SIGCSE 2013 - The 44th ACM Technical Symposium on Computer Science Education
March 6-9, 2013, Denver, USA

Pattern Programming Approach for Teaching Parallel and Distributed Computing

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

Jeremy Villalobos
Formerly of
University of North Carolina Charlotte
9201 University City Blvd.
Charlotte, NC 28223 USA
jeremyvillalobos@gmail.com

Clayton Ferner
University of North Carolina Wilmington
601 S. College Rd.
Wilmington, NC 28409 USA
cferner@uncw.edu

ABSTRACT

In this paper, we describe an approach for teaching parallel and distributed computing at the undergraduate level using computational patterns. The goal is to promote higher-level structured design for parallel programming and make parallel programming easier and more scalable. A pattern programming framework has been developed to create a distributed application that avoids the need to write code in low level message–passing APIs such as MPI. Several patterns have been implemented including workpool, pipeline, synchronous and iterative all-to-all, and stencil. We have redesigned a regular senior undergraduate parallel programming course to begin with a pattern strategy using our framework and provide a detailed syllabus around patterns.

Categories and Subject Descriptors
D.1.3 [**Programming Techniques**]: Concurrent Programming – *distributed programming, parallel programming.*

General Terms
Design, Languages.

Keywords
Pattern Programming; Undergraduate Education.

1. INTRODUCTION

With the widespread introduction of processors with multiple cores, multicore processors have mostly replaced single core designs. Present processors typically have four execution cores. We can expect an increasing number of cores in the future. Intel's first commercial product in its recent Many Integrated Core (MIC) architecture has 32 Intel cores in one package [7]. Systems with a small number of cores such as present 4-core designs have a so-called shared-memory architecture where the cores share common main memory. As the number of cores increases, it becomes necessary to use distributed-memory and hierarchical-memory models. Programming shared-memory and distributed-memory systems is taught in senior undergraduate and first year graduate courses typically called parallel programming. Parallel programming has a very long history as a specialty area for those interested in high performance computing. However, with advent of multicore systems, all computer science students should now be trained in dealing with these systems. A typical parallel programming course focuses on using low-level libraries – MPI for message passing distributed memory systems, OpenMP and other thread tools for shared memory systems, and CUDA/OpenCL for high performance GPU computing. A course typically uses these tools to solve simple problems such as matrix multiplication and sorting. 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 a new software environment that creates a higher level of abstraction for parallel and distributed programming based upon a pattern programming approach. 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 rather than lower level constructs such as fork-join and loop. Our pattern programming framework was developed as part of a PhD project exploring distributed computing by Jeremy Villalobos [13]. 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. More details of the framework will be given later, but the key aspects are 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 programing interface avoiding the complexities of putting message-passing statements in the actual code.

The rest of this paper is organized as follows. Existing work is briefly reviewed in Section 2. Various patterns and the programmer interface for our framework are described in Section 3. In Section 4, we describe how our parallel programming course has been re-structured to start with patterns and Section 5 provides conclusions.

2. EXISTING WORK ON PATTERN PROGRAMMING

Design patterns, that is, “reusable solutions to commonly occurring problems” [12], have been part of software engineering for many years and are well-established [1], [5]. Design patterns provide a guide to “best practices” but not a final implementation. They provide scalable design structures allowing one to focus, understand, and reason more easily about the parallel algorithms. With the widespread introduction of multicore and many core systems, there has been some effort to make parallel and distributed programming easier and useable by using design patterns. Mattson, Sanders, and Massingill wrote an influential book promoting patterns for parallel programs in 2004 [8]. A pattern programming language called OPL (Our Pattern Language) was co-developed at the Universal Parallel Computing Research Centers (UPCRC) at University of Illinois at Urbana-Champaign and University of California, Berkeley with significant funding support from Microsoft and Intel. OPL is described by Keutzer and Mattson [6]. Twelve computational patterns are used: Finite State Machines, Circuits, Graph Algorithms, Structured Grid, Dense Matrix, Sparse Matrix, Spectral (FFT), Dynamic Programming, Particle Methods, Backtrack, Graphical Models, and Unstructured Grid, in seven general application areas. UPCRC promotes patterns in workshops on parallel programming patterns, ParaPLoP 2009, 2010, and 2011, and the Pattern Languages for Programs conference PloP’2009.

