Teaching Parallel and Distributed Computing Concepts Using OpenMPI and Java

Joel C. Adams
Department of Computer Science
Calvin University
Grand Rapids, Michigan, USA
adams@calvin.edu

***Abstract*—The ACM/IEEE CS 2013 curriculum recommendations state that every undergraduate CS major should learn about parallel and distributed computing (PDC). One way to accomplish this is to teach students about the Message Passing Interface (MPI), a platform that is commonly used on modern supercomputers and Beowulf clusters, but can also be used on a Network of Workstations (NoW), or a multicore laptop or desktop. MPI incorporates many PDC concepts and can serve as a platform for hands-on learning activities in which students must apply those concepts. The MPI standard defines language bindings for Fortran and C/C++, but many university instructors lack expertise in these languages, preventing them from using MPI in their courses. OpenMPI is a free implementation of the MPI standard that also provides Java bindings for MPI. This paper describes how to install OpenMPI with these Java bindings; to illustrate the use of these bindings, the paper also presents several *patternlets*—minimalist example programs—that show how to implement PDC design patterns using OpenMPI and Java. This provides a new means of introducing students to PDC concepts.**

Keywords—computing, distributed, education, Java, MPI, OpenMPI, parallel, patternlets, patterns, performance

I. Introduction

Many "real world" problems are sufficiently challenging that parallel and distributed computing (PDC) techniques are needed to solve those problems in a reasonable length of time. Examples of such problems include genetic sequence alignment, weather forecasting, Monte Carlo models of natural phenomena, seismological data analysis, and many others.

Since CS graduates need to be able to solve such problems, the *NSF/IEEE TCPP Curriculum Initiative* [15] and the *ACM/IEEE CS 2013 Curriculum* guidelines [11] state that all undergraduate CS majors should learn specific aspects of PDC. Likewise, the *Accreditation Board for Engineering and Technology* (ABET) requires all ABET-accredited CS programs to expose their majors to PDC [1]. In light of this, how do we (CS educators) equip our students with the PDC concepts and hands-on skills they need to be modern software developers?

A. The Message Passing Interface (MPI)

Beowulf clusters [10] and modern supercomputers are distributed-memory multiprocessors, meaning the system is made up of multiple independent computers called *nodes*, each with its own local memory, but no shared memory. The nodes are connected together via a low-latency, high-bandwidth network through which the nodes can communicate.

The *Message Passing Interface* (MPI) [14] is a software platform that is commonly used on modern supercomputers and Beowulf clusters, but that may also be used effectively on a Network of Workstations (NoW) or a multicore laptop or desktop. This ability to run on almost any hardware platform makes MPI a useful tool for introducing students to PDC.

MPI provides: (i) a library of functions for inter-process communication, and (ii) a runtime environment for launching a multi-process computation across the nodes of a Beowulf cluster or supercomputer. MPI uses the *Single Program, Multiple Data* (SPMD) pattern of parallelism, in which copies of the same program (processes) are launched on the supercomputer's nodes—the *Single Program* part of SPMD. The MPI runtime assigns each process a different number called its rank, which the processes can use to perform different tasks or process different chunks of data—the *Multiple Data* part of SPMD. There are two free, open source versions of MPI available—*MPICH* [5] and *OpenMPI* [16]—plus commercial versions.

The MPI Standard [14] specifies that an implementation of the standard must provide bindings for three languages: Fortran, C, and C++. (Third parties have developed bindings for other languages, but they are not part of the MPI standard.) When installing MPI, a person just specifies their preferred language and its compiler, and the MPI installer handles the details.

However, very few universities teach Fortran anymore, and instruction in C/C++ appears to be gradually declining as new languages like Go and Rust provide attractive alternatives for systems-level programming. Additionally, the computing programs at many universities are heavily Java-oriented, for a variety of reasons, including:

- In U.S. high schools, the Advanced Placement Computer Science 'A' course (AP CS-A) is taught in Java, so many universities match that language in their CS1 course.
- Outside the U.S., the International Baccalaureate (IB) CS course is officially language-agnostic, but Java is commonly used as the hands-on programming language.
- Java is used extensively in industry. As a result, some universities focus on Java to prepare their students for careers as Java developers.
- Some (many?) university instructors have extensive expertise in Java but have limited C/C++ expertise.

This work was supported by U.S. NSF grant DUE#1822486.

