1-1-2015

A thread safe graphics library for visualizing parallelism

Joel C. Adams
Calvin University

Patrick A. Crain
Calvin University

Mark B. Van Der Stel
Calvin University

Patternlets

A Teaching Tool for Introducing Students to Parallel Design Patterns

Joel C. Adams

Department of Computer Science

Calvin College

Grand Rapids, MI, USA

adams@calvin.edu

(616) 526-8666

Abstract

Thanks to the ubiquity of multicore processors, today's CS students must be introduced to parallel computing or they will be ill prepared as modern software developers. Professional developers of parallel software think in terms of *parallel design patterns*, which are markedly different from traditional (sequential) design patterns. It follows that the more we can teach students to think in terms of parallel patterns, the more their thinking will resemble that of parallel software professionals. In this paper, we present *patternlets* – minimalist, scalable, syntactically correct programs, each designed to introduce students to a particular parallel design pattern. The collection currently includes 44 patternlets (16 MPI, 17 OpenMP, 9 Pthreads, and 2 heterogeneous), of which we present a representative sample. We also present data that indicate the use of patternlets to introduce parallelism in CS2 produced a modest improvement in student understanding of parallel concepts.

Keywords:

design patterns; education; MPI; multiprocessing; multithreading; OpenMP; parallel; patternlets; teaching; threads

I. Introduction

Virtually every modern desktop, laptop, tablet, and smartphone contains a multicore CPU. Two-, four-, six-, eight-, ten-, twelve-, fourteen-, sixteen, eighteen-, twenty-, and twenty-two-core CPUs are readily available for building systems [4]. Vendors are also selling *accelerators*, ranging from co-processors with tens of cores [5] to general-purpose graphics processing units (GPUs) with hundreds or thousands of cores [9].

Today's software developers thus have unprecedented computational power at their fingertips, but to take advantage of this power, they must create their software using parallel and distributed computing (PDC) techniques. Unfortunately, very few developers – or CS educators – have received any training in these techniques.

One reason so few developers or educators have any parallel training is that prior to the TCPP [10] and CS2013 [11] curriculum recommendations, parallel computing was an elective topic. As a result, only developers or educators who happened to take an optional course in parallel computing received any training in parallel techniques, and relatively few CS programs offered such courses at the undergraduate level.

However, CS2013 contains a new *Parallel and Distributed Computing* knowledge area that includes 15 hours in the core CS curriculum. The *Systems Fundamentals* knowledge area adds additional PDC topics to be covered in the core.

The addition of PDC topics to the core, combined with the low levels of PDC expertise among CS educators, creates a special challenge for CS education. Before CS educators can train their students in parallel techniques, they must themselves develop sufficient expertise in those techniques. This is a challenge because PDC is a complex area, with multiple technologies to understand, at both the parallel hardware and the parallel software levels.

A. *Parallel Hardware Technologies*

There are three general categories of parallel hardware systems:

1. In *shared-memory systems*, multiple computational cores share the same primary memory, through which communication can take place. Modern desktops, laptops, tablets, and smartphones are all shared-memory systems. Most accelerator-equipped systems are a special case of shared-memory systems.
2. In *distributed-memory systems*, multiple computational nodes have their own local primary memories, but share no memory between one another. The nodes generally communicate through a network. Older Beowulf clusters and supercomputers were distributed-memory systems.
3. *Heterogeneous systems* are distributed-memory systems whose nodes are shared-memory systems. These include newer Beowulf clusters and supercomputers.

B. *Parallel Software Technologies*

Given the available hardware technologies, the situation is even more complex at the software layer:

1. *Shared-Memory Software Options*

Programs for shared-memory systems can be written using either:

- *Multithreading*, in which a program consists of multiple *threads* (flows of execution sharing the same address space) that communicate using the shared memory. Such programs can be written in a language with built-in thread support (e.g., C++11, Java), or in a sequential language using an external multithreading system (e.g., C with OpenMP or POSIX threads).
- *Multiprocessing*, in which a program consists of multiple *processes* (flows of execution, each having its own address space that is inaccessible to other processes) that communicate with one another by sending/receiving messages. Such programs may be written in a language that supports message passing (e.g., Erlang, Scala), or in a traditional language using an external message-passing library (e.g., C with MPI – the *message passing interface*).

For high performance computing, C with OpenMP is a commonly used approach, on both CPUs and co-processor accelerators. Nearly all modern C compilers include built-in support for OpenMP.

For the special case of GPU accelerators:

- Nvidia's CUDA (a dialect of C) is a popular technology, but is limited to Nvidia devices.
- OpenCL is platform independent and can use any of a system's cores, but it is far more complex than CUDA.
- OpenACC and OpenMP-4 promise to greatly simplify GPU programming, but often at a significant performance penalty compared to CUDA.

2. *Distributed-Memory Software Options*

Depending on the problem, programs for distributed-memory systems may be written in:

- A language that explicitly supports message-passing (e.g., Erlang, Scala).
- A traditional language with a message-passing library (e.g., C with MPI).
- Any language supported by the MapReduce/Hadoop framework (e.g., Java, Python, etc.).

For high performance computing, C with MPI is arguably the most commonly used of these approaches. Distributed parallelism is also useful for processing data too big to fit in main memory. For example, the MapReduce/Hadoop framework is popular for "big data" problems in which solutions can be computed using (*key, value*) pairs.