Intel has developed somewhat competing tools that use patterns; Thread Building blocks (TBB), Cilk Plus, and Array Building Blocks (ArBB). They embody low-level patterns such as fork-join. McCool, Reinders, and Robison wrote a recent book on using patterns in parallel programming, focusing on the Intel tools [9]. Microsoft describes using patterns in the .NET/C# environment [10] but again with low level patterns such as parallel for loops.

The closest to our work is Fastflow [4] from the University of Torino, Italy/Università di Pisa but uses C++ and focuses on shared-memory platforms.

Much of the quoted work tends to concentrate upon lower-level constructs such as fork-join and parallel loops – we concentrate upon a few higher-level and widely applicable computational patterns such as workpool and synchronous all-to-all. We also provide an automatic and seamless conversion into executable code for parallel and distributed platforms. All the related work mentioned in this section concentrates upon professional or research environments. We are interested in training students in a better way and introducing the pattern programming strategy in an undergraduate parallel programming course with automated pattern programming tools.

Note on terminology. Sometimes term “skeleton” is used to describe a “pattern”, especially a directed acyclic graph with a source, a computation, and a sink. We do not make that distinction and use the term “pattern” whether a directed or undirected graph and whether acyclic or cyclic.

3. PATTERN PROGRAMMING FRAMEWORK

3.1 Patterns

Figure 1 shows perhaps the most useful pattern, the workpool pattern. In this pattern, the master process sends tasks to worker or slave processes for them to perform. Our framework implements this pattern with a task queue within the master that hands out tasks to waiting worker processes. When a worker process completes a task and returns results to the master, the master gives it a new task to do, which provides automatic load balancing for the programmer. A tutorial has been written to use this pattern in our framework, complete with sample code [15].

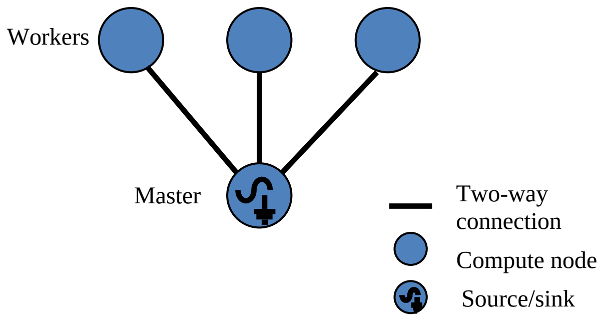

Figure 1 Workpool pattern.

Figure 2 shows another pattern implemented in our framework, the pipeline pattern. The master process sends a task to the first worker, which does some computation and passes the result onto the next worker and so on. In some ways, a pipeline models a normal sequential program, which has statements executed one after the other and could be introduced to students that way, but the pattern would particularly suit a situation in which data is fed from one process to the next. Bubble sort can map onto a pipeline as well as digital filtering. A tutorial has been written to use this pattern in our framework, complete with sample code [16].

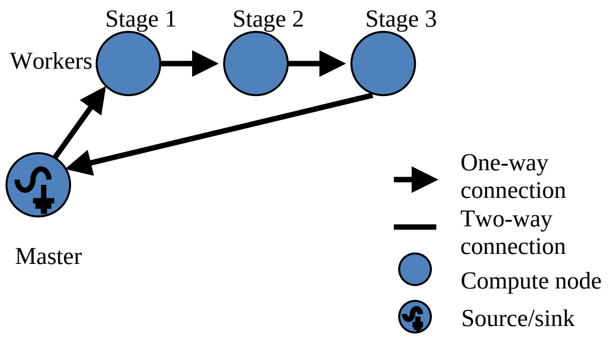

Figure 2 Pipeline pattern.

Figure 3 show the divide and conquer pattern, which has not yet been fully implemented in the framework, but the programmer can create it using more advanced tools in the framework. In the (binary) divide and conquer pattern, a problem is divided into two parts and then these parts are divided into two parts and so on until the problems has been divided into sufficiently small sub-problems. These are then computed and a reverse process is used to create the final result. Several well-known algorithms use divide and conquer such as Quicksort and Mergesort.

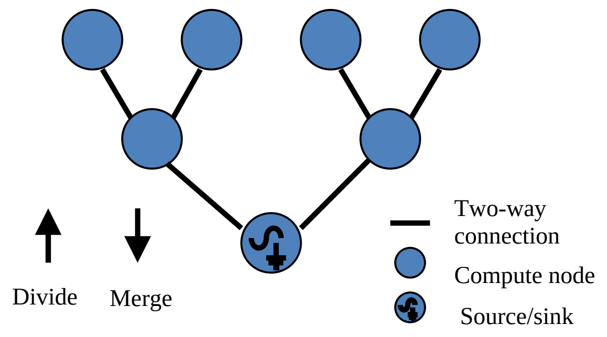

Figure 3 Divide and conquer pattern.