For these and other reasons, it can be challenging for some universities' CS departments to teach their students about PDC using C/C++ and MPI. The author is a member of *CSinParallel*, an NSF-supported project to promote PDC in undergraduate CS education [6]. Since 2012, this project has sponsored over twenty faculty development workshops around the U.S.; most of these workshops included introductions to MPI programming using C/C++. At virtually every one of these workshops, one or more faculty participants has asked, "*Is there any way to do this in Java? My department is heavily Java-oriented.*"

B. OpenMPI

Those in the OpenMPI project seem to have heard similar comments. Since version 1.7 in 2013 (the current version is 4.1), OpenMPI has included Java bindings that supersede those of older 3rd party efforts such as *mpiJava* [7] or *MPJ Express* [8].

OpenMPI's Java bindings are not enabled by default; if one downloads and installs an OpenMPI binary package for Linux, MacOS, or Windows, the Java bindings are not usually enabled.

To use the Java bindings in OpenMPI, one must currently configure, build, and install OpenMPI from its source code. The following steps illustrate the process:

1. Have the Java SDK and a C compiler (e.g., GNU's **`gcc`**) already installed.
2. Download the current stable OpenMPI source code package (e.g., *openmpi-4.1.1.tar.gz*) from www.open-mpi.org.
3. Extract the OpenMPI source code from the package, e.g.:

```
tar -xcvf openmpi-4.1.1.tar.gz
```

4. Configure OpenMPI to enable the Java bindings:

```
cd openmpi-4.1.1
./configure --prefix=/usr/local \
   --enable-mpi-java --disable-mpi-fortran
```

   (If one has a Fortran compiler already installed, that final **`disable-mpi-fortran`** switch may be omitted).

5. Build and install the OpenMPI library from its source code:

```
make all install
```

This sequence of instructions should work on any Unix-derivative system, such as *Linux* or *MacOS*, or on a Unix-derivative subsystem such as *Cygwin* [17] for Windows or the *Windows Subsystem for Linux* (WSL) [13].

When these steps have been successfully completed, OpenMPI's binary, include, and library files will have been installed in the directory **`/usr/local/`**. In particular:

- an MPI compiler-script named **`mpijavac`** , and
- an MPI run-time launcher named **`mpirun`**

will have been installed in **`/usr/local/bin/`**. If that directory is present in one's environment's **`PATH`** variable, then the programs **`mpijavac`** and **`mpirun`** may be invoked from anywhere on one's system. The **`mpijavac`** program is used to compile Java programs using the OpenMPI bindings; the **`mpirun`** program is used to execute the resulting Java class files. We will illustrate their uses in Section II.

C. Parallel Design Patternlets

In software engineering, *design patterns* are repeatable, general solutions to commonly occurring problems. The seminal work on such patterns is found in the 1994 book *Design Patterns* [9]. However, the patterns described therein are limited to sequential computing, so in 2004, Mattson, Sanders, and Massingil published *Patterns for Parallel Programming* [12], which classifies and defines patterns that are useful for solving problems that commonly occur in parallel computing.

These patterns originate in the professional software engineering community, not academia. That is, software engineers have been writing parallel programs for decades; the parallel patterns stem from those decades of professional practice—they are the best practices that have been discovered through years of professional experience. Working parallel professionals think in terms of these patterns, so the more a CS educator can do to get her students to think in terms of these patterns, the more like professionals her students will be.

The MPI standard (currently 4.0) provides function calls that implement for many of the parallel patterns. From this perspective, MPI might be seen as an application programmer's interface (API) for many of the parallel design patterns.

In 2005, the author was teaching an *MPI Distributed Computing* course at what was then the *Technological University of Iceland*. The language of instruction was English, but several of the author's students were foreign-exchange students who were not fluent in English. However, these students were proficient at reading C code, so for each of the basic parallel patterns, the author wrote a minimalist, working MPI program to demonstrate that pattern to his students. These programs illustrated the correct MPI syntax for each pattern, and by comparing the program's source code against the output it produced, the non-English-speaking students could deduce the parallel behavior produced by the MPI code.

These minimalist programs proved to be very successful at helping students understand MPI functionality, so the author continued to develop such pattern-illustrating programs for other platforms (e.g., OpenMP, Pthreads, etc.). Since each program provided a minimalist representation of a parallel pattern, the author named such programs *patternlets*. A 2014 experiment indicated that replacing traditional lectures on parallel loops with live demos of parallel loop patternlets improved student engagement and understanding of that concept [2].

The patternlets collection has continued to grow; the full collection is publicly available via the author's Github repository [4]. A recent addition to that repository is a set of MPI patternlets using OpenMPI's Java bindings. The rest of this paper introduces these Java+MPI patternlets.