3. *Heterogenous Software Options*

Many programs for heterogeneous systems are built using C and MPI+X, with MPI being used to distribute processes across the system's nodes and enable communication between them, and the X depending on the underlying nodes. For example:

- *MPI+OpenMP* in a system whose nodes contained just multicore CPUs, or CPUs plus coprocessor accelerators.
- *MPI+CUDA* in a system whose nodes contained CPUs plus Nvidia GPU accelerators.
- *MPI+OpenCL* in a system whose nodes contain a mix of CPUs and non-Nvidia accelerators.

II. PARALLEL DESIGN PATTERNS

The large number of technologies available at the hardware and software layers can seem overwhelming to CS educators who are new to PDC. Thankfully, a higher-level conceptual framework exists that can provide some relief.

A. Parallel Patterns Background

Software developers have been writing software for parallel systems for over 30 years. During that time, these developers have noticed that certain parallel strategies and techniques were useful in solving many different problems. The professional developers began to name these strategies and techniques, so they could refer to them more conveniently. Eventually, this body of strategies and techniques was given a name of its own: *parallel design patterns*. Parallel design patterns are thus a collection of named strategies and techniques that industrial software developers have found to be useful in a variety of different contexts. Parallel professionals tend to think and design their software in terms of these patterns; as such, they exist at a "meta" or cognitive level above that of language syntax.

To illustrate, suppose a problem requires that a series of values be processed multiple times. Any traditional programmer knows to store these values in an array, and then use repetition (usually a for loop) to process them. This is a design pattern – a "meta" level strategy that ignores the language-specific syntax of arrays and loops, and that can be applied to any problem involving values that must be processed multiple times. If a person thinks in terms of such patterns, s/he can design a solution without worrying about the syntactic details.

In the same way, a professional parallel software developer thinks in terms of design patterns that:

1. Contribute to solving the problem at hand.
2. Will take advantage of the parallel capabilities of the underlying hardware.
3. Should automatically deliver improved performance when the program is ported from a system with fewer cores to one with more cores (scalability).

To illustrate, it is frequently the case that the threads or processes that make up a parallel solution compute local values that comprise partial solutions to the problem, and that solving the overall problem requires these local values to be combined in some way. The *reduction* pattern can be used to combine these local values in a way that (i) takes advantage of the system's parallel capabilities, and (ii) scales well as more cores are used. We examine this pattern in more detail in section III.D.

B. Cataloging and Categorizing the Parallel Patterns

There have been multiple efforts to catalog the parallel design patterns. Two of the most prominent are:

1. "Parallel Programming Patterns", by Johnson, Chen, Tasharofi, and Kjolstad (UIUC). This effort identifies 62 patterns and organizes them into ten categories [6].
2. "Our Pattern Language" (OPL) by Keutzer (Berkeley) and Mattson (Intel). This effort identifies 56 parallel patterns and organizes them into ten categories [7].

Both efforts organize the categories into hierarchical layers, with higher layers naming patterns that describe software architectures useful in solving broad classes of problems, middle layers naming patterns that describe broad algorithmic approaches, and lower layers naming patterns that are useful for implementing algorithmic steps. For example, *N-body Problems* and *Monte Carlo Simulations* are two of the high-level patterns. Algorithmic strategies like *Data Decomposition* and *Task Decomposition* are mid-level patterns. *Barrier*, *Reduction*, and *Message Passing* are all lower-level patterns. The two efforts are similar, but use slightly different names for some patterns and categories, and contain other subtle differences.

In recognition of the importance of these patterns, textbook authors have begun incorporating them into their parallel computing textbooks (e.g., [2] and [8]). One reason is that these patterns represent the wisdom and best practices accumulated by parallel software developers over 30+ years. As such, these patterns provide a body of knowledge that is far less ephemeral than hardware architectures or syntax details of the language *du jour*. The parallel patterns thus provide a stable framework for organizing parallel knowledge.

III. Patternlets

With experts thinking in terms of parallel design patterns and authors increasingly incorporating them into textbooks, CS educators must begin developing pedagogical tools to help their students learn about and master these patterns.

To that end, we have developed a collection of standalone programs, each illustrating the behavior of one or more parallel patterns. Each program is designed to be:

- *Minimalist*, eliminating extraneous features, so that students can easily grasp the pattern's essence.

- *Scalable*, so that students can see the pattern's behavior change as the number of threads or processes changes.
- *Syntactically correct*, so that students can use the code as a working model for their own coding.

Being minimalist, these are "little programs" in the tradition of applets and servlets, so we call them *patternlets*. The collection currently contains 44 programs: 16 for MPI, 17 for OpenMP, 9 for Pthreads, and 2 heterogeneous (MPI+OpenMP).

Patternlets can be used in different ways, including:

1. By an instructor in a traditional classroom setting, as part of a live coding / demonstration activity.
2. By a student in a closed lab session, as part of a self-paced lab exercise.

In the rest of this section, we present a small subset of the current collection of patternlets. We will also use the term *tasks* as a general term for threads or processes.

A. The Single Program Multiple Data (SPMD) Pattern

Most MPI and OpenMP programs employ the single program multiple data (SPMD) pattern, in which different instances of the same (single) program operate on different (multiple) data values. Figure 1 shows an OpenMP patternlet for this pattern.

```
#include <stdio.h>    // printf()
#include <omp.h>      // OpenMP functions

int main(int argc, char** argv) {
  printf("\n");

//  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    int numThreads = omp_get_num_threads();
    printf("Hello from thread %d of %d\n", id, numThreads);
  }

  printf("\n");
  return 0;
}
```

Figure 1. *spmd.c* (OpenMP version)

When compiled and run, *spmd.c* produces the output shown in Figure 2.

```
Hello from thread 0 of 1
```

Figure 2. Running *spmd.c* (OpenMP, 1 Thread)

We then uncomment the `#pragma omp parallel` directive, recompile and rerun the program. On a quad core computer, this produces behavior like that shown in Figure 3.

```
Hello from thread 0 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 1 of 4
```

Figure 3. Running *spmd.c* (OpenMP, 4 Threads)

When uncommenting a single line of code produces such a significant difference in the program's behavior, it sparks a student's curiosity, as does the non-deterministic order in which the "Hello" messages appear. The code and its different behaviors thus provide an experiential basis for explaining the concept of multithreading, and how the **`omp_get_thread_num()`** call provides each thread with a different identification number, which provides a simple illustration of the SPMD pattern.

