Patterns and Exemplars: Compelling Strategies for Teaching Parallel and Distributed Computing to CS Undergraduates

Joel Adams
Calvin College
adams@calvin.edu

Richard Brown
St. Olaf College
rab@stolaf.edu

Elizabeth Shoop
Macalester College
shoop@macalester.edu

Abstract—**Parallel programming patterns provide enduring principles that serve as a conceptual framework to orient students when they set out to solve problems. Learning patterns enables students to quickly gain the intellectual and coding skills they will need to embrace the future of parallel and distributed computing (PDC). Exemplars consisting of representative and compelling applied problems, together with implementations in different parallel technologies, constitute a valuable resource for learning and teaching. Parallel programming patterns and exemplar applications naturally complement each other, and together provide a unified and practical strategy for PDC education at multiple course levels. We present two strategies we have used for incorporating patterns into undergraduate CS courses, examine the potential of exemplars, and indicate how patterns and exemplars reinforce each other.**

Keywords-parallel and distributed computing; CS education; parallel design patterns; exemplars; applications

I. Introduction

While the need for undergraduate computer science students to learn parallel and distributed computing (PDC) has become indisputable, efforts to meet that need face some significant pedagogical challenges. Which new knowledge units must be added to undergraduate computer science curricula? How can those topics best be incorporated into course offerings? What pedagogical strategies will enable students to learn that PDC content [1]?

The combined efforts of individual universities and colleges and of multi-institution initiatives have fostered significant progress on these challenges in recent years. In particular, the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing, the ACM/IEEE CS2013 Joint Task Force, and other entities have developed draft curriculum recommendations, itemizing and categorizing the PDC body of knowledge for today's undergraduates [2]–[4]. At EDUPAR and other venues, TCPP Early Adopters and others have published approaches to incorporating PDC topics into curricula [5]. Our own efforts in this direction have advocated a systematic, modular strategy for incrementally adding hands-on PDC learning experiences to existing courses at all undergraduate levels, consistent with the approach recommended in the first version of the TCPP curriculum [6]. We argue that an incremental, modular strategy can describe not only what to teach and how to teach it, but also how to realistically incorporate this now essential content into existing, already crowded course syllabi, by providing materials that are appropriate and relevant to those existing courses [7],[8].

Curricular recommendations serve an invaluable purpose as concrete benchmarks for assessing the breadth and depth of content coverage in an academic curriculum. The specification of Bloom levels in the NSF/TCPP Curriculum provides more nuanced guidance to professors and curriculum designers. But curriculum recommendations can only capture a snapshot in time of a dynamically evolving body of knowledge. For PDC, the evolutionary unfolding of curricular content is particularly volatile at present. For example, in parallel architectures, specialized computational cores for linear algebra and vector computations are leaping from GPGPU cards directly onto the same silicon chips hosting CPU cores (e.g. AMD Fusion, Intel Xeon), and a recently released processor with dozens of CPU cores behaves largely like a cluster on a chip [9]. In software, new developments in parallel programming languages (e.g., Chapel, Go) and an urgent search for effective and productive PDC programming models such as map-reduce [10] follow the now-ubiquitous multicore architectures and the growth of distributed cloud computation in industry. The PDC curricular recommendations a decade from now may differ as radically from today's curricular recommendations as the PDC aspects of CS2013 differ from those of CS2001.

These exciting developments raise two questions:

1. Apart from parallel algorithms, *are there principles that we can teach our students in the context of today's PDC hardware and software milieu that will remain relevant and abiding for the unforeseeable technologies of the next decade?*
2. Since all computing students must now learn about PDC and many of them may not be interested in PDC, *how can we motivate today's computing students to engage with this rapidly evolving field of parallel and distributed computing*?

These big-picture questions of enduring principle and compelling motivation can now come to the fore as the community creates strategies to implement curricular recommendations and assesses their effectiveness.

We propose a two-pronged approach to addressing these issues. Our first "prong" is *parallel programming patterns*, which emerge directly from the experience of professional practitioners. These offer a practical and relevant collection of problem-solving strategies that transcend particular technologies. Patterns provide a fast track for students to acquire enduring intellectual and useful skills in PDC by preparing them for current and emerging hardware and software systems. Moreover, this pattern-based problem solving approach provides the "parallel thinking" skills we seek for all of our computing students.

Our second "prong" is *exemplar applications* that apply PDC topics and techniques in solving some (more or less) realistic problem. Such exemplars provide convenient and useful resources for presenting and comparing parallel and distributed computing technologies. They also motivate students to learn PDC principles and practices by helping them see how PDC makes an impact in other fields. The combination of patterns and exemplars provides a unified approach for motivating undergraduates to develop parallel thinking, often with modest or even negligible added costs to existing course syllabi.