II. MPI Patternlets in Java

At the time of this writing, there are 25 Java+MPI patternlets available [4]. Each is stored in its own folder, along with a *Makefile* to simplify building it, and a *run* script that illustrates how to run it. Many of the patternlets have built-in interactive mini-exercises, which are detailed in the file's header-comment. Space limitations prevent us from presenting even a majority of these patternlets; what follows is a representative sample.

A. The Single Program Multiple Data Pattern

As noted earlier, MPI follows the *Single Program Multiple Data* (SPMD) pattern, in which a developer writes one program (the SP part); running the program spawns different instances that get different data values to process (the MD part). The *Spmd.java* patternlet shown in Figure 1 (minus the file's header-comment) is a simple way to introduce this pattern to students:

```
import mpi.*;

public class Spmd {
 public static void main(String [] args)
                          throws MPIException {
    MPI.Init(args);

    int id           = MPI.COMM_WORLD.getRank();
    int numProcesses = MPI.COMM_WORLD.getSize();
    String hostName  = MPI.getProcessorName();
    String message   = "Greetings from process #"
                       + id + " of "
                       + numProcesses + " on "
                       + hostName + "\n";
    System.out.print(message);

    MPI.Finalize();
  }
}
```

Fig. 1. *Spmd.java*

Similar to an C-MPI program, each Java-MPI program contains a call to **`MPI.Init(args)`** that uses any provided command-line arguments to launch the distributed computation. This includes creating an object named **`MPI.COMM_WORLD`** that is an instance of a class named **`Comm`**. In OpenMPI's Java bindings, **`getSize()`** and **`getRank()`** are **`Comm`** class methods, so a process invokes them on the **`MPI.COMM_WORLD`** object to discover: (i) its process id and (ii) how many processes are performing the computation. Last, the program ends with **`MPI.Finalize()`** that shuts down the distributed computation created by **`MPI.Init()`**.

To build *Spmd.java* manually, we may enter:

```
mpijavac Spmd.java
```

This compiles the *Spmd.java* source code to Java bytecode and stores that bytecode in a file named *Spmd.class*. To run that bytecode with one process, we may enter:

```
mpirun -np 1 java Spmd
```

On a computer named *turing*, this will produce:

```
Greetings from process #0 of 1 on turing
```

To run it with two processes, one can enter:

```
mpirun -np 2 java Spmd
```

On *turing*, doing so might produce either this:

```
Greetings from process #0 of 2 on turing
Greetings from process #1 of 2 on turing
```

or this:

```
Greetings from process #1 of 2 on turing
Greetings from process #0 of 2 on turing
```

To run the program with four processes on four networked machines named *node1*, *node2*, *node3*, and *node4*, whose names are stored in a text file named *hosts*, we can enter:

```
mpirun -np 4 -machinefile hosts java Spmd
```

and the output will be some variation of the following:

```
Greetings from process #2 of 4 on node3
Greetings from process #3 of 4 on node4
Greetings from process #0 of 4 on node1
Greetings from process #1 of 4 on node2
```

We are invoking OpenMPI's **`mpirun`**, so any of its standard command-line switches can be used. However, each process that **`mpirun`** launches is a Java virtual machine that runs the bytecode found in *Spmd.class*. Each process is running the same program but will get different values for its **`id`** and **`hostname`** variables, thus introducing students to the SPMD pattern.

The patternlets collection also includes minimalist examples for other basic patterns, including the Master-Worker, Parallel Loop, Message-Passing, Barrier, and other patterns. In the next subsection, we examine two Parallel Loop patternlets.

B. The Parallel Loop Patterns

Long-running programs often contain one or more program loops that consume the majority of the run-time. A Parallel Loop may be useful for reducing the run-time of such programs.

There are two forms of the Parallel Loop: a simple form that divides the loop's iterations among the processes in round-robin fashion, and a complex form that divides the loop's iterations into contiguous "chunks". Figure 2 presents a patternlet for the simpler form, with the parallel loop highlighted in red:

```
import mpi.*;

public class ParallelLoopChunksOf1 {

 public static final int REPS   = 8;
 public static final int MASTER = 0;

 public static void main(String [] args)
                           throws MPIException {
   MPI.Init(args);

   int id           = MPI.COMM_WORLD.getRank();
   int numProcs     = MPI.COMM_WORLD.getSize();
   String message   = "";

   if (numProcs <= REPS) {
      for (int i = id; i < REPS; i += numProcs) {
         message = "Process " + id
                   + " is performing iteration "
                   + i + "\n";
         System.out.print(message);
      }
    } else if (id == MASTER) {
       System.out.print("\nPlease run with "
                   + "-np less than or equal to "
                   +  REPS + "\n\n");
   }
   MPI.Finalize();
 }
}
```

