Teaching parallel computing in the context of patterns has a positive impact on student learning; 2) Teaching the lower level tools first would be beneficial; 3) The improvements made to the Paraguin compiler directives significantly improved the students confidence in using the tool; and 4) The lower level tools can still be taught in the context of patterns.**

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

I. INTRODUCTION

Parallel computing and programming, once regarded as specialized area, is now becoming central part of the undergraduate computer science curriculum especially with the advent of multicore processors. Computer science students should understand how to program multicore and distributed-memory computer systems now that these systems are widespread. An approach to parallel programming that is becoming recognized as a way to create parallel programs that are scalable and maintainable in a professional environment involves using parallel design patterns. This concept comes directly from using design patterns in software engineering, and tutorials now appear in major scientific computing conferences [8]. Bringing these concepts into the undergraduate curriculum for parallel programming began recently, with work by the authors [2], [11], and others [1].

Pattern programming involves writing programs that conform to standard and well-known parallel computational patterns. These patterns can be high level application patterns such as workpool, pipeline or stencil patterns or lower level patterns such as broadcast and scatter. Higher level patterns are in fact constructed from lower level patterns. Our interest is mostly in higher level patterns and in producing and using automated ways of creating executable code without writing any code in low level MPI or OpenMP APIs.

In Fall 2012, we first taught a course on parallel computing using two tools that allowed the students to create parallel program expressed as patterns. This course was taught jointly at the University of North Carolina Charlotte and the University of North Carolina Wilmington. It was delivered to the two campuses using the North Carolina Research and Education televideo network (NCREN) connecting universities across North Carolina. Authors Ferner and Wilkinson co-taught the course from their respective universities. The course was taught a second time in Fall 2013, again using NCREN. The university participating in Fall 2013 were: University of North Carolina Wilmington, University of North Carolina Charlotte, North Carolina A&T University (a minority serving institution), East Carolina University, and University of North Carolina Greensboro.

In this paper, we will report on how students find these automated ways of created pattern-based parallel programs compared to creating more traditional MPI or OpenMP programs. The two high-level tools that we used were the *Seeds Framework*, developed at UNC-Charlotte, and the *Paraguin compiler*, developed at UNC-Wilmington. 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. 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 69 students in Fall 2013 (compared to 58 students in Fall 2012). The class was a mix of undergraduate and graduate students, all studying computer science, with approximately half being undergraduate students. The prerequisites for the course are two semesters of programming plus a course on data structures.

Co-author Heath gathered the same teaching effectiveness data on both Fall 2012 and Fall 2013 offerings. The results from the analysis of the data collected from the Fall 2012 course were published in [2]. Using these results, we made modifications to our materials. What we found from the Fall 2012 course is that: 1) students prefer the flexibility and control the lower level tools provide; and 2) the Paraguin compiler directives were difficult to understand and use. Based upon those conclusions, the Paraguin compiler directives were redesigned to make them more intuitive. Furthermore, we refined and rearranged the presentation of our materials to have a more pattern-based approach. In this paper, we present the results of the finding from the Fall 2013 course, compare those results with the results from Fall 2012, and present the lessons we learned from teaching using our tools and new materials.

The rest of this paper is organized as follows. Existing work is briefly reviewed in Section II. In Section III, we describe our pattern programming approach. In Section IV, we describe the survey instruments. In Section V, we present and discuss the results. Section I describes future work. Finally, Section II concludes.

II. Existing Work

Mattson et al. [6] wrote a book on design patterns in parallel programming, published in 2004. Subsequently a number of research projects explored parallel patterns. A pattern programming language called OPL (Our Pattern Language) was developed by Keutzer et al.[5] Intel and Microsoft have interest in pattern programming. McCool et al. [7] wrote a text published in 2012 on parallel pattern using Intel tools notably Threading Building Blocks (TBB), Intel Cilk plus, Intel Array Building Blocks (ArBB),and presented a tutorial at Supercomputing in 2013 [8].