This paper discusses the value of parallel programming patterns and exemplar applications in computing curricula, and our own experiences incorporating these pedagogical strategies into undergraduate CS courses at multiple levels.

II. Design Patterns

Most programmers know the notion of a software *design pattern* from the "gang of four" book [11]. A good design pattern captures the essence of a coding strategy a programmer can use to solve a class of programming problems. A programmer must think at various levels when problem solving, from architectural frameworks, to high levels that are near the domain of the problem being solved, to low levels near the hardware on which the application will run. Design patterns capture reusable programming strategies at all levels. Given the challenging nature of parallel and distributed programming, it is natural to seek design patterns for these technologies.

A. *History of Parallel Design Patterns*

Design patterns began in the 1970's with the work of Christopher Alexander, a (building) architect who observed that expert architects could judge the quality of a design by looking at it. This led him to a series of books in which he described the design patterns that master architects naturally (often intuitively) use to design buildings, and proposed a *pattern language* for communicating about and applying such patterns when designing a building [12]. In this sense, a "pattern language" means more than a catalog of design patterns that provides names and descriptions, because a pattern language also describes a methodology for using those patterns to accomplish a quality building design.

Following [11], Lea published a book on patterns for concurrency in Java [13], which was highly influential in shaping the understanding of how to apply patterns to parallel programming. More recently, Mattson, Saunders, and Massingil documented an extensive collection of parallel programming patterns [14]. Inspired by Alexander [12], this effort provided a "pattern language", including both a methodology and a catalog of design patterns, called the Pattern Language of Parallel Programming (PLPP).

The PLPP team now collaborates with Kurt Kreutzer and the community of researchers at UC Berkeley's Parlab on a new pattern language called OPL (for 'Our Pattern Language') [15]–[17]. OPL combines the PLPP work on parallel algorithms with higher level structural patterns (largely drawn from [18]) and computational patterns drawn from [19],[20]. OPL seeks to address the software architecture of an entire application, including parallel and sequential aspects, by providing (i) patterns organized into a hierarchy of levels, and (ii) a pattern-based methodology for developing the application at each of those levels.

It is important to note that the term *language* as used in PLPP and OPL does *not* refer to programming languages. Instead, that term "language" derives from its use by Alexander and refers to having a way to communicate with patterns in architectural problem solving. In the OPL and PLPP systems, "language" means a *structured catalog of design patterns, together with a design methodology based on a web of connected patterns*.

On the other hand, patterns of parallel programming do appear in programming languages or their extensions. For example, in languages such as Erlang and Go, processes or threads communicate using the Message-Passing pattern. The OpenMP library for the C, C++, and Fortran languages provides 'pragmas' that implement several common parallel programming patterns. At a higher level of system design, frameworks such as Rails and Django conveniently implement the Model-View-Control pattern for web applications.

Among other initiatives to develop parallel patterns, Johnson and collaborators have been active in both refining the notions of patterns [21] and in describing the use of patterns with .NET [22]. The pragmatic applications of patterns in the latter book focus on teaching software engineers to become productive parallel programmers quickly. Taken together, these developments indicate how the parallel patterns movement is (i) helping programmers learn about PDC, (ii) aiding professionals increase their productivity, and (iii) inspiring the evolution of new software design methodologies.

B. *The OPL Patterns*

The OPL pattern site [17] and the Illinois pattern catalog [21] have many patterns in common, and both choose to organize the patterns into layers, ranging from high-level software architecture patterns down to small patterns used to solve smaller problems on particular hardware. Though these are a work in progress, the current patterns present an intellectual core that can provide a useful framework for both practitioners and educators. In the next section, we briefly describe the OPL layers and patterns that we have found useful as an organizing principle for PDC curriculum design.