Figure 4 shows the synchronous stencil pattern, which is implemented in our framework. This pattern performs a number of iterations to converge on a solution, for example for solving Laplace's/heat equation by iteration. A tutorial has been written to use this pattern in our framework, complete with sample code [17].

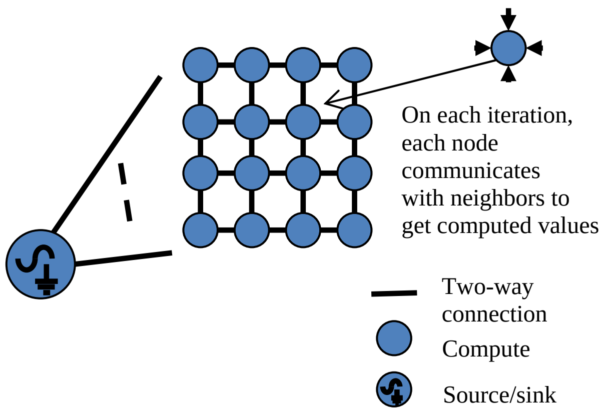
On each iteration, each node communicates with neighbors to get computed values
Two-way connection
Compute
Source/sink

Figure 4 Stencil pattern.

Figure 5 shows the all-to-all pattern. In this pattern, each slave process can communicate with any of the other slave processes, which provides the most general of communication pattern. This is also the worst case scenario for message passing but certain problems may require it. An example of this pattern is the gravitational N-body problem using the brute force $O(N^2)$ algorithm that computes the force on each body due to all the other bodies. Solving a dense system of linear equations with n equations and n unknowns may also use an all-to-all pattern.

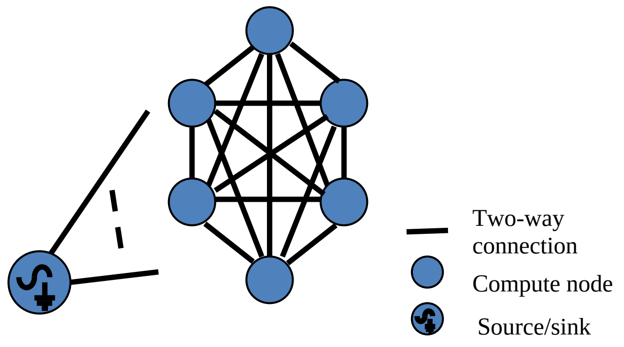

Figure 5 All-to-all pattern.

Both the N-body problem and iterative problems that converge on a solution such as solving linear equations by iteration require the all-to-all pattern to be repeated a number of times with the results of one iteration passed to all the nodes before the next iteration. Our framework implements the all-to-all pattern that includes this synchronous iteration feature. We call the pattern the CompleteSyncGraph pattern. At the end of each iteration, workers synchronize and update their data and proceed to new computations. The master node and framework will not gain control of the data flow until all the iterations have finished. A tutorial has been written to use this pattern in our framework, complete with sample code [20].

Our CompleteSyncGraph pattern is really two patterns, an all-to-all pattern and a loop iteration. We do have a pattern operator that can combine patterns. An example is shown in Figure 6. Here, the synchronous all-to-all pattern is added to a stencil pattern.

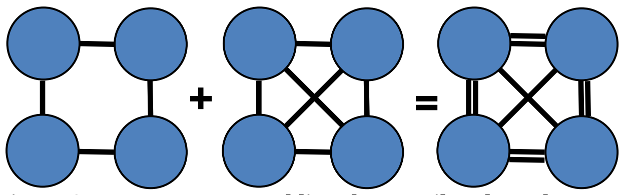

Figure 6 Pattern operator adding the stencil and synchronous all-to-all pattern.

An example use of the pattern in Figure 6 would be solving the Heat equation (which calculates static heat distribution). The stencil pattern is repeated until global termination conditions are met. Since the programmer does not have direct control over the repeating pattern, it is difficult to check for termination based upon computed values without stopping the computation completely. The added all-to-all pattern enables termination to be checked at each node. More information on using our pattern operator in this fashion can be found at [18].

3.2 User Interface

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 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.