Fig. 2. *ParallelLoopChunksOf1.java*

This program can be compiled by entering:

```
mpijavac ParallelLoopChunksOf1.java
```

To run the resulting Java bytecode with 1 process, we can enter:

```
mpirun -np 1 java ParallelLoopChunksOf1
```

and the traditional sequential output will be generated:

```
Process 0 is performing iteration 0
Process 0 is performing iteration 1
Process 0 is performing iteration 2
Process 0 is performing iteration 3
Process 0 is performing iteration 4
Process 0 is performing iteration 5
Process 0 is performing iteration 6
Process 0 is performing iteration 7
```

But if we run it using 2 processes:

```
mpirun -np 2 java ParallelLoopChunksOf1
```

then the output will be some variation on the following, depending on the order in which the operating system schedules processes 0 and 1 relative to one another:

```
Process 0 is performing iteration 0
Process 1 is performing iteration 1
Process 0 is performing iteration 2
Process 1 is performing iteration 3
Process 0 is performing iteration 4
Process 1 is performing iteration 5
Process 0 is performing iteration 6
Process 1 is performing iteration 7
```

With a bit of study, it can be seen that process 0 is performing the loop's even iterations (0, 2, 4, and 6) and process 1 is performing the odd iterations (1, 3, 5, and 7). With two processes, this pattern thus divides the loop's iterations so that a given process performs every other iteration.

To run the program with 4 processes, we can enter:

```
mpirun -np 4 java ParallelLoopChunksOf1
```

and the output will be a variation on the following:

```
Process 0 is performing iteration 0
Process 1 is performing iteration 1
Process 2 is performing iteration 2
Process 3 is performing iteration 3
Process 0 is performing iteration 4
Process 1 is performing iteration 5
Process 2 is performing iteration 6
Process 3 is performing iteration 7
```

That is, process 0 is performing iterations 0 and 4, process 1 is performing iterations 1 and 5, process 2 is performing iterations 2 and 6, and process 3 is performing iterations 3 and 7. With 4 processes, this pattern divides the loop's iterations so that a given process performs every 4th iteration. Generalizing, we can see that when using P processes, this pattern divides the loop's iterations so that a given process performs every P^{th} iteration. If we think of a parallel loop as an abstraction that divides the loop's iterations into "chunks" for a process to perform, then this simple version of the parallel loop gives each process a series of chunks, each of size 1; hence the name of this patternlet.

By contrast, the more complex form of the parallel loop divides the loop's iterations into contiguous "chunks" whose sizes are as equal as possible. Figure 3 provides a patternlet for this parallel loop, with the key code highlighted in red:

```
import mpi.*;

public class ParallelLoopEqualChunks {

 public static final int REPS = 8;
 public static final int MASTER = 0;

 public static void main(String [] args)
                            throws MPIException {
    MPI.Init(args);

    int id           = MPI.COMM_WORLD.getRank();
    int numProcesses = MPI.COMM_WORLD.getSize();
    int start        = 0;
    int stop         = 0;
    String message   = "";

    if (numProcesses > REPS) {
      if (id == MASTER) {
          System.out.print("\nPlease run with "
                  + "-np N less than or equal to "
                  +  REPS + "\n\n");
      }
      MPI.Finalize();
      System.exit(0);
    }

    double doubleReps = REPS;
    int chunkSize1 = (int) Math.ceil(doubleReps
                             / numProcesses);
    int chunkSize2 = chunkSize1 - 1;
    int remainder = REPS % numProcesses;

    if (remainder == 0 ||
          (remainder != 0 && id < remainder)) {
        start = id * chunkSize1;
        stop = start + chunkSize1;
    } else {
        start = (remainder*chunkSize1) +
                  (chunkSize2 * (id - remainder));
        stop = start + chunkSize2;
    }

    for (int i = start; i < stop; i++) {
        message = "Process " + id
                    + " is performing iteration "
                    + i + "\n";
        System.out.print(message);
    }

    MPI.Finalize();
  }
}
```

Fig. 3. *ParallelLoopEqualChunks.java*

This "equal chunks" parallel loop is more complex than the simpler loop because: (i) the iteration range must be divided among the P processes as evenly as possible, and (ii) each process must compute the **`start`** and **`stop`** numbers for its "chunk" of the iteration range. Unlike the simpler parallel loop, this "equal chunks" parallel loop is preferable for processing data stored in arrays because when the contiguous iteration-ranges it produces are used as array indexes, this loop will take full advantage of the caching mechanisms of modern CPUs.