Using parallel patterns in undergraduate parallel programming classes is a recent development. A 2007-2011 UNC-Charlotte PhD project [13] exploring pattern programming directly led to using pattern programming in the classroom at UNC-Charlotte [12]. Funding was obtained from National Science Foundation in 2012 to develop educational materials based upon the pattern programming approach. This collaborative NSF project brought together two complementary approaches: the high-level Seeds pattern programming framework developed at UNC-Charlotte [13][14], and a compiler directive approach (Paraguin compiler) developed at UNC-Wilmington [3][4], both as research efforts. We combined the approaches with an integrated pattern programming based course presented here.

Pattern programming is being promoted in the classroom elsewhere. Adams et al. describe *patternlets*, “fully operational but minimalist programs that illustrate the pattern’s use and behavior in a given parallel platform.” They also introduce *exemplars*- “representative and compelling applied problems together with implementations in different parallel technologies” and combine exemplars with patterns.[1]

In the area of compiler directives, Renault and Parrot [9] created a pre-processor that can automatically generate MPI derived datatypes from the C data types. This pre-processor does not generate the code to parallelize an algorithm but rather assists the programmer with creating the complex MPI datatypes needed for the transmission of user-defined datatypes.

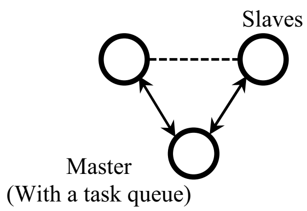
The closest work to our research work on compiler directives is the llCoMP compiler for the llc language [10]. The llc language allows the programmer to specify parallel constructs for both MPI and OpenMP using llc and OpenMP pragma statements. It appears that the use of patterns in the llc compiler is specific to 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; however, the constructs are specifically intended to describe the pattern to which the algorithm adheres.

Although others are using patterns to teach parallel computing, we are using tools that automatically generate code based upon a pattern.

III. Heirarchical approach to pattern programming

Our approach begins with high level message passing patterns, implemented through the Seeds framework. Then, we use the Paraguin compiler approach to implement the message passing patterns, still hiding the underlying implementation, and finally we delve into low level MPI. Figure 1 shows the first pattern we introduce, the workpool pattern, and the kernel of the workpool pattern algorithm implemented using the Seeds framework. The algorithm is the Monte Carlo method for approximating π. The Seeds framework is Java-based that will self-deploy on multicore or distributed computing platforms and is described fully elsewhere [11],[12],[13],[14]. Several patterns are pre-implemented and we start with the workpool pattern. Students install Seeds on their own computer to do the assignment. The key aspect is that the programmer does not need to write the code to create the selected pattern or the message passing for the selected pattern. In the workpool pattern (Figure 1), there are three principal methods that need to be implemented, a method that specifies the data to be sent from the master to the slaves (DuffuseData()), the method that specifies the computation that the slaves do (Compute()) and the method that specifies what data is collected by the master at the end (GatherData()).

In the compiler directive approach, students use pragma statements to guide the Paraguin compiler for how to create the message-passing code to implement the desired pattern. The Paraguin compiler directives were designed to model OpenMP yet create MPI code suitable for execution on a distributed-memory system. Figure 2 demonstrates how the students would implement the scatter/gather pattern using the Paraguin compiler as well as using MPI. The scatter/gather pattern is a low-level pattern, so we discuss that pattern early



(a) Workpool pattern

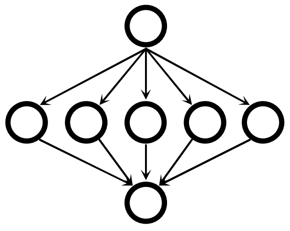
```
public Data DiffuseData (int segment) {
    DataMap<String, Object> d=
              new DataMap<String, Object>();
    d.put("seed", R.nextLong());
    // returns a random seed for each job unit
    return d;
}

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 (inti = 0; i<DoubleDataSize ; i++) {
        double x = r.nextDouble();
        double y = r.nextDouble();
        doubledist = 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;
}
```