To create and execute parallel programs, the basic layer programmer selects an existing pattern and implements three principal Java methods:

- Diffuse method – to distribute pieces of data.
- Compute method – the actual computation
- Gather method – to gather the results

The programmer also completes a "bootstrap" class to deploy and start the framework with the selected pattern. The framework self-deploys on a single multicore computer, a local cluster, or a geographically distributed platform and executes the pattern with the programmer's computation.

As an illustration, consider deploying a workpool pattern to compute π using the well-known Monte Carlo method [15]. The basis of Monte Carlo calculations is the use of random selections. To compute π, a circle is formed within and touching a 2 x 2 square so that the radius of the circle is 1. Points are chosen randomly within the square. The fraction of points that fall within the circle will converge on π/4 as the number of points increases because (area of circle)/(area of the square) = $\pi r^2/(2 \times 2)$. The code implementing the required framework interface is shown below. Points are chosen in one quadrant in the code.

```
package edu.uncc.grid.example.workpool;
... // import statements
public class MonteCarloPiModule extends Workpool {
        private static final long serialVersionUID = 1L;
        private static final int DoubleDataSize = 1000;
        double total;
        int random_samples;
        Random R;
        public MonteCarloPiModule() {
         R = new Random();
        }
        public void initializeModule(String[] args) {
         total = 0;
         random_samples = 3000; // random samples
        }
        public Data Compute (Data data) {
         DataMap<String,Object>
           input= (DataMap<String,Object>)data;
         DataMap<String, Object>
           output = new DataMap<String, Object>();
         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;
        }
        output.put("inside", inside);
        return output;
    }
    public Data DiffuseData (int segment) {
       DataMap<String, Object>
         d =new DataMap<String, Object>();
       d.put("seed", R.nextLong());
       return d; // returns a random seed for each job unit
    }
    public void GatherData (int segment, Data dat) {
       DataMap<String,Object>
         out = (DataMap<String,Object>) dat;
       Long inside = (Long) out.get("inside");
       total += inside; // aggregate answer from all the worker nodes.
       }
       public double getPi() {
         double pi = (total /(random_samples*DoubleDataSize)) * 4;
         return pi;
       }
       public int getDataCount() {
         return random_samples;
       }
    }
```

The bootstrap class to start the framework and deploy the code is given below.

```
  package edu.uncc.grid.example.workpool;
  … //import statements
  public class RunMonteCarloPiModule {
    public static void main(String[] args) {
    try {
      MonteCarloPiModule pi = new MonteCarloPiModule();
      Seeds.start( "/path-to-seeds-folder" , false);
      PipeID id = Seeds.startPattern(new Operand( (String[])null,
        new Anchor( "hostname" ,
        Types.DataFlowRoll.SINK_SOURCE), pi ) );
      System.out.println(id.toString() );
      Seeds.waitOnPattern(id);
      System.out.println( "The result is: " + pi.getPi() ) ;
      Seeds.stop();
      } catch
        … // exceptions
      }
}
```

The code is given to show the simplicity of the approach. Note that the programmer does not have to write any message passing code. Other patterns are very similar in their requirements, and template code is given on the framework home page [14] together with sample code that can be run on a single PC or a distributed platform. In our class, we ask students to use their own PCs. Tutorials are provided describing code development, compilation and execution using a command line and through the Eclipse IDE [19], [21]. Our students generally use Eclipse because syntax errors and missing classes/methods are quickly highlighted.

4. TEACHING WITH PATTERNS

We propose a new way of teaching parallel programming using the framework described above and have tried this approach at the two universities: UNC-Charlotte and UNC-Wilmington. Classroom teaching at UNC-Charlotte and UNC-Wilmington is based upon a 15-week semester. The regular parallel programming course at UNC-Charlotte is the senior undergraduate course/first year graduate course ITCS 4145/5145, which has been taught for many years. Previously, the course followed the traditional approach of teaching low-level parallel programming tools - MPI message passing programming, OpenMP and thread-based shared memory programming, and since 2010, high performance computing GPU programming using CUDA. Parallel algorithms (numerical algorithms, sorting, searching, and other applications) are covered after the programming tools but over the years fewer algorithms as the focus has been on learning the tools. Similarly, theoretical aspects have been given less prominence. Some of this is driven by the desire to provide students with programming skills using MPI, OpenMP, and CUDA. The applications that students tackle are quite limited – such as matrix multiplication, sorting, and the gravitational *N*-body problem. Students use a cluster of servers dedicated to teaching parallel and distributed computing. The course is very popular and typically closes very quickly at its maximum of 50 students. A similar class has been taught at UNC-Wilmington for many years.