The same pattern is also commonly used in MPI. Figure 4 shows an MPI patternlet for the SPMD pattern.

```
#include <stdio.h>   // printf()
#include <mpi.h>     // MPI functions

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

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
    MPI_Get_processor_name(myHostName, &length);

    printf("Hello from process %d of %d on %s\n", id, numProcesses, myHostName);

    MPI_Finalize();
    return 0;
}
```

Figure 4. *spmd.c* (MPI version)

When run on our Beowulf cluster with this command:

```
mpirun -np 1 ./spmd
```

this patternlet produces the output shown in Figure 5.

```
Hello from process 0 of 1 on node-01
```

Figure 5. Running *spmd.c* (MPI, 1 Process)

However, when we run it and specify more processes:

```
mpirun -np 4 ./spmd
```

the same program produces output as shown in Figure 6.

```
Hello from process 3 of 4 on node-04
Hello from process 0 of 4 on node-01
Hello from process 2 of 4 on node-03
Hello from process 1 of 4 on node-02
```

Figure 6. Running *spmd.c* (MPI, 4 Processes)

This patternlet and its behavior can thus be used to teach students about multiprocessing. By making clear that **`MPI_Comm_rank()`** and **`MPI_Get_processor_name()`** return different values in different processes, this same patternlet can be used to help students understand the SPMD pattern. Finally, by displaying the name of the node on which each process is running, it can be used to help students see the difference between distributed and non-distributed computations.

B. The Barrier Pattern

It is sometimes necessary to synchronize the executions of the different tasks that make up a parallel computation. The *barrier* pattern provides this capability. Figure 7 presents an OpenMP patternlet for this pattern.

```
#include <stdio.h>    // printf()
#include <omp.h>      // OpenMP functions
#include <stdlib.h>   // atoi()

int main(int argc, char** argv) {
    printf("\n");
    if (argc > 1) {
        omp_set_num_threads( atoi(argv[1]) );
    }

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
        printf("Thread %d of %d is BEFORE the barrier.\n", id, numThreads);

//        #pragma omp barrier

        printf("Thread %d of %d is AFTER the barrier.\n", id, numThreads);
    }

    printf("\n");
    return 0;
}
```

Figure 7. *barrier.c* (OpenMP version)

When run with four threads, as follows:

```
./barrier 4
```

the program produces output like that shown in Figure 8.

```
Thread 1 of 4 is BEFORE the barrier.
Thread 0 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 3 of 4 is AFTER the barrier.
```

Figure 8. Running *barrier.c* (OpenMP, Initial Behavior)

We then uncomment the **`#pragma omp barrier`** directive, recompile and rerun the program exactly the same way. This simple change produces output like that in Figure 9.

```
Thread 1 of 4 is BEFORE the barrier.
Thread 0 of 4 is BEFORE the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 3 of 4 is AFTER the barrier.
```

Figure 9. Running *barrier.c*, (OpenMP, Modified Behavior)

Without the barrier, the before-and-after behaviors of the threads are interleaved. With the barrier, a thread can only perform its ‘after’ behavior after all threads have reached the barrier (i.e., having completed their ‘before’ behaviors).

MPI also provides a barrier construct; however, C’s standard output (stdout) may not preserve the order of write operations from multiple distributed processes, so an MPI *barrier* patternlet is more complex than Figure 7, as can be seen in Figure 10.

```
#include <stdio.h>    // printf()
#include <mpi.h>      // MPI

#define BUFFER_SIZE 200
#define MASTER      0

void sendReceivePrint(int id, int numProcesses, char* hostName, char* position) {
    char buffer[BUFFER_SIZE] = {'\0'};;
    MPI_Status status;

    if (id != MASTER) {     // Worker: Build a message and send it to the Master
        int length = sprintf(buffer, "Process #%d of %d on %s is %s the barrier.\n",
                                       id, numProcesses, hostName, position);
        MPI_Send(buffer, length+1, MPI_CHAR, MASTER, 0, MPI_COMM_WORLD);
    } else {                // Master: Receive and print the messages from all Workers
        for(int i = 1; i < numProcesses; ++i) {
           MPI_Recv(buffer, BUFFER_SIZE, MPI_CHAR, MPI_ANY_SOURCE,
                      MPI_ANY_TAG, MPI_COMM_WORLD, &status);
           printf(buffer);
        }
    }
}

int main(int argc, char** argv) {
    int id = -1, numProcesses = -1, length = -1;
    char myHostName[MPI_MAX_PROCESSOR_NAME] = {'\0'};

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
    MPI_Get_processor_name (myHostName, &length);

    sendReceivePrint(id, numProcesses, myHostName, "BEFORE");

//    MPI_Barrier(MPI_COMM_WORLD);

    sendReceivePrint(id, numProcesses, myHostName, "AFTER");

    MPI_Finalize();
    return 0;
}
```

Figure 10. *barrier.c* (MPI version)