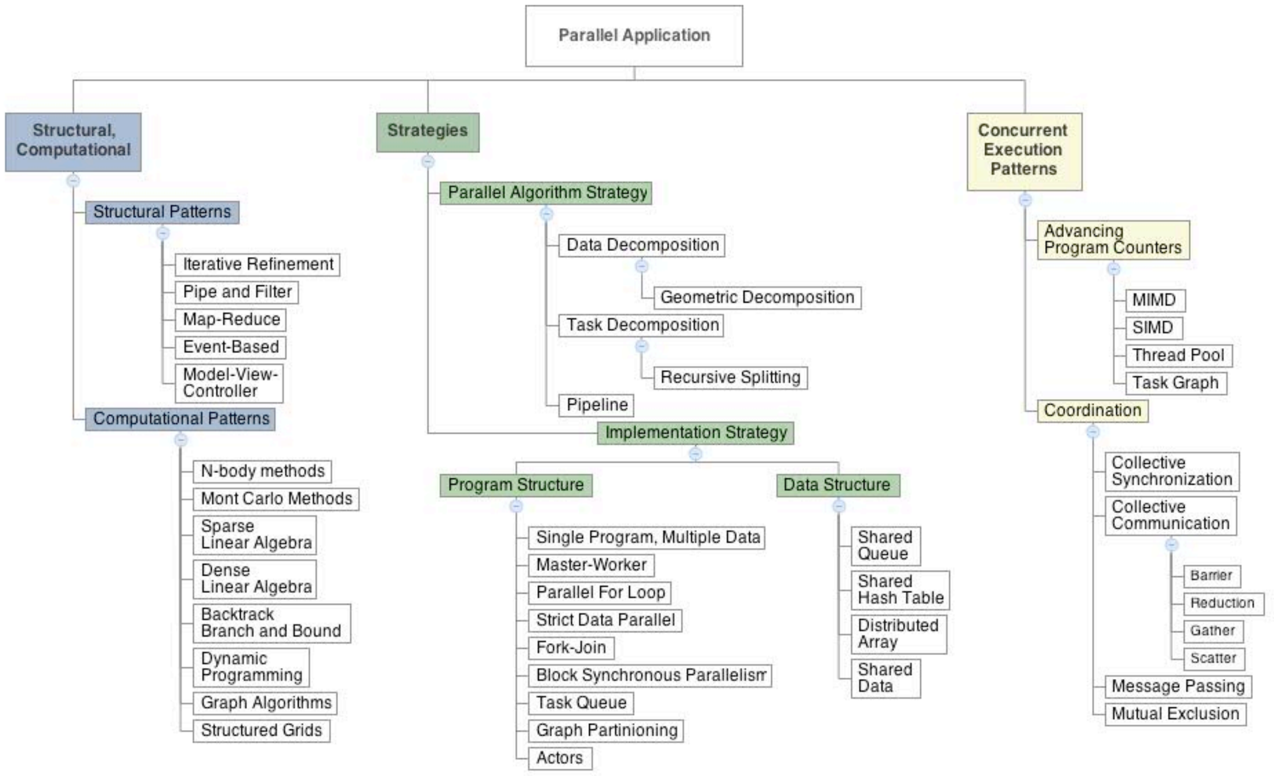

Figure 1. Parallel patterns as organized on http://parlab.eecs.berkeley.edu/wiki/patterns/patterns. Here we show a subset most likely to be found in an undergraduate curriculum. Some pattern names have been changed for simplicity of presentation to undergraduates.

Figure 1 shows a subset of the OPL patterns organized into three primary categories (or layers):

1) *Structural/Computational,*
2) *Strategies, and*
3) *Concurrent Execution.*

Some parallel applications will use a structural pattern (blue, upper left in Figure 1) by using a large software framework for carrying out computations in parallel. Popular examples of these frameworks include Google's MapReduce system [10], its open-source counterpart Hadoop [23], and application development frameworks for implementing applications using the Model-View-Controller pattern. Some parallel applications may use a high-level computational pattern (blue, lower left in Figure 1). These patterns come from well-developed methods used in high-performance computing applications. Some applications may simply employ an algorithm strategy and be implemented using program structure strategies and data structure strategies (indicated in green in the middle of Figure 1). Any parallel application will use some set of concurrent execution patterns based on the hardware it will run on (indicated in beige on the right in Figure 1).

The parallel algorithm strategy patterns that a programmer may employ are at a higher level of abstraction than the implementation strategies. The Data Decomposition pattern choice involves deciding how the data that the parallel tasks needs will be distributed among them. The Task Decomposition pattern involves deciding how computations will be distributed among processing units. The programmer will choose program structure and data structure implementation strategies for portions of the problem to be solved. These choices need to match the Concurrent Execution patterns for the hardware and software libraries the programmer is using to build the application.

Some of the Strategy and Concurrent Execution patterns occur so frequently in practice that programming language libraries include them. For example, programs that use the Message Passing Interface (MPI) library are inherently Single-Program, Multiple-Data, use Message Passing, and have built-in collective synchronization constructs (e.g. Broadcast, Reduction, Scatter, and Gather). Programs using the OpenMP library for shared-memory systems inherently use the Fork-Join and Thread Pool patterns, and have high-level constructs for the Parallel For Loop, Mutual Exclusion, and Reduction patterns.

C. *Patterns in Educational Practice*

We have found the diagram in Figure 1 to be a useful tool to (i) introduce students to the patterns, and (ii) describe the coding patterns in the context of solving a problem on a particular type of hardware. We next describe how we introduce the patterns, and then describe our use of patterns in a course devoted to parallel programming.