(b) Workpool pattern implemented in Seeds Framework

Figure 1: Workpool Pattern



(a) Scatter andgather patterns

```
#pragma paraguinbegin_parallel
#pragma paraguin scatter A B
 ...
#pragma paraguin gather C
#pragma paraguinend_parallel
```

(b) Paraguinscatter/gather pattern code

```
MPI_Scatter(A,N,MPI_DOUBLE,A,
          N,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Scatter(B,N,MPI_DOUBLE,B,
          N,MPI_DOUBLE,0,MPI_COMM_WORLD);
 ...
MPI_Gather(C,N,MPI_DOUBLE,C,
          N,MPI_DOUBLE,0,MPI_COMM_WORLD);
```

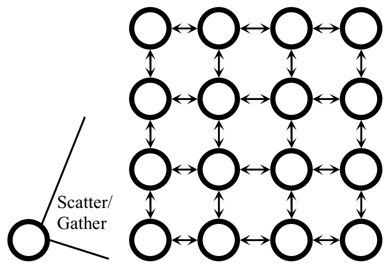
(c) MPI scatter/gather pattern code

Figure 2: Scatter/Gather Pattern

in the semester. The scatter/gather pattern is one where the master process distributes the input data to the workers. Each worker computes partial results from the input, and then the partial results are gathered back to the master. Because the individual workers compute their partial results completely independently, this is a fairly easy pattern to understand for the students as well as it can produce speedup even on a distributed-memory system.

Figure 3 shows the stencil pattern using the Paraguin compiler. The stencil pattern is a pattern where neighboring processors communicate data after each iteration. Heat distribution is an algorithm that fits this pattern, and we ask the students to model the heat distribution throughout a room from a fireplace.

To use the Paraguin stencil pattern, the students must declare a 3-dimensional array (the 1st dimension is for the new and old values computed at each iteration) as well as fill it with initial values. The students must also provide a function that will compute the new values based upon the values of the last iteration. All of the remaining code to implement the stencil pattern in a message-passing environment is created for them. The code to scatter and gather the input and partial results and the code to have each processor communicate its computed values with its neighbors is all created for them.



(a) Stencil Pattern

```
double f (double A[][M], int i, int j) {...}

int main (int argc, char *argv[]) {
    int n, m, max_iters;
    double A[2][N][M];
    ...
    #pragma paraguin begin_parallel
    n = N;
    m = M;
    max_iters = TOTAL_TIME;
    #pragma paraguin stencil A n m max_iters f
    #pragma paraguin end_parallel
```

(b) Paraguin stencil pattern code

Figure 3: Stencil Pattern

Figure 4 shows the steps that are inserted into the resulting code to implement the stencil pattern. The code that is inserted is MPI code.

The results of the study from Fall 2012 [2] showed that the students found the Paraguin compiler difficult to use. In the year between Fall 2012 and Fall 2013, the compiler directives were redone to make them simpler and more intuitive, and better documentation was provided. Furthermore, the stencil pattern was added to the Paraguin compiler during this year.

After the students complete the assignment using the Seeds framework and the assignment using the Paraguin compiler, we have them use MPI to implement the scatter/gather pattern and the workpool pattern. We do not have the students implement the stencil pattern using MPI because of the level of difficulty; however, the students can see the implementation. One advantage of using the Paraguin compiler is that it is a source-to-source compiler. The user may inspect as well as modify and re-compile the resulting MPI code it produces.

IV. Survey Instruments

During both offerings of our course in Fall 2012 and Fall 2013, students were invited to provide feedback that would assist with the development of the future course offerings.

The Stencil Pragma is Replaced with Code to do:

1) The 3-dimensional array given as an argument to the stencil pragma is broadcast to all available processors.
2) current is set to zero and next is set to one.
3) A loop is created to iterate max_iteration number of times. Within that loop, code is inserted to perform the following steps:
    a. Each processor (except the last one) will send its last row to the processor with rank one more than its own rank.
    b. Each processor (except the first one) will receive the last row from the processor with rank one less than its own rank.
    c. Each processor (except the first one) will send its first row to the processor with rank one less than its own rank.
    d. Each processor (except the last one) will receive the first row from the processor with rank one more than its own rank.
    e. Each processor will iterate through the values of the rows for which it is responsible and use the function provided compute the next value.
    f. current and next toggle
4) The data is gathered back to the root processor (rank 0).