When run with four processes, as follows:

```
mpirun -np 4 ./barrier
```

the program produces output like that shown in Figure 11.

```
Process 2 of 4 on node-03 is BEFORE the barrier.
Process 2 of 4 on node-03 is AFTER the barrier.
Process 3 of 4 on node-04 is BEFORE the barrier.
Process 1 of 4 on node-02 is BEFORE the barrier.
Process 1 of 4 on node-02 is AFTER the barrier.
Process 3 of 4 on node-04 is AFTER the barrier.
```

Figure 11. Running *barrier.c* (MPI, Initial Behavior)

We then uncomment the `MPI_Barrier()` function, recompile and rerun the program. Run exactly the same way as before, the inclusion of the barrier produces output like that in Figure 12.

```
Process 2 of 4 on node-03 is BEFORE the barrier.
Process 1 of 4 on node-02 is BEFORE the barrier.
Process 3 of 4 on node-04 is BEFORE the barrier.
Process 1 of 4 on node-02 is AFTER the barrier.
Process 3 of 4 on node-04 is AFTER the barrier.
Process 2 of 4 on node-03 is AFTER the barrier.
```

Figure 12. Running *barrier.c* (MPI, Modified Behavior)

Once again, we see that in the absence of a barrier, the before-and-after behaviors of the worker processes are interleaved. With the barrier, no worker process can perform its ‘after’ behavior until all processes have completed their ‘before’ behaviors and then reached the barrier.

The *SPMD* and *Barrier* patterns are relatively simple concepts, and the patternlets make it fairly easy for students to understand them, so we now turn to some more interesting patterns.

C. The Parallel Loop Pattern

Sequential programs typically spend much of their time iterating through a loop’s statements. If a loop’s iterations are independent of one another, the *Parallel Loop* pattern can be used to divide the loop’s iterations among the tasks.

OpenMP provides built-in support for this pattern, as can be seen in the patternlet shown in Figure 13.

```
#include <stdio.h>    // printf()
#include <stdlib.h>   // atoi()
#include <omp.h>      // OpenMP

int main(int argc, char** argv) {
  const int REPS = 8;

  printf("\n");
  if (argc > 1) {
    omp_set_num_threads( atoi(argv[1]) );
  }

  #pragma omp parallel for
  for (int i = 0; i < REPS; i++) {
     int id = omp_get_thread_num();
     printf("Thread %d performed iteration %d\n", id, i);
  }

  printf("\n");
  return 0;
}
```

Figure 13. *parallelLoopEqualChunks.c* (OpenMP)

When run with a single thread, as follows:

```
./parallelLoopEqualChunks 1
```

the program produces the output shown in Figure 14.

```
Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 0 performed iteration 4
Thread 0 performed iteration 5
Thread 0 performed iteration 6
Thread 0 performed iteration 7
```

Figure 14. Running *parallelLoopEqualChunks.c* (OpenMP, 1 Thread)

But when run with two threads, the behavior changes to something like what is shown in Figure 15.

```
Thread 0 performed iteration 0
Thread 1 performed iteration 4
Thread 0 performed iteration 1
Thread 1 performed iteration 5
Thread 0 performed iteration 2
Thread 1 performed iteration 6
Thread 0 performed iteration 3
Thread 1 performed iteration 7
```

Figure 15. Running *parallelLoopEqualChunks.c* (OpenMP, 2 Threads)

While the two threads' outputs are interleaved, it is still fairly easy to see that thread 0 is performing iterations 0-3 and thread 1 is performing iterations 4-7, illustrating exactly how this pattern divides a loop's iterations among the threads. The patternlet makes it easy to try different numbers of threads, change the number of iterations, and so on.

Unlike OpenMP, MPI provides no built-in support for the *Parallel Loop* pattern, so we must implement the pattern ourselves. Figure 16 shows one possible implementation.

```
#include <stdio.h>    // printf()
#include <mpi.h>      // MPI
#include <math.h>     // ceil()

int main(int argc, char** argv) {
  const int REPS = 8;
  int id = -1, numProcesses = -1, i = -1,
      start = -1, stop = -1, chunkSize = -1;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);

  chunkSize = (int)ceil( (double)REPS / numProcesses );
  start = id * chunkSize;

  if ( id < numProcesses-1 ) {
     stop = (id + 1) * chunkSize;
  } else {
     stop = REPS;
  }

  for (i = start; i < stop; i++) {
    printf("Process %d performed iteration %d\n", id, i);
```

```
   }

   MPI_Finalize();
   return 0;
 }
```

Figure 16. *parallelLoopEqualChunks.c* (MPI)

When run with a single process:

```
mpirun -np 1 ./parallelLoopEqualChunks
```

the program produces output similar to that of Figure 14, but with the word “Process” replacing the word “Thread”. When run with two processes, the program produces output like that shown in Figure 17.

```
Process 1 performed iteration 4
Process 1 performed iteration 5
Process 1 performed iteration 6
Process 1 performed iteration 7
Process 0 performed iteration 0
Process 0 performed iteration 1
Process 0 performed iteration 2
Process 0 performed iteration 3
```

Figure 17. Running *parallelLoopEqualChunks.c* (MPI, 2 Processes_

As before, varying the number of processes or iterations is easy; Figure 18 shows an execution with four processes.

```
Process 1 performed iteration 2
Process 1 performed iteration 3
Process 2 performed iteration 4
Process 2 performed iteration 5
Process 0 performed iteration 0
Process 0 performed iteration 1
Process 3 performed iteration 6
Process 3 performed iteration 7
```

Figure 18. Running *parallelLoopEqualChunks.c* (MPI, 4 Processes)

D. The Reduction Pattern

In a parallel computation, the tasks often compute local partial results, which must then be combined into a global, overall result. The *Reduction* pattern takes advantage of a computation’s existing parallelism to speed up this combining step.