One way to introduce students to parallel patterns is through the use of *patternlets* -- fully operational but minimalist programs that illustrate the pattern's use and behavior in a given parallel platform. Just as "hello world" is frequently used to introduce the syntax of a fully functional but minimalist program, a patternlet can be used to introduce a pattern's expression in a given parallel platform. To illustrate, suppose we want to introduce students to the Master-Worker pattern. Figure 2 presents a patternlet for this pattern in C using OpenMP.

```
// ompMasterWorker.c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv) {

   //#pragma omp parallel
   {
     int id = omp_get_thread_num();
     int numWorkers = omp_get_num_threads();

     if ( id == 0 ) {  // thread 0 is the master
       printf("Greetings from the master, # %d of %d threads\n",
            id, numWorkers);
     } else {          // threads with ids > 0 are workers
       printf("Greetings from a worker, # %d of %d threads\n",
            id, numWorkers);
     }
   }
   return 0;
}
```

Figure 2. The master-worker pattern in C and OpenMP

When compiled and run on a computer with a hyperthreaded quad core processor, the program in Figure 2 produces the following output:

```
Greetings from the master, # 0 of 1 threads
```

However, if the OpenMP `#pragma` directive is uncommented, the program is recompiled and rerun on the same computer, then output like the following is produced:

```
Greetings from a worker, # 7 of 8 threads
Greetings from a worker, # 1 of 8 threads
Greetings from a worker, # 2 of 8 threads
Greetings from a worker, # 6 of 8 threads
Greetings from a worker, # 4 of 8 threads
Greetings from a worker, # 3 of 8 threads
Greetings from the master, # 0 of 8 threads
Greetings from a worker, # 5 of 8 threads
```

When uncommenting a single line triggers such a large change in program behavior, students will naturally be curious as to what is happening. The exercise thus leverages a student's curiosity to motivate learning, and the patternlet's minimalist structure strives to make it as easy as possible for a student to trace and understand the program's execution.

Once a student has seen the pattern in one language or platform, the same pattern can be presented in a different context. For example, Figure 3 presents a patternlet for the same pattern in C using MPI.

```
// mpiMasterWorker.c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
   int id = -1, numWorkers = -1, length = -1;
   char hostName[MPI_MAX_PROCESSOR_NAME];

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &id);
   MPI_Comm_size(MPI_COMM_WORLD, &numWorkers);
   MPI_Get_processor_name (hostName, &length);

   if ( id == 0 ) {  // process 0 is the master
     printf("Greetings from the master, \
            # %d (%s) of %d processes\n",
            id, hostName, numWorkers);
   } else {          // processes with ids > 0 are workers
     printf("Greetings from a worker, \
            # %d (%s) of %d processes\n",
            id, hostName, numWorkers);
   }

   MPI_Finalize();
   return 0;
}
```

Figure 3. The master-worker pattern in C and MPI

Unlike OpenMP, MPI computations can run across distributed multiprocessors, so our patternlet computes and displays the name of the host on which the process is running. This extra information can be very helpful when students are seeing distributed computing for the first time.

When students have seen the same pattern in at least two different contexts, they can then appreciate a general form for the pattern, such as this for the master-worker pattern:

```
id = who am I?
numWorkers = how many of us are there?
if ( id == 0 ) {
  // put master code here: e.g., doMaster(numWorkers);
} else {
  // put worker code here: e.g., doWorker(id, numWorkers);
}
```

This code also illustrates the Single Program, Multiple Data pattern, in which one program describes how different sets of threads/processes work on different data elements, as determined by the thread/process id.

The advantages of using patternlets to introduce students to patterns include:

- Since patternlets are *fully operational*, instructors can use them "live" to illustrate their lectures, or students can explore them within hands-on exercises. In either case, the patternlet's behavior can be easily dissected to help students understand the pattern's behavior.
- Since patternlets are *minimalist*, they allow a student to quickly grasp the essence of the pattern, without becoming bogged down in extraneous or distracting details, or having to learn a new problem.

Once a student has been introduced to a pattern via a patternlet, then as an immediate followup, that pattern should be used to solve a "real" problem (such as the exemplars we discuss in section III), so that the student sees the pattern's use in a less trivial context. By seeing a pattern used in multiple contexts, a student will gain a deeper appreciation of its usefulness.

Patternlets are a work in progress. At this time, we have developed twenty-one of them (twelve in OpenMP and nine in MPI). These are freely available from CSinParallel.org for instructors or students to download and use (see serc.carleton.edu/csinparallel/modules/patternlets.html).