Figure 4: Implementation of the Stencil Pattern

Feedback was collected by the external evaluator via three 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 in Fall 2012, 58 invitations were sent to students at both campuses. The response rates for the three surveys were: 36%, 29%, and 28%, respectively. In Fall 2013, 69 invitations were sent to students at all campuses. The response rates for the three surveys were: 55%, 32%, and 45%, 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 five 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.

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 difficult” (1) through “very easy” (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.

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 able to use the Paraguin compiler (with compiler directives) to create a parallel implementation of an algorithm.
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.

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).

V. RESULTS

At the outset of the course for both offerings, 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). The number of participants (*N*) is smaller here because not all students answered all questions on the survey and the external evaluator matched the responses of the students between the two surveys. In other words, the number of participants shown in Table III is the number of participants that answered BOTH sets of questions.

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 was not expected was the result from Fall 2012 indicating the students had greater confidence in using the lower level parallel tools (MPI and OpenMP) than in using our new approaches (patterns and the Paraguin compiler). Based upon that information, we improved our materials on the Seeds Framework as well as the Paraguin compiler directives. Most notably, the Paraguin compiler directives were re-done so as to make them more intuitive.

There was an improvement in the percentage gain of the students' ability to use our tools from the Fall 2012 offering to the Fall 2013 offering. We attribute this to the improvements made to the materials throughout the 2013 calendar year. These improvements seem to have had a positive impact on student learning, particularly on the students' ability to use the Paraguin compiler which increased a full point from "mildly agree" to "agree" with a lower standard deviation.

Table IV rates the relative difficulty between Seeds, Paraguin, and MPI using a six-point Likert scale from "very difficult" (1) through "very easy" (6). With the improvements made to the Paraguin compiler directives, the students found the compiler much easier to use. On the other hand, students found MPI to be more difficult. The reason for this is shown in the following tables and discussed next.

The students were asked to provide open-ended comments comparing and contrasting the Seeds tool (Pattern Programming) with MPI and the compiler directive approach with MPI. Tables V and VI give some of the answers by students that shed some light on the difficulty students had

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

Item	Fall 2012		Fall 2013	
	Pre	Post	Pre	Post
	Mean (sd) N=16	Mean (sd) N=16	Mean (sd) N=21	Mean (sd) N=21
I am familiar with the topic of parallel patterns for structured parallel programming.	2.56 (1.59)	4.44 (1.09)	3.14 (1.42)	4.95 (1.07)
I am able to use the pattern programming framework to create a parallel implementation of an algorithm.	2.25 (1.61)	4.25 (0.86)	3.00 (1.52)	5.05 (0.50)
I am able to use the Paraguin compiler (with compiler directives) to create a parallel implementation of an algorithm.	1.69 (0.95)	4.13 (1.15)	2.43 (1.50)	5.14 (0.79)
I am able to use MPI to create a parallel implementation of an algorithm.	2.31 (1.40)	4.88 (0.81)	2.14 (1.24)	4.95 (0.74)
I am able to use OpenMP to create a parallel implementation of an algorithm.	2.19 (1.17)	5.06 (1.24)	2.33 (1.46)	5.19 (0.75)

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

	Fall 2012	Fall 2013
	Mean (sd)	Mean (sd)
Pattern Programming	3.63 (0.89)	3.95 (1.19)
MPI	3.25 (1.13)	2.05 (0.94)
Paraguin Compiler Directives	2.56 (1.26)	3.37 (1.26)

with MPI. In particular, the 2nd comment from Table V and the 3rd comment from Table VI indicate that the students would have benefited from learning MPI before using our tools, although not all agree (see comment 6 from Table VI). First, students would have a better understanding of what the tools are doing for them and how the patterns are actually implemented. Second, MPI seemed significantly harder after covering the more abstract tools. Third, the Paraguin compiler requires some knowledge of both OpenMP and MPI. Teaching the Paraguin compiler directives first required us to introduce concepts without the full context in which they would normally be taught.