To illustrate, suppose that we need to determine how many red pixels an image contains, and that we use the *Parallel Loop* pattern to divide the scanning of this image among eight tasks, which respectively find 6, 8, 9, 1, 5, 7, 2, and 4 red pixels. To solve the problem these intermediate values must be summed. For *t* tasks, doing this summing sequentially takes time *O(t)*, but the *Reduction* pattern can exploit the computation’s existing parallelism to reduce this time to *O(lg(t))*, as shown in Figure 19.

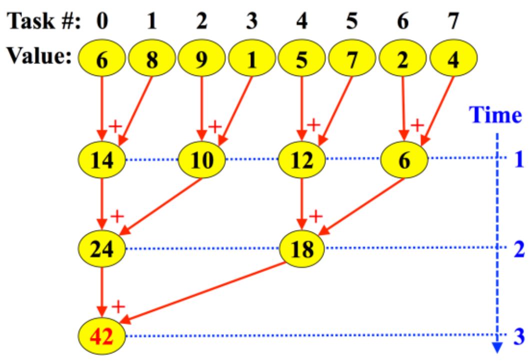

Figure 19. The Reduction Pattern's Behavior

Note that the *Reduction* pattern performs the same number of total additions (*t-1*) as a sequential summing. However by performing *t/2* of the additions in parallel at time 1, *t/4* of the additions in parallel at time 2, and so on, the *Reduction* pattern combines local results with better time-efficiency.

Because it is so commonly needed, both OpenMP and MPI provide built-in support for the *Reduction* pattern. Figure 20 presents an OpenMP patternlet for this pattern.

```
#include <stdio.h>    // printf()
#include <omp.h>      // OpenMP
#include <stdlib.h>   // rand()

void initialize(int* a, int n);
int sequentialSum(int* a, int n);
int parallelSum(int* a, int n);

#define SIZE 1000000

int main(int argc, char** argv) {
   int array[SIZE];

   if (argc > 1) {
        omp_set_num_threads( atoi(argv[1]) );
   }

   initialize(array, SIZE);
   printf("\nSeq. sum: \t%d\nPar. sum: \t%d\n\n",
            sequentialSum(array, SIZE),
            parallelSum(array, SIZE) );

   return 0;
}

void initialize(int* a, int n) {         // fill array with random values
   for (int i = 0; i < n; ++i) {
      a[i] = rand() % 1000;
   }
}

int sequentialSum(int* a, int n) {       // sum the array sequentially
   int sum = 0;

   for (int i = 0; i < n; ++i) {
      sum += a[i];
   }

   return sum;
}
```

```
int parallelSum(int* a, int n) {         // sum the array using multiple threads
   int sum = 0;

//   #pragma omp parallel for // reduction(+:sum)
   for (int i = 0; i < n; ++i) {
      sum += a[i];
   }

   return sum;
}
```
Figure 20. *reduction.c* (OpenMP)

This patternlet builds an array of random values, and then sums the array twice, once in a function named **`sequentialSum()`** and again in a function named **`parallelSum()`**. Initially, the two functions produce identical output, as shown in Figure 21.

```
Seq. sum:     499079147
Par. sum:     499079147
```
Figure 21. *reduction.c* (OpenMP, 1 Thread)

If we uncomment the **`#pragma omp parallel for`** directive but leave the **`reduction(+:sum)`** clause commented out, the program produces output like that shown in Figure 22, which is incorrect.

```
Seq. sum:     499079147
Par. sum:     166972254
```
Figure 22. *reduction.c* (OpenMP, 4 Threads, No Reduction)

The problem is that each thread is adding its array items to a shared **`sum`** variable, creating a data race. This provides an opportunity to introduce the topic of race conditions, and have students brainstorm possible solutions to the problem.

Students eventually arrive at the idea to have each thread sum its array items in its own local **`sum`** variable, and for these local sums to be added together at the end. This provides a natural means of introducing and explaining how the *Reduction* pattern and OpenMP's **`reduction(+:sum)`** clause work. When this clause is uncommented, the program will again produce correct output, identical to that shown in Figure 21, despite using multiple threads.

Besides the **`+`** operator shown in Figure 20, OpenMP also permits the **`*`**, **`-`**, **`&`**, **`|`**, **`^`**, **`&&`**, and **`||`** operators to be used within a **`reduction`** clause. User-defined reductions are also supported as of OpenMP 4.0.

MPI also includes support for the *Reduction* pattern. Figure 23 shows a different MPI patternlet for this pattern.

```
#include <stdio.h>   // printf()
#include <mpi.h>     // MPI

#define MASTER 0

int main(int argc, char** argv) {
  int numProcs = -1, myRank = -1,
      square = -1, max = -1, sum = 0;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
```

```
   square = (myRank+1) * (myRank+1);

   printf("Process %d computed %d\n", myRank, square);

   MPI_Reduce(&square, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

   MPI_Reduce(&square, &max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

   if (myRank == MASTER) {
      printf("\nThe sum of the squares is %d\n", sum);
      printf("\nThe max of the squares is %d\n\n", max);
   }

   MPI_Finalize();

   return 0;
}
```

Figure 23. *reduction.c* (MPI)

In Figure 23, a process computes the square of (its id number plus 1). The program then uses the `MPI_Reduce()` function twice: once to sum these squares, and again to find the maximum of these squares. When run with 10 processes, the program produces output like that shown in Figure 24.

```
Process 7 computed 64
Process 8 computed 81
Process 9 computed 100
Process 0 computed 1
Process 1 computed 4
Process 2 computed 9
Process 3 computed 16
Process 4 computed 25
Process 5 computed 36
Process 6 computed 49

The sum of the squares is 385

The max of the squares is 100
```

Figure 24. *reduction.c*, (MPI, 10 Processes)

Besides the *sum* and *maximum* operations used in Figure 23, MPI supports the *product*, *minimum*, *minimum and its location*, *maximum and its location*, *logical and*, *logical or*, *logical exclusive or*, *bitwise and*, *bitwise or*, and *bitwise exclusive or* operations for combining local results. MPI also supports user-defined operations, which must be associative.