In a course devoted to programming parallel applications, Figure 1 serves as an organizing framework when discussing solutions to problems, and patternlets form the basic introduction. Our preference in such a course is to include several hardware configurations, ranging from shared-memory machines with a small number of cores to a cluster of many machines. When using a particular hardware platform, we can point to the diagram and describe which concurrent execution patterns on the right are pertinent. For example, on a cluster of computers using MPI, the concurrent execution patterns available would be MIMD with Collective Synchronization, Collective Communication, and Message Passing, whereas a shared-memory multicore machine using the OpenMP library would be MIMD with a Thread Pool, and problem solutions that involve modifying shared data will require Mutual Exclusion.

Once the hardware configuration for a problem solution is established for the students, we can use Figure 1 to discuss the types of patterns employed in the examples we provide (see exemplars below), and suggest patterns for the new problems for which they must develop solutions. We can also point out patterns that are used in textbook examples. The patterns become clearer to the students if we identify them in the diagram each time we use them in a program.

III. EXEMPLARS

We will use the term *exemplar* to describe a representative applied problem together with a collection of illustrative computational solutions to that problem. Here, we follow the lead of the Educational Alliance for a Parallel Future (EAPF), which is developing the EAPF Practicum, a collection of exemplars involving multiple approaches to parallel and distributed computation. The EAPF Practicum is an invaluable resource both for individuals seeking to develop their PDC knowledge and skills, and for educators. By providing solutions to the same problem in using differing tools and technologies, a reader can obtain working example code, compare differences and similarities between different software and hardware platforms, and begin developing a sense of how different PDC technologies can be applied to a particular problem.

Exemplars are also interesting for their potential value in motivation. Naturally, many CS students pursue the study of computing because they like the subject for itself, and some of those students will find themselves drawn to PDC topics. However, every computing graduate must now possess a foundation in PDC, due to the omnipresent multicore architectures, plus the commercial growth of distributed computing. This required exposure to PDC may be a challenge if students do not have natural interests in those topics. Exemplars, with their basis in computational problems in other domains, can help to increase the interest in PDC through useful and relevant applications.

It is well established that domain applications increase the interest of women and underrepresented groups. For example, Margolis and Fisher found that more women than men linked their interest in CS to other areas such as medicine, the arts, space exploration, etc. They also found that early experiences that situate technology in realistic settings, plus curricula that exploit the connections between CS and other disciplines, contributed to an increase of participation and retention of women in computing [24], [25]. Anecdotally, we find that applications to other domains attract students of diverse backgrounds, both men and women, including many with interests in technology for its own sake. Thus, we believe that compelling exemplars in PDC will help motivate all computing undergraduates to learn these now-essential topics.

A focus on applications can often be adopted with minimal impact on a course syllabus. Simply providing applied contexts for in-class examples and course assignments helps to convey a message about relevance of computing. Furthermore, resources are now emerging for introducing PDC topics and technologies in the context of exemplar applications [26], [27].

The EAPF Practicum provides two kinds of exemplars:

- "Method exemplars" take a general problem, such as *Finding the area under a curve -- Numerical Integration*, and present solutions to it using a large number of PDC technologies.
- "Mini-application exemplars" feature more specific problems, such as *Drug Design* or *The Game of Akari*, and solve it using fewer PDC technologies.

At this writing, the numerical integration exemplar [29] presents nine implementations that perform numerical integration using a straightforward Riemann sum approach, and use it compute the value of π. Starting with a sequential computation for comparison, the site also provides shared-memory solutions (OpenMP, TBB, pthreads, Windows threads, Go language), a distributed computing solution (MPI), and two other alternatives (CUDA, Intel's Array Building Blocks). Each solution includes explanatory text accompanied by working code. The code for all exemplars resides in a Google Code repository, together with support items such as Makefiles for building the code.

The OpenMP implementation for this exemplar is listed in Figure 4. It can be used in different ways, including:

- The program and explanation on the Practicum website can provide a starting point for experienced students and professionals who need only a concrete example to begin exploring OpenMP, especially if they also have access to technical references on parallel computing.
- For less experienced students, this code can form the basis for short teaching module, such as [29].
- Comparisons may be drawn between this solution and a solution to the same problem using a different parallel technology, such as the pthreads solution in Figure 5.
- For another type of comparison, one could consider this OpenMP code for calculating π using numerical integration together with the OpenMP code solution in a different method exemplar in the collection, such as one that computes π using a Monte Carlo approach.

The mini-application exemplars in the Practicum offer more substantial illustrations of the usefulness and value of computing in other fields. For instance, the *Drug Design* exemplar [30] provides an introduction to the docking problem for computationally scoring candidate drugs (*ligands*) to protein sequences. The text presents a sequential solution to the computation based on the Map-Reduce pattern, using a scalable computational load instead of an actual molecular dynamics algorithm. Parallel implementations (all adopting the map-reduce approach) include OpenMP, C++-11 (Boost) threads, the Go language, and Hadoop. Each implementation is accompanied by substantial discussion and exercises for further exploration.