This patternlet may be built and run in a manner similar to the previous examples; when run with 2 processes, it produces output that is a variation of the following:

```
Process 0 is performing iteration 0
Process 0 is performing iteration 1
Process 0 is performing iteration 2
Process 0 is performing iteration 3
Process 1 is performing iteration 4
Process 1 is performing iteration 5
Process 1 is performing iteration 6
Process 1 is performing iteration 7
```

In this form of the loop, process 0 performs the contiguous "chunk" of iterations 0-3, while process 1 performs the contiguous "chunk" of iterations 4-7. If we instead use 4 processes, this patternlet produces output such as this:

```
Process 0 is performing iteration 0
Process 0 is performing iteration 1
Process 1 is performing iteration 2
Process 1 is performing iteration 3
Process 2 is performing iteration 4
Process 2 is performing iteration 5
Process 3 is performing iteration 6
Process 3 is performing iteration 7
```

Using 4 processes, process 0 performs iterations 0 and 1; process 1 performs iterations 2 and 3, process 2 performs iterations 4 and 5, and process 3 performs iterations 6 and 7.

When the number of processes *P* divides evenly into the number of iterations *N*, each process gets an equal "chunk" of the iterations whose size is *N/P*. When this is not the case—when **`remainder ==`** *R* and *R* > 0—this pattern gives each of processes 0..*R-1* one additional iteration to perform, thus spreading the remainder iterations evenly across those first *R* processes. To illustrate, if we run this program with 3 processes, the program produces output that is a variation on the following:

```
Process 0 is performing iteration 0
Process 0 is performing iteration 1
Process 0 is performing iteration 2
Process 1 is performing iteration 3
Process 1 is performing iteration 4
Process 1 is performing iteration 5
Process 2 is performing iteration 6
Process 2 is performing iteration 7
```

With 8 iterations to divide among 3 processes, there are 2 remainder iterations, so processes 0 and 1 each get one extra iteration resulting in "chunks" of size 3, while process 2 gets a "chunk" whose size is 2.

This may be too complex for first-year students. For them, the patternlets collection also includes a much simpler "equal chunks" parallel loop that assumes the number of iterations *N* is evenly divisible by the number of processes *P*, so each process just gets a "chunk" of size *N/P*.

C. Communication and Sychronization Patterns

The patternlets also includes examples that illustrate the various communication patterns supported in MPI, including:

- *Barriers*, for synchronizing all processes.
- *Send-Receive message passing*, for both arrays of values and individual values.
- *Broadcast* for efficiently sending the same value(s) to all of a computation's processes.
- *Reduce* for efficiently combining distributed partial results into an overall result.
- *Scatter* for distributing the values of an array from one process amongst all the processes.
- *Gather* for combining distributed arrays from all processes to a single array in one process.

To illustrate, Figure 4 presents *Barrier.java* that illustrates how to synchronize all of an MPI computation's processes, with the MPI **`barrier()`** call highlighted in red:

```
import mpi.*;
import java.nio.CharBuffer;

public class Barrier {

 public static void main(String [] args)
                          throws MPIException {
   MPI.Init(args);

   int numProcs  = MPI.COMM_WORLD.getSize();
   int id        = MPI.COMM_WORLD.getRank();
   String host   = MPI.getProcessorName();

   sendReceivePrint(id, numProcs, host, "BEFORE");

//   MPI.COMM_WORLD.barrier();

    sendReceivePrint(id, numProcs, host, "AFTER");

    MPI.Finalize();
  }

  private static
  void sendReceivePrint(int id, int numProcesses,
                        String host, String position)
                            throws MPIException {
    if ( id != MASTER ) { // Worker
       CharBuffer buf = MPI.newCharBuffer(SIZE);
       String msg = "Process " + id
                    + " of " + numProcs
       //             + " on " + host
                    + " is " + position
                    + " the barrier.";
       buf.put(msg);
       MPI.COMM_WORLD.send(buf, msg.length(),
                           MPI.CHAR, 0, 0);
    } else {              // Master
       for (int i = 1; i < numProcs; ++i) {
          CharBuffer buf = MPI.newCharBuffer(SIZE);
          MPI.COMM_WORLD.recv(buf, SIZE, MPI.CHAR,
                              MPI.ANY_SOURCE,
                              MPI.ANY_TAG);
          System.out.println( buf.toString() );
       }
    }
  }

  private static final int SIZE   = 128;
  private static final int MASTER = 0;
}
```

Fig. 4. *Barrier.java*

In Figure 4, the **`main()`** method contains two calls to a method named **`sendReceivePrint()`**, one before and one after a barrier, which is initially commented out. In each of these calls, each worker process builds a unique message and sends it to the master process. As its part of the method, the master process receives and prints the messages the worker processes send it. The method includes a position parameter that a worker uses to indicate whether the method was called before or after the barrier. The header comment exercise instructs the user to build and run the program with varying numbers of processes, uncomment the barrier call, and then rerun the program.