E. Other Patterns

We also have patternlets for the *Fork-Join* (in both OpenMP and Pthreads), *Master-Worker*, *Critical Section*, *Broadcast*, *Scatter*, *Gather*, *Parallel Loop* patterns with different chunk sizes or scheduling algorithms, and others. Space limitations prevent us from presenting all of them; as one example, Figure 25 shows our MPI patternlet for the *Gather* pattern.

```
#include <stdio.h>      // printf()
#include <stdlib.h>     // malloc()
#include <mpi.h>        // MPI

#define SIZE 3
```

```
#define MASTER 0

void print(int id, char* arrName, int* arr, int arrSize);


int main(int argc, char** argv) {
   int  computeArray[SIZE];                            // array1
   int* gatherArray = NULL;                            // array2
   int  numProcs = -1, myRank = -1, totalGatheredVals = -1;

   MPI_Init(&argc, &argv);                             // initialize
   MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

   for (int i = 0; i < SIZE; i++) {                    // everyone: load array1 with
      computeArray[i] = myRank * 10 + i;               //   3 distinct values
   }

   print(myRank, "computeArray", computeArray, SIZE);  // everyone: show array1

   if (myRank == MASTER) {                             // master:
      totalGatheredVals = SIZE * numProcs;             //  allocate array2
      gatherArray = malloc( totalGatheredVals * sizeof(int) );
   }

   MPI_Gather(computeArray, SIZE, MPI_INT,             //  gather array1 values
               gatherArray, SIZE, MPI_INT,             //   into array2
               MASTER, MPI_COMM_WORLD);                //   at master process

   if (myRank == MASTER) {                             // master: show array2
      print(myRank, "gatherArray", gatherArray, totalGatheredVals);
   }

   free(gatherArray);                                  // clean up
   MPI_Finalize();
   return 0;
}

void print(int id, char* arrName, int* arr, int arrSize) {
    printf("Process %d, %s: ", id, arrName);
    for (int i = 0; i < arrSize; ++i) {
        printf(" %d", arr[i]);
    }
    printf("\n");
}
```

Figure 25. *gather.c* (MPI)

In this patternlet, each process builds an array containing three unique values and then prints its array. The master process then allocates an array in which to gather these distributed arrays. All processes then collectively execute the **`MPI_Gather()`** command; and finally the master process prints the array containing the gathered values. When compiled and run with two processes, the program produces output like that shown in Figure 26.

```
Process 0, computeArray:  0 1 2
Process 1, computeArray:  10 11 12
Process 0, gatherArray:  0 1 2 10 11 12
```

Figure 26. *gather.c* (MPI, 2 Processes)

When run with four processes, the same program produces output like that shown in Figure 27.

```
Process 0, computeArray:  0 1 2
Process 2, computeArray:  20 21 22
Process 1, computeArray:  10 11 12
Process 3, computeArray:  30 31 32
Process 0, gatherArray:  0 1 2 10 11 12 20 21 22 30 31 32
```

Figure 27. *gather.c* (MPI, 4 Processes)

When run with six processes, the program produces output like that shown in Figure 28.

```
Process 5, computeArray:  50 51 52
Process 0, computeArray:  0 1 2
Process 1, computeArray:  10 11 12
Process 4, computeArray:  40 41 42
Process 3, computeArray:  30 31 32
Process 2, computeArray:  20 21 22
Process 0, gatherArray:  0 1 2 10 11 12 20 21 22 30 31 32 40 41 42 50 51 52
```

Figure 28. *gather.c* (MPI, 8 Processes)

Most of the patternlets let a student (or instructor) interactively vary the number of tasks; through such experimental activity, they can explore a pattern's scalability and behavioral changes.

Another important parallel design pattern is the *Mutual Exclusion* pattern, a synchronization pattern that keeps two threads from entering a critical section of the program (e.g., writing to shared memory) at the same time. Figure 29 presents one of our OpenMP patternlets for this pattern, in which we compare the execution times of the **`atomic`** and **`critical`** directives:

```
#include<stdio.h>
#include<omp.h>

void print(char* label, int reps, double balance, double total, double average) {
    printf("\nAfter %d $1 deposits using '%s': \
            \n\t- balance = %0.2f, \
            \n\t- total time = %0.12f, \
            \n\t- average time per deposit = %0.12f\n\n",
               reps, label, balance, total, average);
}

int main() {
    const int REPS = 1000000;
    int i;
    double balance = 0.0,
           startTime = 0.0,
           stopTime = 0.0,
           atomicTime = 0.0,
           criticalTime = 0.0;

    printf("\nYour starting bank account balance is %0.2f\n", balance);

    // simulate many deposits using atomic
    startTime = omp_get_wtime();
    #pragma omp parallel for
    for (i = 0; i < REPS; i++) {
        #pragma omp atomic
        balance += 1.0;
    }
    stopTime = omp_get_wtime();
    atomicTime = stopTime - startTime;
    print("atomic", REPS, balance, atomicTime, atomicTime/REPS);
    // simulate the same number of deposits using critical
    balance = 0.0;
    startTime = omp_get_wtime();
    #pragma omp parallel for
    for (i = 0; i < REPS; i++) {
         #pragma omp critical
         {
             balance += 1.0;
         }
    }
    stopTime = omp_get_wtime();
    criticalTime = stopTime - startTime;
    print("critical", REPS, balance, criticalTime, criticalTime/REPS);

    printf("criticalTime / atomicTime ratio: %0.12f\n\n",
             criticalTime / atomicTime);

    return 0;
}
```