The more realistic and compelling the application, the more value students will perceive in learning the parallel technologies that can solve those problems more quickly. Many other sources exist for application examples that can motivate students to study parallel and distributed computing. For example, the Undergraduate Petascale Education Program's modules include 30 fully developed application examples of PDC, including graph-theoretic computations in epidemiology, n-body simulations, carbon nanotubes, and scientific visualization in CUDA [27].

IV. Combining Patterns and Exemplars

Parallel programming patterns, which capture reusable programming strategies at levels ranging from near the hardware to abstracted far above it, provide problem-solving insight. Exemplars supply motivation for mastering PDC in terms of the value of parallel computation for addressing real-world problems; and having multiple implementations for a particular exemplar provides insight into particular parallel technologies. Patterns and exemplars together form a potent and dynamic team for parallel computing education. We will briefly suggest some of the possibilities by considering patterns that appear in an exemplar.

```
/* Estimate pi as twice the area under a semicircle
  Command-line arguments detail omitted below:
   1. first command line arg is integer number of rectangles to use
   2. if two command-line args, second arg is number of OpenMP
threads
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/* parameters that may be overridden on the command-line */
long n_rect = 10485760;      /* default number of rectangles */
int threadct = 8;        /* default number of OpenMP threads to use */

int main(int argc, char** argv) {
 double a = -1.0, b = 1.0;  /* lower and upper interval endpoints */
 double h;                  /* width of a rectangle subinterval */
 double f(double x);        /* declare function that defines curve */
 double sum;                /* accumulates areas all rectangles so far */
 long i;                    /* loop control */
 /* parse command-line arguments, if any */
 // …

 h = (b - a)/n_rect;

 /* compute the estimate */
 sum = 0;
 #pragma omp parallel for num_threads(threadct) \
   shared (a, n_rect, h) private(i) reduction(+: sum)
 for (i = 0; i < n_rect; i++) {
  sum += f(a + (i+0.5)*h) * h;
 }

 printf("With n = %d rectangles and %d threads, ", n_rect,
threadct);
 printf("the estimate of pi is %.20g\n", sum);
 return 0;
}
double f(double x) {
 return 2*sqrt(1-x*x);
}
```

Figure 4. C OpenMP Exemplar, pi_area_omp.c

A. *Patterns in the Numerical Integration Examplar*

The example programs compute the area of a semicircle as an approximation for pi exhibit a number of parallel design patterns. As an illustration, we will list the patterns present in the OpenMP example of Figure 4.

Parallel algorithm strategy patterns. Like most implementations of this problem, pi_area_omp.c in Figure 4 exhibits the Geometric Decomposition parallel-algorithm strategy pattern. The OpenMP system parallelizes a for loop by dividing its loop-control range into subranges. We can view each subrange as a rectangle-based geometric approximation of some portion of the semicircle. In fact, we use the pattern name Geometric Decomposition for *any* decomposition into "chunks" of data, such as contiguous subranges of a loop-control range, even if the resulting computation does not correspond to a geometric figure.

Implementation strategy patterns. Two types of patterns appear at this level: *program structure* patterns focus on organizing a program, and *data structure* patterns describe common data structures specific to parallel programming. Two such patterns appear in this example:

- The Parallel For Loop Program Structure pattern, because the OpenMP pragma requests that the summation loop be carried out in parallel, using OpenMP's automatic mechanism for carrying out subranges of the summation in parallel.
- The Fork-Join Program Structure Pattern, because the *parallel* portion of an OpenMP pragma implicitly forks a set of threads and subsequently joins them when the loop is completed.

Concurrent execution patterns. This level consists of *advancing program counters* patterns, which concern the timing relationships for executing instructions among different threads or processes, and *coordination* patterns, which provide mechanisms for processes or threads to correctly access the data they need (cf. interprocess communication). The pi_area_omp.c example contains the following patterns:

- The Thread Pool advancing-program-counter pattern appears by default because it is built into OpenMP.
- OpenMP's *reduction(+:sum)* operation implements the Collective Communication group's Reduction pattern.

B. The Numerical Integration Examplar in Pthreads

Figure 4 solves the numerical integration problem using OpenMP's implicit multithreading mechanism. For comparison purposes, Figure 5 solves the same problem, but does so using Posix threads. This ability to directly compare different but equivalent solutions to the same problem is invaluable in helping students appreciate subtle concepts, such as implicit vs. explicit multithreading.