When run with 4 processes (with the barrier commented out), the patternlet displays chaotically interleaved before-and-after output, such as this:

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

The more processes are used, the more chaotic the output. For example, an 8-process run might produce this:

```
Process 2 of 8 is BEFORE the barrier.
Process 1 of 8 is BEFORE the barrier.
Process 2 of 8 is AFTER the barrier.
Process 3 of 8 is BEFORE the barrier.
Process 5 of 8 is BEFORE the barrier.
Process 6 of 8 is BEFORE the barrier.
Process 7 of 8 is BEFORE the barrier.
Process 1 of 8 is AFTER the barrier.
Process 3 of 8 is AFTER the barrier.
Process 5 of 8 is AFTER the barrier.
Process 6 of 8 is AFTER the barrier.
Process 7 of 8 is AFTER the barrier.
Process 4 of 8 is BEFORE the barrier.
Process 4 of 8 is AFTER the barrier.
```

However, if we uncomment the barrier call, rebuild and run the program, then all 'before' and 'after' calls are cleanly separated:

```
Process 1 of 8 is BEFORE the barrier.
Process 3 of 8 is BEFORE the barrier.
Process 6 of 8 is BEFORE the barrier.
Process 7 of 8 is BEFORE the barrier.
Process 5 of 8 is BEFORE the barrier.
Process 4 of 8 is BEFORE the barrier.
Process 2 of 8 is BEFORE the barrier.
Process 1 of 8 is AFTER the barrier.
Process 2 of 8 is AFTER the barrier.
Process 4 of 8 is AFTER the barrier.
Process 7 of 8 is AFTER the barrier.
Process 6 of 8 is AFTER the barrier.
Process 5 of 8 is AFTER the barrier.
Process 3 of 8 is AFTER the barrier.
```

Barrier.java thus provides a simple way to introduce students to the barrier mechanism for synchronizing processes. Note that barrier is a *collective* communication pattern: all processes must perform it; otherwise, the processes that do perform it will deadlock, waiting for the processes that do not.

Another collective communication pattern is the *Reduction* pattern by which distributed, partial results can be combined into a total result. Figure 5 presents *Reduction.java* that, for P processes, computes and displays both P^2 and the sum $1^2 + 2^2 + \ldots + (P\text{-}1)^2 + P^2$, with the reduction steps highlighted:

```
import mpi.*;
import java.nio.IntBuffer;

public class Reduction {

 public static void main(String [] args)
                              throws MPIException {
    MPI.Init(args);

    Comm comm        = MPI.COMM_WORLD;
    int id           = comm.getRank();
    int numProcesses = comm.getSize();

    int square       = (id+1) * (id+1);
    IntBuffer squareBuf = MPI.newIntBuffer(SIZE);
    squareBuf.put(square);

    IntBuffer sumBuf = MPI.newIntBuffer(SIZE);
    comm.reduce(squareBuf, sumBuf, SIZE,
                MPI.INT, MPI.SUM, MASTER);

    IntBuffer maxBuf = MPI.newIntBuffer(SIZE);
    comm.reduce(squareBuf, maxBuf, SIZE,
                MPI.INT, MPI.MAX, MASTER);

    if (id == MASTER) {
        String squareMsg =
               "\nThe sum of the squares from 1 to "
                + numProcesses + " is "
                + sumBuf.get(0) + "\n";
        String maxMsg    =
               "The max of the squares from 1 to "
                + numProcesses + " is "
                + maxBuf.get(0) + "\n\n";
        System.out.print(squareMsg);
        System.out.print(maxMsg);
    }

    MPI.Finalize();
  }

  private static int SIZE    = 1;
  private static int MASTER  = 0;
}
```

Fig. 5. *Reduction.java*

One detail to note in Figures 4 and 5 is that where MPI's C bindings send and receive data using pointers (to scalars or arrays), OpenMPI's Java bindings use references to subclasses of Java's **`Buffer`** class (**`CharBuffer`**, **`DoubleBuffer`**, **`IntBuffer`**, etc.). To send or receive a scalar **`int`** value, one defines an **`IntBuffer`** of size 1; to transmit multiple values, one just defines a larger buffer. To interact with a buffer, one uses Java's standard **`Buffer`** methods: **`put()`**, **`get()`**, and so on.