Figure 29. *critical2.c* (OpenMP)

When run, this patternlet produces the output like that shown in Figure 30, illustrating that while both **`atomic`** and **`critical`** provide mutual exclusive behavior, the **`critical`** directive is much more time-expensive:

```
Your starting bank account balance is 0.00

After 1000000 $1 deposits using 'atomic':
       - balance = 1000000.00,
       - total time = 0.148041009903,
       - average time per deposit = 0.000000148041


After 1000000 $1 deposits using 'critical':
       - balance = 1000000.00,
       - total time = 2.442871809006,
       - average time per deposit = 0.000002442872

criticalTime / atomicTime ratio: 16.501318186137
```

Figure 30. *critical2.c* (OpenMP, 8 threads)

A previous patternlet shows students that if the **`atomic`** and **`critical`** directives are “commented out” so that multiple threads can enter the critical section simultaneously (i.e., trying to add $1 to variable **`balance`**), then their bank account balance is less than $1,000,000 at the end; the resulting race condition costs them imaginary money.

The situations in which the **`atomic`** directive can be used are limited to those in which the underlying hardware provides instruction-level support for performing the subsequent operations atomically. To illustrate this, other patternlets present race conditions for which the subsequent operations cannot be performed atomically. Trying to use **`atomic`** in such situations produces a compilation error, leaving **`critical`** as the only viable OpenMP alternative.

IV. USING PATTERNLETS TO TEACH PARALLEL THINKING

We have spread parallel topics across our curriculum, including:

- In *Data Structures* (CS2, a first-year required course): using OpenMP to solve embarrassingly parallel problems.
- In *Algorithms* (CS3, a second-year required course): exploring a variety of parallel algorithms (searching, sorting, graph, etc.).
- In *Programming Languages* (a second-year required course): programming language constructs for message passing and synchronization of parallel programs.
- In *Operating Systems & Networking* (a third-year required course): different ways to implement the various constructs for synchronization and message passing.
- In *High Performance Computing* (a third- or fourth-year elective course): an in-depth look at writing scalable parallel programs using MPI, OpenMP, CUDA, and Hadoop.

By spreading these topics across our curriculum, every student is exposed to PDC, and students who want more depth may get it. We have used the patternlets in several of these courses to introduce our students to different parallel design patterns. In the next section, we describe one of these uses: our CS2 course.

A. Example: Use of Patternlets in CS2

In Fall 2008, we began introducing first-year students to shared-memory parallelism in our CS2 *Data Structures* course. For the first few years, we spent a week on parallel topics in this course, as follows:

1. *Monday*: An introductory lecture on multicore CPUs, multithreading as a means of taking advantage of all of CPU’s cores, and OpenMP as a multithreading library.
2. *Tuesday*: A 2-hour closed lab session in which we provided the students with a Matrix class. Students (a) timed its sequential addition and transpose operations on large matrices, (b) used OpenMP to “parallelize” those operations, and then (c) timed those parallel operations using varying numbers of threads. They then (d) used a spreadsheet to create charts that visualize the relationship between the number of threads employed and the speed at which a given problem is solved.
3. *Wednesday*: A follow up lecture that continued the exploration of multithreading concepts, to reinforce and expand on what the students had done in the lab session.
4. *Friday*: A session introducing the basic ideas of parallel algorithm design, using an active learning exercise in which the students explored parallel sorting, culminating in the parallel merge-sort algorithm.

In the Spring 2013 semester, we proceeded as in the Fall course, except that (i) we concluded the Monday session with a live-coding demo using OpenMP patternlets, and (ii) we replaced our Wednesday concepts lecture with a live-coding demo using OpenMP patternlets that demonstrated those concepts in action. Everything else that week remained the same.

Our Spring students were much more engaged by the live-coding patternlet demos than the students of previous semesters who had traditional lectures. Many asked *"What if you change…"* kinds of questions during the Monday and Wednesday classes; using the patternlets, we could answer those questions immediately by changing the code, recompiling the patternlet, running the program, and observing how the change affected the program's behavior. The ability to change the code and view the resulting difference in behavior helps make PDC topics less abstract and more engaging for students.

B. Effect of Using of Patternlets in CS2

One way that we assess student learning from that week is through the use of four final exam questions on parallelism and OpenMP. Since we had not used the patternlets in our Fall course but had used them in our Spring course, we analyzed student performance on these questions, to see if our use of the patternlets had any measurable effect on student learning.

The average score on these exam questions for students in the Fall course (the "no patternlets" group) was 2.95 out of 4. The students in the Spring course (the "with patternlets" group) scored 3.05 out of 4 on those same questions, a 2.5% improvement. This improvement was not statistically significant ($p = 0.293$), perhaps due to small sample sizes (41 students in the Fall course, 38 students in the Spring course).

However, the two student populations were not the same: Most of the students in our Fall course were 3rd-year electrical engineering majors who had successfully completed two years of our Engineering curriculum, had strong quantitative and analytical skills, and were keenly interested in learning how to use all the cores in a multicore processor. By contrast, most of the students in our Spring course were 1st-year students with just one semester of college experience, who were still deciding whether or not to pursue a computer science major. Given this difference in experience, maturity, and commitment in the two student populations, we believe that the improved learning shown by our Spring students was much more significant than the statistics indicate.