V. Conclusion

Given the increasing importance of parallel and distributed computing, CS educators naturally seek a body of principles to provide a conceptual framework for their students, not only to organize the many aspects of parallelism, but also as a guide to parallel computing best practices. We argue that parallel programming patterns satisfy that search: since a good design pattern captures the essence of an experienced programmer's thinking when solving a programming problem, design patterns embody much of the "parallel thinking" we want our students to carry with them as they enter their careers. Hardware and software technologies will come and go in years to come, but patterns will remain relevant throughout those changes.

"Exemplars" that include multiple implementations of solutions to a realistic problem, often from a non-CS domain, are a useful resource for learning and teaching about PDC. They also help motivate students who may or may not have an intrinsic interest in the topics and techniques of

```
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#define NUM_RECT 10000000
#define NUMTHREADS 4
double gPi = 0.0; // global accumulator for areas computed
pthread_mutex_t gLock;

void *Area(void *pArg){
    int myNum = *((int *)pArg);
    double h = 2.0 / NUM_RECT;
    double partialSum = 0.0, x; // local to each thread

    // use every NUMTHREADS-th step
    for (int i = myNum; i < NUM_RECT; i+=NUMTHREADS) {
      x = -1 + (i + 0.5f) * h;
      partialSum += sqrt(1.0 - x*x) * h;
    }
    pthread_mutex_lock(&gLock);
    gPi += partialSum; // add partial to global final answer
    pthread_mutex_unlock(&gLock);
    return 0;
}

int main(int argc, char **argv) {
    pthread_t tHandles[NUMTHREADS];
    int tNum[NUMTHREADS], i;
    pthread_mutex_init(&gLock, NULL);
    for ( i = 0; i < NUMTHREADS; ++i ) {
      tNum[i] = i;
      pthread_create(&tHandles[i],  // Returned Thread handle
        NULL,                       // Thread Attributes
        Area,                       // Thread function
        (void)&tNum[i])             // Data for Area()
   }
   for ( i = 0; i < NUMTHREADS; ++i ) {
      pthread_join(tHandles[i], NULL);
   }
    gPi *= 2.0;
    printf("Computed value of Pi: %12.9f\n", gPi );
    pthread_mutex_destroy(&gLock);
    return 0;
}
```

Figure 5. C Pthreads solution to the numerical integration exemplar.

parallelism, especially when those applications are realistic and involve compelling concerns. Such exemplars should include sequential and parallel solutions, so that students can directly experience the benefit that parallelism provides.

The similarity between the patternlet resources and the exemplar resources, each focusing largely on compact coding examples, suggests a unified approach for developing, extending, enlivening, and inspiring students about PDC topics. Indeed, patterns and exemplars naturally complement and enhance each other in a CS curriculum, since implementation code for exemplars yield examples of patterns, and since pattern-thinking helps to guide the coding problem solutions.

Finally, we note that our recommendations from experience can be incorporated into a CS curriculum with only modest impact on course syllabi, and can be included in courses at any undergraduate level. For example, the structured reference diagram of patterns in Fig. 1 enables a professor quickly to identify and contextualize patterns that are present in a parallel coding example; patternlets and exemplars are effective example-driven pedagogical tools for presenting PDC, and need not be time-consuming additions to a syllabus. Choosing an intriguing applied context for presenting CS topics requires little or no more time than using a generic example, as long as exemplar resources are available that incorporate meaningful domain applications. The EAPF Practicum and CSinParallel.org provide such resources for computing educators.

ACKNOWLEDGMENTS

We have benefited from the contributions of the parallel patterns community, particularly Tim Mattson of Intel Corporation, who has been a great personal support in our emerging efforts. We are indebted to the EAPF, led by Matt Wolf of Georgia Tech and Paul Steinberg of Intel, and to the numerous EAPF members who have generously contributed to the Practicum and its exemplars.

REFERENCES

[1] R. Brown, E. Shoop, J. Adams, C. Clifton, M. Garnder, M. Haupt, and P. Hinsbeeck, "Strategies for Preparing Computer Science Students for the Multicore World," in *Proceedings of 15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, 2010, pp. 99-115.

[2] S. K. Prasad, A. Chtchelkanova, S. Das, F. Dehne, M. Gouda, A. Gupta, J. Jaja, K. Kant, A. La Salle, R. LeBlanc, M. Lumsdaine, D. Padua, M. Parashar, V. Prasanna, Y. Robert, A. Rosenberg, S. Sahni, B. Shirazi, A. Sussman, C. Weems, and J. Wu, "NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing: core topics for undergraduates," in *Proceedings of the 42nd ACM technical symposium on Computer science education*, New York, NY, USA, 2011, pp. 617–618.