It is interesting that several students indicate the better control over implementation provided by MPI was desirable. Computer science students seem to prefer control as well as the concreteness of the implementation of their algorithms. We observed similar comment from students in the Fall 2012 offering.

The lessons we learned from this data collected in Fall 2013 are:

1) Teaching parallel computing in the context of patterns has a positive impact on student learning.
2) Teaching the lower level tools first would be beneficial because:
   a. MPI is more difficult to use and learn
   b. Computer science students understand higher level tools better by first seeing their implementation
   c. Teaching the Paraguin compiler directives requires some knowledge of OpenMP and MPI
3) The improvements made to the Paraguin compiler directives significantly improved the students confidence in using the tool
4) The lower level tools can still be taught in the context of patterns

I. FUTURE WORK

In our first programming assignment using the Seed pattern programming framework, students install the framework on their own computer to do the assignment. Students report they like this approach. The subsequent assignments involving Paraguin, MPI, and OpenMP were done on remotely accessed clusters. Using one's own computer would have a number of advantages. Programs can be quickly compiled and tested. It reduces issues such as poor connections to the cluster and faulty user programs running on the cluster that are affecting response time. With this in mind, we have changed our MPI assignments for Spring 2014 at UNC – Charlotte so that students first test their programs on their own computer before uploading and testing on a cluster. This requires students to install an implementation of MPI (usually OpenMPI or MPICH). We currently accept any approach including direct installation within the native OS or installing a VM such as VirtualBox together with Linux, or even Cygwin, which comes with OpenMPI libraries. Most students in our classes use a Windows personal computer. A similar approach can be done with OpenMP.

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

"Benefits: MPI provided a deeper understanding of low-level code and the utilization of such. Drawbacks: Assignment 3 left me feeling the most lost of any assignment yet this semester. Lots of banging my head against my desk, so to speak."
"Pattern programming was somewhat helpful in seeing how certain patterns worked. However, it would have been easier for me to start with MPI first to get a lower level knowledge of parallel programming."
"Pattern programming allows for a higher level of abstraction, which in turn allows the programmer to focus on the computation rather than passing of information between the processes. MPI uses a low level of abstraction. While at times more challenging, it allows the programmer more control over the processes, communication, and resource allocation."
"Pattern programming is simpler to code and understand--at first glance. Simple logic dictates the algorithm and pattern. However, compared to MPI, it is not as close to the parallel concept of programming. Hence, it is more expensive and creating a data structure (hash) require additional expense and implementations. MPI, although complex, distributes and executes in various processors assigned. More control on parallel algorithm implementation and execution."
"Patterns programming is definitely easier. I think knowing how MPI works is very useful."
"Pattern is very easy to use and MPI seems a little awful to use it, but MPI give me a better understanding about how these patterns are implemented step by step and I think it's very useful."
"MPI is more difficult to use."
"Seeds is a higher level programming construct, therefore easier to implement. MPI is more powerful because it can be implemented however the user desires."

We will be teaching our collaborative course in Fall 2014 on NCREN. What we plan to do differently is introduce the lower level tools (MPI and OpenMP) before introducing the higher-level tools (Seeds and Paraguin). Moreover, we plan to reorganize the materials to be more pattern-centric than tool-centric. In the two previous NCREN offerings of the course, we started with one tool and discussed specifics, patterns and examples of that tool before moving on to the next. In the next offering, we plan to start with one pattern, patterns, or classes of patterns, then introduce the different tools, specifics, and examples of implementing the patterns using the tools. We will start with the lower-level tools for a given pattern or class of patterns before moving on to the higher-level tools. This will give the computer science students the desired feel of "control" as well as give them an