For the Fall 2012 class, we substantially revised the course to start with the pattern approach. In addition, the course was co-taught between UNC-Charlotte and UNC-Wilmington on the North Carolina Research and Education Network (NCREN), a televideo network that connects most North Carolina universities and colleges. The course now focuses on higher-level computational strategies and not only uses the pattern programming approach described here but also introduces a C/C++ based compiler directive approach described elsewhere [2], [3]. We still cover MPI, OpenMP, and CUDA but after computation strategies and using patterns with our framework. Students will still need to understand how to program at a lower level but now with a design philosophy that is applicable to larger applications. Table 1 shows the outline of the re-designed course.

Table 1. Course topics

Parallel Computing	Demand for computational speed, grand challenge problems, potential speed-up using multiple processors, speed-up factor, max speed-up, Amdahl's law, Gustafson's law.
Parallel Computers	Types of parallel computers, shared memory systems, multicore, distributed memory systems, networked computers clusters, GPU systems.
Pattern Programming	Parallel patterns for structured parallel programming, workpool, pipeline, divide and conquer, stencil, all-to-all patterns, advantages of patterns, Seeds framework, user interface, programming examples
Assignment 1	Using the Seeds Pattern Programming Framework: 1 - Workpool
Lower-level message-passing computing - MPI	Message-passing programming, MPI, point-to-point message passing, message tags, MPI communicator, blocking send/recv, compiling and executing MPI programs, instrumenting code for execution time, Eclipse IDE Parallel Tools Platform. MPI collective routines, broadcast, scatter, gather, reduce, barrier, synchronous message passing, asynchronous (non-blocking) message passing, changing to synchronous message passing.
Assignment 2	Compiling and executing MPI programs. Comparison with pattern framework
Synchronous All-To-All pattern	Synchronous All-To-All pattern, gravitational *N*-body problem, Barnes-Hut algorithm, Seeds CompleteSynchGraph pattern, pattern framework code for *N*-body problem.
Divide and conquer pattern	Recursive divide and conquer pattern, example: numerical integration with adaptive quadrature.
Pipeline pattern	Pipeline pattern, examples unfolding loops, insertion sort, prime numbers, upper triangular linear equations. Seeds pipeline pattern, sorting code.
Iterative synchronous All-To-All pattern	Iterative synchronous All-To-All pattern, solving system of linear equations by iteration, Seeds CompleteSynchGraph pattern, Jacobi iteration, convergence rate.
Stencil pattern	Stencil pattern, applications, heat distribution problem, Seeds code, cellular automata, game of life, partially synchronous method.
Compiler directive approach	Introduction to Paraguin compiler, parallel region, forall, broadcast, gather, examples
Assignment 3	Using Paraguin to create MPI programs using the workpool pattern.
Programming with Shared Memory	Processes, threads, issues, interleaved statements, thread safe routines, re-ordering code, compiler/processor optimizations, accessing shared data, critical sections, locks, condition variables, deadlock, semaphores, monitors, Pthreads program example, dependency analysis (Bernstein's conditions), serializing code, cache false sharing, sequential consistency.
OpenMP	OpenMP directives/constructs, parallel, shared and local variables, work-sharing, sections, for, loop scheduling, for reduction, single master, critical, barrier, atomic, flush.
Assignment 4	Using Paraguin to create MPI programs, Sobel edge detection, and hybrid MPI/OpenMP
Data parallel pattern	Data parallel pattern, examples, data parallel prefix sum algorithm, matrix multiplication, introduction to HPC GPU systems and CUDA
Assignment 5	CUDA programs using GPU server, vector addition and heat distribution problem, with graphics.
Parallel algorithms (in various places throughout course)	Parallelizing matrix multiplication, block multiplication, recursive algorithm, mesh algorithms, Cannon's algorithm, systolic array, solving system of linear equations, direct and iterative methods, red-black, multigrid, potential speedup of sorting in parallel, compare and exchange, bubble sort, odd-even transposition sort, mergesort, quicksort, odd-even mergesort, bitonic mergesort, shearsort, rank sort, counting sort, radix sort.