We have also used the patternlets in our more advanced courses, including *Operating Systems & Networking* and *High Performance Computing*. Since each patternlet is a working program for one or more design patterns, our students have made extensive use of these as models for their own code and expressed appreciation for them in end-of-course evaluations. Faculty members at other universities have also begun using the patternlets in courses such as *Algorithms*, *Computer Organization*, and *Parallel Computing*, and have similarly reported very positive feedback from their student course evaluations.

V. Conclusions

Since PDC is now a core topic in the CS curriculum, CS educators must begin creating the pedagogical tools and materials to help their students understand PDC concepts.

Professional parallel software developers think in terms of parallel design patterns, so the more CS educators can help their students think in terms of these patterns, the closer those students' computational thinking skills will be to those of professional developers.

In this paper, we have presented the *patternlets*, a collection of software programs, each designed to introduce students to a particular parallel design pattern. The patternlets are (i) *minimalist*, to present the pattern and its behavior as simply as possible; (ii) *scalable*, so that students can vary the number of tasks and see how the pattern's behavior changes; and (iii) *syntactically correct*, so that students have a correct, working model on which they can base their own code. We believe that patternlets are an ideal way to introduce students to parallel patterns. After this first exposure, we believe it is important to show students an *exemplar* – a "real world" problem whose solution uses the same pattern(s) – to illustrate that pattern's use in the context of a socially relevant problem. See [1] for more on this teaching strategy.

We have also argued that replacing traditional lectures with live-coding demos using the patternlets improved student learning in our CS2 course. Students are motivated to learn how to take full advantage of the multicore hardware in their desktops, laptops, tablets, and/or phones; the interactive nature of the patternlets allows students to explore the relationship between the code and the behavior it produces, providing experiential learning that helps make the PDC concepts less abstract.

The evolving patternlet collection currently includes 44 programs – 16 for MPI, 17 for OpenMP, 9 for Pthreads, and 2 heterogeneous – making them suitable for use in courses throughout the undergraduate curriculum. Each patternlet resides in a folder containing a Makefile that can be used to build it using the GNU C compiler (gcc). Each patternlet source file also contains a header comment that details an exercise for students to complete with that patternlet. All patternlets are currently written in C or C++, but we are beginning to explore the development of patternlets written in Python. The collection continues to grow; the most recent release is always freely available at the *CSinParallel.org* website [3].

We have received enthusiastic feedback from faculty members at other universities who are using the patternlets to introduce their students to parallel patterns; we hope that other instructors will find them equally useful. We also invite others to contribute new patternlets to the collection, and we welcome feedback on ways in which the collection can be improved.

Acknowledgment

We wish to thank the National Science Foundation, whose support through grant DUE-1225739 made this work possible. We also wish to thank the NSF-IEEE/TCPP group for providing this venue for publicizing our work.

References

[1] J. Adams, R. Brown, E. Shoop, Patterns and Exemplars: Compelling Strategies for Teaching Parallel and Distributed Computing to CS Undergraduates. *2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum* (Boston, MA, May 2013). EduPar-13. 1244-1251. doi: 10.1109/IPDPSW.2013.275.

[2] G. Barlas, Multicore and GPU Programming: An Integrated Approach, Morgan Kaufmann, 2015.

[3] R. Brown, E. Shoop, J. Adams, CSinParallel. Online, accessed 2016-05-25, http://csinparallel.org.

[4] Intel Corp., Intel Xeon Processor E5 Family. Online, accessed 2016-05-25 http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e5-family.html.

[5] Intel Corp., Intel Xeon Phi Product Family. Online, accessed 2016-05-25, http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[6] R. Johnson, Center for Programming Models for Scalable Parallel Computing, Parallel Programming Patterns, Final Report. April, 2013. Online, accessed 2016-05-25, http://hdl.handle.net/2142/43368.

[7] K. Keutzer, T. Mattson, A Pattern Language for Parallel Programming ver2.0: Online, accessed 2016-05-25, http://parlab.eecs.berkeley.edu/wiki/patterns/patterns.

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

[9] Nvidia Corp., CUDA GPUs, Online, accessed 2016-05-25, https://developer.nvidia.com/cuda-gpus.

[10] S. K. Prasad, A. Chtchelkanova, F. Dehne, M. Gouda, A. Gupta, J. Jaja, K. Kant, A., La Salle, R. LeBlanc, A. Lumsdaine, D. Padua, M. Parashar, V. Prasanna, Y. Robert, A. Rosenberg, S. Sahni, B. Shirazi, A. Sussman, C. Weems, and J. Wu. 2012. NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates, Version I, Online: http://www.cs.gsu.edu/~tcpp/curriculum/index.php, 55 pages.

[11] M. Sahami, S. Roach, E. Cuadro-Vargas, and R. LeBlanc. ACM/IEEE-CS Computer Science Curriculum 2013. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education* (Denver, CO, March 2013). SIGCSE'13, 13-14. doi: http://dx.doi.org/10.1145/2445196.2445206.

*Author Biography & Photograph

Author Bio

Joel Adams is professor and Chair of the Department of Computer Science at Calvin College. He has been teaching his students about parallel and distributed computing since 1997. He has designed four Beowulf clusters including Microwulf, the first cluster to break the $100/GFLOP barrier. He is one of the PIs on *CSinParallel.org*, an NSF-funded project to create and distribute high quality pedagogical materials for teaching students about parallel and distributed computing. He is a two-time Fulbright scholar (Mauritius, 1998; Iceland, 2005) and is an ACM Distinguished Educator.

Author Photo