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

“Paraguin Compiler Directives could generate MPI code, it's easier to apply than MPI; however, not as flexible and powerful as MPI. It can only generates simple MPI routine.”
“Paraguin removes the difficulty of message passing, and incorporates some useful design patterns. MPI allows complete control, but it is up to the programmer to implement the needed patterns for computation.”
“Paraguin should be covered after MPI. The documentation provided for MPI used variables with no explanation as to where the variables came from. I'm interested in doing further MPI work, but it appears than I'm going to have to research it myself because the course material was extremely vague and confusing to follow. Doing a better job organizing the MPI material to grasp the concepts will be more beneficial to understand what the Paraguin compiler is actually doing. Endstate, Paraguin is much more clear, concise and intuitive than MPI.”
“Paraguin is easier to use than MPI and offers built in patterns.”
“Once again, Paraguin is easier to utilize. I still don't get MPI.”
“I've gone back and forth on this, but I think I enjoyed doing Paraguin before MPI. It introduced the concepts behind MPI before they were forced on us in assignment 3. Still though, assignment 3 was difficult. Assignment 2 was not. I feel like both were helpful in terms of gaining knowledge.”
“Paraguin compiler directives call MPI routines automatically however using MPI directly lets the developer be more explicit.”

appreciation for all that the higher level tools are doing on their behalf. It will also allow us to demonstrate how to use the higher level tools in a way that gives the full control MPI provides.

We are also continuing to develop our tools. We have plans to implement additional patterns such as: pipeline, divide and conquer, and all-to-all.

II. CONCLUSIONS

In this paper, we described the results of teaching a parallel programming course using a pattern programming approach in a course taught across the State of North Carolina on a televideo network in Fall 2013. Five universities participated in this study. The course begins with a higher-level tool called the Seeds framework that creates and executes high-level message passing patterns such as a workpool without writing low level MPI code. To avoid going directly to MPI next, we used another tool (Paraguin compiler) which uses compiler directives to create MPI code for patterns. Once students understand the pattern programming approach we then present low level MPI routines and their more complex parameters but now with the knowledge of parallel patterns. An independent professional evaluator is employed to deploy survey instruments and produce an analysis of the results.

In Fall 2012, students had indicated that they found the lower level tools easier to use than the higher level tools. With the modification made over the 2012-2013 academic year reported in this paper, students in the Fall 2013 course indicated the opposite. The students found the tools easier to use that the low-level APIs, such as MPI. Furthermore, the students understood how to implement an algorithm using the pattern that fits it. Many of our students felt MPI was very difficult and some expressed to us verbally a sincere disdain for MPI. However intellectual it might be to do a top-down approach, computer science students learn this material better using a bottom-up approach. It was interesting to note that several students preferred the increased control provided by the lower level tools. Still, introducing the material, whether it be first with the lower-level tools or the higher-level tools, by presenting them in the context of implementing a pattern improves student learning. Regardless of the tool being used, implementing an algorithm by first identifying its pattern and then following known implementations for that pattern have a positive impact on students’ ability to create parallel programs.

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] Adams, J., Brown, R., and Shoop E. 2013. Patterns and Exemplars: Compelling Strategies for Teaching Parallel and Distributed Computing to CS Undergraduates,*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.

[2] Ferner, C., Wilkinson, B., and Heath, B. 2013. Toward using higher-level abstractions to teach Parallel Computing, *Third NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar-13),* Boston, MA, May 20, 2013.

[3] 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.

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

[5] 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.

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

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

[8] McCool, M. D. , Reinders, J. R. , Robison, A., and Hebenstreit, M. 2013. Structured Parallel Programming with Patterns,*Supercomputing, SC 13 Technical Program Tutorial*, Denver CO, Nov 17, 2013.

[9] Renault, E., and Parrot, C. 2006. 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.

[10] Reyes, R., Dorta, A.J., Almeida, F., and Sande, F., 2009. 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,.

[11] Wilkinson, B., and Ferner, C. 2013. Workshop 31: Developing a Hands-on Undergraduate Parallel Programming Course with Pattern Programming,*SIGCSE 2013, The 44th ACM Technical Symposium on Computer Science Education*, Denver, USA, March 9.

[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, March 8.

[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/.