In *Reduction.java*, each of the P processes uses its unique **`id`** number (0..*P-1*) to compute the value $(\texttt{id+1})^2$. The sum of these values will be the sum of the squares, so the patternlet performs a reduction using the **`MPI.SUM`** operator to sum these values in parallel. The maximum of these values will be P^2, so the patternlet performs a reduction using the **`MPI.MAX`** operator to find the maximum of these values in parallel. In each case, the master process collects and displays the result.

To illustrate, if we run this program using 1 process, it displays:

```
The sum of the squares from 1 to 1 is 1
The max of the squares from 1 to 1 is 1
```

If we run it using 3 processes, it displays:

```
The sum of the squares from 1 to 3 is 14
The max of the squares from 1 to 3 is 9
```

If we run it using 4 processes, it displays:

```
The sum of the squares from 1 to 4 is 30
The max of the squares from 1 to 4 is 16
```

If we run it using 32 processes, it displays:

```
The sum of the squares from 1 to 32 is 11440
The max of the squares from 1 to 32 is 1024
```

This patternlet thus illustrates two different ways to use MPI's reduce operation to combine partial results distributed across multiple processes into an overall result at one process.

Another important collective communication pattern is the *Broadcast*, by which one process can efficiently send the same value(s) to all processes. Figure 6 presents *Broadcast2.java* that demonstrates how to broadcast multiple values:

```
import mpi.*;
import java.nio.IntBuffer;

public class Broadcast2 {

 public static void main(String [] args)
                        throws MPIException {
   MPI.Init(args);

   Comm comm         = MPI.COMM_WORLD;
   int id            = comm.getRank();
   IntBuffer buf     = MPI.newIntBuffer(SIZE);

   if ( id == MASTER ) { fill(buf); }

   printBuffer("BEFORE", id, buf);
   printSeparator("----", id, comm);

   comm.bcast(buf, buf.capacity(), MPI.INT, 0);

   printBuffer("AFTER", id, buf);

   MPI.Finalize();
 }

 private static void fill(IntBuffer buf) {
   for (int i = 0; i < buf.capacity(); ++i) {
         buf.put(i, i+11);
   }
 }

 private static void
 printBuffer(String label, int id, IntBuffer buf) {
   String msg = label
               + " the broadcast, proc " + id
               + "'s buffer contains:";
   for (int i = 0; i < buf.capacity(); ++i) {
      msg += (" " + buf.get(i));
   }
   msg += "\n";
   System.out.print(msg);
 }

 public static
 void printSeparator(String sep, int id, Comm comm)
                        throws MPIException {
   comm.barrier();
   if (id == MASTER) { System.out.println(sep); }
   comm.barrier();
 }

 private static final int MASTER = 0;
 private static final int SIZE   = 8;
}
```

Fig. 6. *Broadcast2.java*

If we run *Broadcast2* using 4 processes, it might display:

```
 BEFORE the broadcast, proc 3's buffer contains:
0 0 0 0 0 0 0 0
 BEFORE the broadcast, proc 0's buffer contains:
11 12 13 14 15 16 17 18
 BEFORE the broadcast, proc 2's buffer contains:
0 0 0 0 0 0 0 0
 BEFORE the broadcast, proc 1's buffer contains:
0 0 0 0 0 0 0 0
 ----
 AFTER the broadcast, proc 0's buffer contains:
11 12 13 14 15 16 17 18
 AFTER the broadcast, proc 2's buffer contains:
11 12 13 14 15 16 17 18
 AFTER the broadcast, proc 1's buffer contains:
11 12 13 14 15 16 17 18
 AFTER the broadcast, proc 3's buffer contains:
11 12 13 14 15 16 17 18
```

This allows students to clearly see the effects of a broadcast: before the broadcast, each worker's buffer contains all zeros; afterwards, its buffer contains the values sent by the master.

D. Pedagogical Uses

Figures 1-6 present just 6 of the 25 patternlets currently available in the patternlets GitHub repository [4]. Space limitations prevent us from presenting additional examples, but each patternlet is designed to help students understand the syntax for and the subtle semantic details of a particular parallel pattern. The pedagogical advantages of the patternlets include:

- Their minimalist nature eliminating (most) extraneous details, helping students grasp the pattern's key concept.
- Instructors can use the patternlets in different ways, ranging from embedding them in closed lab exercises where the students run and experiment with them on their own, to the instructor running a patternlet during a lecture to illustrate a particular concept in an interactive, engaging way.
- Instructors and/or students can "play" with the code to see the effects of changing key parameters, such as the number of processes used, the number of iterations of a loop, etc.
- Students' *"What if…?"* questions can be answered by altering the source code to see the effects of a modification.
- Students can use a patternlet's syntactically correct, working code as a model when writing their own code.