As with the pre-pattern programming version of the parallel programming course, we have programming assignments (one every 2-3 weeks) to support the concepts presented in the lectures. The first programming assignment asks students to use our framework with the workpool pattern on their own PC (or a lab computer). From our previous experiences in large distributed computing courses [11], we find it much better to have students use their own computers rather than log onto centralized servers to do assignments, especially for code development, when possible to avoid servers being overloaded with a large number of concurrent processes. For Assignment 1, students have to install our framework (and Java and Eclipse if they do not have these). The basic Monte Carlo π code is given but students also have to develop code to perform matrix multiplication. The next assignment asks students to implement the same problems with the workpool pattern but using MPI. Students compare using the pattern framework with lower-level MPI coding approach.

Subsequent assignments explore more patterns and also OpenMP with MPI. Compiler directives are also explored as a higher-level approach. Finally, we cover GPUs and CUDA as in previous course offerings but only after introducing the data parallel pattern that is the basis of GPUs and CUDA.

5. CONCLUSIONS

This paper describes an approach for teaching parallel programming by first starting with higher-level computational patterns. We have developed a new software framework that enables parallel and distributed programs to be implemented and executed on a parallel or distributed platform without needing to write low-level message passing code. Advantages of patterns include:

- Reduces programmer errors
- Abstracts/hides underlying computing environment
- Reduces source code size (lines of code)
- Leads to an automated conversion into parallel programs without using low level message-passing routines.
- Hierarchical designs can be created with patterns embedded into patterns, and pattern operators to combine patterns.

Disadvantages of patterns include:

- New approach to learn
- Takes away some of the freedom from programmer
- Performance is reduced slightly

We strongly believe that our approach builds a foundation for students to tackle larger professional applications by thinking about established higher-level patterns first.

6. ACKNOWLEDGMENTS

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.

7. 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. F. 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. Retrieved December 5, 2012 from 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. n.d. Our Pattern Language (OPL): A Design Pattern Language for Engineering (Parallel) Software. Retrieved December 5, 2012 from http://parlab.eecs.berkeley.edu/wiki/_media/patterns/paraplop_g1_1.pdf.

[7] Intel. n.d. Introducing Intel Many Integrated Core Architecture. Retrieved December 5, 2012 from http://www.intel.com/technology/architecture-silicon/mic/index.htm.

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

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

[10] Microsoft. n.d. Patterns of Parallel Programming Understanding and Applying Parallel Patterns with the .Net Framework 4. Retrieved December 5, 2012 from http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=19222

[11] Wilkinson, B., and Ferner, C. 2008. Towards a Top-Down Approach to Teaching an Undergraduate Grid Computing Course. *SIGCSE 2008 Technical Symposium on Computer Science Education*. Portland, Oregon.

[12] Wikipedia. n.d. Software Design Patterns. Retrieved December 5, 2012 from http://en.wikipedia.org/wiki/Design_pattern_(computer_science)

[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. n.d. Parallel Grid Application Framework. Retrieved December 5, 2012 from http://coit-grid01.uncc.edu/seeds/

[15] Villalobos, J. and Adibolo, Y. K. 2012. Seeds Framework Workpool Template Tutorial. Retrieved December 5, 2012 from http://coitweb.uncc.edu/~abw/seeds/docs/WorkpoolTutorial.pdf

[16] Villalobos, J. n.d. Seeds Framework Pipeline Template Tutorial. Retrieved December 5, 2012 from http://coit-grid01.uncc.edu/seeds/docs/pipeline_tutorial.pdf

[17] Villalobos, J. n.d. Seeds Framework Stencil Template Tutorial. Retrieved December 5, 2012 from http://coit-grid01.uncc.edu/seeds/docs/stencil.pdf

[18] Villalobos, J. n.d. Seeds Framework Pattern Operator Template Tutorial. Retrieved December 5, 2012 from http://coit-grid01.uncc.edu/seeds/docs/PatternOperator.pdf

[19] Villalobos, J. n.d. Seeds Development with Eclipse. Retrieved December 5, 2012 from http://coit-grid01.uncc.edu/seeds/docs/seeds_eclipse_tutorial.pdf

[20] Villalobos, J., and Adibolo, Y. K. 2012. Seeds Framework. The CompleteSynchGraph Template Tutorial. Retrieved December 5, 2012 from http://coitweb.uncc.edu/~abw/seeds/docs/CompleteSynchGraphTutorial.pdf

[21] Villalobos, J., and Adibolo, Y. K. 2012. Getting Started With the Seeds Framework: An Introduction to the Seeds Framework in a Shared Memory Environment. Retrieved December 5, 2012 from http://coitweb.uncc.edu/~abw/seeds/docs/GettingStartedSeeds.pdf