[3] "NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing – Core Topics for Undergraduates." [Online]. Available: http://www.cs.gsu.edu/~tcpp/curriculum/. [Accessed: 19-Jan-2013].

[4] ACM/IEEE-CS Joint Task Force, "Computer Science Curricula 2013." [Online]. Available: http://ai.stanford.edu/users/sahami/CS2013/. [Accessed: 19-Jan-2013].

[5] NSF/IEEE-TCPP Curriculum Initiative, "TECHNICAL PROGRAM and ONLINE PROCEEDINGS of EduPar-12." [Online]. Available: http://cs.gsu.edu/~tcpp/curriculum/?q=advanced-technical-program. [Accessed: 19-Jan-2013].

[6] R. Brown and E. Shoop, "CSinParallel and Synergy for Rapid Incremental Addition of PDC Into CS Curricula," in *IEEE 26th International Parallel and Distributed Processing Symposium, EduPar Workshop*, 2012, pp. 1329–1334.

[7] R. Brown and E. Shoop, "Modules in Community: Injecting More Parallelism Into Computer Science Curricula," in *Proceedings of The 42nd ACM Technical Symposium on Computer Science Education*, 2011, p. to appear.

[8] R. Brown and E. Shoop, "CSInParallel: Parallel Computing in the Computer Science Curriculum," 01-Jun-2010. [Online]. Available: http://csinparallel.org. [Accessed: 10-Jun-2011].

[9] J. Burt, "Intel Wraps Xeon Phi Branding Around MIC Coprocessors." [Online]. Available: http://www.eweek.com/c/a/IT-Infrastructure/Intel-Wraps-Xeon-Phi-Branding-Around-MIC-CoProcessors-655038/. [Accessed: 14-Jan-2013].

[10] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, 2004, pp. 137–150.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.

[12] C. Alexander, S. Ishikawa, and M. Silverstein, *A pattern language : towns, buildings, construction*. New York: Oxford University Press, 1977.

[13] Douglas Lea, *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Pub (Sd), 1996.

[14] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for parallel programming*. Boston: Addison-Wesley, 2005.

[15] K. Keutzer and T. Mattson, "Our Pattern Language (OPL): A design pattern language for engineering (parallel) software," in *ParaPLoP Workshop on Parallel Programming Patterns*, 2009.

[16] K. Keutzer, B. L. Massingill, T. G. Mattson, and B. A. Sanders, "A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, New York, NY, USA, 2010, pp. 9:1–9:8.

[17] OPL Working group, "A Pattern Language for Parallel Programming ver2.0." [Online]. Available: http://parlab.eecs.berkeley.edu/wiki/patterns/patterns. [Accessed: 19-Jan-2013].

[18] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[19] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[20] K. Asanovic, B. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech Report UCB/EECS-2006-183, Dec. 2006.

[21] R. Johnson, N. Chen, S. Tasharofi, and F. Kjolstad, "Parallel Programming Patterns," 2010. [Online]. Available: https://wiki.engr.illinois.edu/display/ppp/Home. [Accessed: 19-Jan-2013].

[22] C. Campbell, R. Johnson, A. Miller, and S. Toub, *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures*, 1st ed. Microsoft Press, 2010.

[23] "Apache Hadoop Project." [Online]. Available: http://hadoop.apache.org/. [Accessed: 28-Jun-2010].

[24] J. Margolis and A. Fisher, *Unlocking the Clubhouse: Women in Computing*. The MIT Press, 2003.

[25] A. Fisher and J. Margolis, "Unlocking the clubhouse: the Carnegie Mellon experience," *SIGCSE Bull.*, vol. 34, no. 2, pp. 79–83, Jun. 2002.

[26] EAPF, "Parallelism: Practice and Experience." [Online]. Available: http://cercs-ed.gatech.edu/parallel_practicum. [Accessed: 22-Jan-2013].

[27] Shodor Foundation, "Petascale : Undergraduate Petascale Modules." [Online]. Available: http://shodor.org/petascale/materials/modules/. [Accessed: 22-Jan-2013].

[28] EAPF, "Finding the area under a curve -- Numerical Integration." [Online]. Available: http://cercs-ed.gatech.edu/node/14. [Accessed: 22-Jan-2013].

[29] R. Brown and E. Shoop, "CSinParallel Module: Multicore Programming with OpenMP." [Online]. Available: http://serc.carleton.edu/csinparallel/modules/mtl_openmp.html. [Accessed: 22-Jan-2013].

[30] EAPF, "Drug Design ini-Application Example." [Online]. Available: http://cercs-ed.gatech.edu/node/21. [Accessed: 22-Jan-2013].