Patternlets have proven successful for introducing both faculty (in *CSinParallel* workshops) and students (in courses) to PDC.

III. Conclusions

Previously, instructors wanting to teach their students PDC concepts using standard MPI were limited to the standard-supported languages: Fortran, C, and C++. Recognizing the popularity of Java, OpenMPI has added Java support to their implementation of the MPI standard; this paper has shown how to configure and install OpenMPI with Java support enabled.

Professional software engineers—whether sequential or parallel programmers—think in terms of programming patterns. The more that CS instructors can do to help their students grasp and incorporate these patterns into their own thought processes, the closer to professionals the students will be. *Patternlets* provide a simple mechanism for introducing students (or instructors) to many of the parallel patterns. Each patternlet focuses on a particular parallel pattern to help a student grasp the essential nature of that pattern and see the syntax needed to implement the pattern.

Previously, the MPI patternlets were only available in C/C++, limiting their utility to instructors and students who were reasonably proficient in those languages. By providing Java versions of the MPI patternlets, instructors and students who are proficient in Java have a new way to introduce and explore PDC concepts without having to learn a new language.

It is important to note that the patternlets are designed to *introduce* students to parallel patterns. That is, a patternlet generally avoids using its pattern to solve a significant problem, so that a student can learn the pattern's core concept in a way that minimizes the student's cognitive load. For students to see *why a particular pattern is important*, we recommend that instructors follow a patternlet-based introduction with an *exemplar*—a non-trivial MPI program in which the pattern is used to solve a socially-relevant problem—so that the students see why the pattern is important by seeing it "in action" in the context of solving a significant problem [3].

There are a variety of exemplars available from the NSF-supported site *CSinParallel.org*, but these are currently mostly written in C or C++. The information presented in this paper may be used to create Java versions of those or other exemplars; the author encourages the interested reader: (i) to create such Java exemplars, (ii) to submit them to the *CSinParallel.org* site for inclusion, and (iii) to publish the results in a venue such as the one in which this paper appears. The author looks forward to reading about such work in the future.

References

[1] ABET, *The Accreditation Board for Engineering and Technology*. Online, accessed 2021-11-01, http://abet.org.

[2] J. Adams. "Patternlets: A Teaching Tool for Introducing Students to Parallel Design Patterns," *Journal of Parallel and Distributed Computing*, April 2017, 105 (2017), pp. 31-41, doi: 10.1016/j.jpdc.2017.01.008.

[3] J. Adams, R. Brown and E. Shoop, "Patterns and Exemplars: Compelling Strategies for Teaching Parallel and Distributed Computing to CS Undergraduates," *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 2013, pp. 1244-1251, doi: 10.1109/IPDPSW.2013.275.

[4] J. Adams, E. Shoop, *The Patternlets Github Repository*. Online, accessed 2021-11-01, https://github.com/joeladams/patternlets/.

[5] Argonne National Labs, *MPICH*. Online, accessed 2021-11-01. https://www.mpich.org/.

[6] R. Brown, E. Shoop, J. Adams, S. Matthews, *CSinParallel: Parallel Computing in the Computer Science Curriculum*. Online, accessed 2021-11-01, https://csinparallel.org/index.html.

[7] B. Carpenter, et al, *mpiJava Home Page*. Online, accessed 2021-11-01, http://www.hpjava.org/mpiJava.html.

[8] B. Carpenter, et al, *MPJ Express*. Online, accessed 2021-11-01, http://mpjexpress.org.

[9] E. Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994. Addison Wesley.

[10] W. Gropp, E. Lusk, and T. Sterling, *Beowulf Cluster Computing with Linux* (Sec. Ed.), Nov. 2003, MIT Press.

[11] Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Association for Computing Machinery, New York, NY, USA.

[12] T. Mattson, et al., *Patterns for Parallel Programming*. 2004. Addison-Wesley.

[13] Microsoft Corp, *Windows Subsystem for Linux*. Online, accessed 2021-11-01, https://docs.microsoft.com/en-us/windows/wsl/.

[14] MPI Forum, *MPI Documents*. Online, accessed 2021-11-01, https://www.mpi-forum.org/docs/.

[15] S. Prasad, et al., *NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates*. Online, accessed 2021-11-01, https://tcpp.cs.gsu.edu/curriculum/.

[16] Software in the Public Interest, *OpenMPI: Open Source High Performance Computing*. Online, accessed 2021-11-01, https://www.open-mpi.org/.

[17] C. Vinschen, et al, *Cygwin: Get that Linux Feeling on Windows*. Online, accessed 2021-11-01, https://www.cygwin.com.