OpenMPI+Java as a High Performance Language

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

***Abstract*—The Message Passing Interface (MPI) is a software platform that can utilize the parallel capabilities of most multi-processors. This makes it useful for teaching students about parallel and distributed computing (PDC). MPI provides 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 MPI that also provides Java bindings, allowing instructors who know Java but not C/C++ or Fortran to teach PDC. However, Java has a reputation as a "slow" language, so some say it is unsuitable for teaching PDC. This paper gives a head-to-head comparison of the performance of OpenMPI's Java and C bindings. Our study shows that by default, Java can be faster than C unless one takes special measures, and it exhibits similar speedup, efficiency, and scalability. We conclude that Java is a suitable language for teaching PDC.**

Keywords—computing, distributed, education, exemplar, Java, MPI, OpenMPI, parallel, performance

I. Introduction

Multiprocessors are ubiquitous in today's world:

- Virtually all of today's commodity desktop and laptop computers have multicore central processing units (CPUs), making them *shared-memory multiprocessors*.
- Any modern computer lab can be configured as a Network of Workstations (NoW) *distributed-memory multiprocessor* with the lab's computers serving as computing nodes.
- A Beowulf cluster [8] is a dedicated, *distributed-memory multiprocessor* made from commodity, off-the-shelf computing nodes that communicate through a standard network fabric, such as Ethernet. Its nodes may be expensive high-end computers or inexpensive single-board computers, such as the Raspberry Pi.
- Nearly all modern supercomputers are dedicated, *heterogenous distributed-memory multiprocessors*, where a high-performance network connects nodes consisting of high-performance CPUs and accelerators.

To ensure that CS graduates can program such machines, the *NSF/IEEE TCPP Curriculum Initiative* [13] and the *ACM/IEEE CS 2013 Curriculum* guidelines [9] recommend that all undergraduate CS majors learn about PDC. Likewise, the *Accreditation Board for Engineering and Technology* (ABET) requires that all CS majors in ABET-accredited CS programs be exposed to PDC [1]. CS educators are thus expected to teach their students how to program multiprocessors.

A. The Message Passing Interface (MPI)

The *Message Passing Interface* (MPI) [12] 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 virtually 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 cores of a shared-memory multiprocessor or the computing nodes of a distributed-memory multiprocessor. MPI uses the *Single Program, Multiple Data* (SPMD) pattern of parallelism, in which copies of the same program are launched as processes on the multiprocessor's cores or 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* [15]—plus commercial versions.

The MPI standard [12] specifies that an implementation of MPI 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 slowly declining as new languages provide more-attractive alternatives. 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 language of instruction.
- 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.

For these and other reasons, it can be challenging for some universities' CS departments to teach their students about PDC using MPI and C/C++. The author is a member of *CSinParallel*,

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

an NSF-supported project to promote PDC in undergraduate CS education. 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 of the participants has asked, "*Is there any way to do this in Java? My department is heavily Java-oriented.*"

B. OpenMPI

The OpenMPI project seems to have heard such comments, as OpenMPI introduced Java bindings in version 1.7 in 2013 (the current version is 4.1). These bindings supersede those of older 3rd-party efforts such as *mpiJava* [6] or *MPJ Express* [7].

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 unlikely to be enabled.

To use the Java bindings in OpenMPI, one must currently configure, build, and install OpenMPI from its source code. In a previous paper [3], we have shown the five steps needed to do this, so we will not repeat them here, but we refer the interested reader to that paper. Those steps should work on any Unix-family system, such as *Linux* or *MacOS*, or on a Unix-family subsystem such as *Cygwin* [16] for Windows or the *Windows Subsystem for Linux* [11].

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 any folder 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. Previous Work

A *patternlet* [2] is a minimalist, complete, working program that students or instructors can run to study the behavior of a particular parallel design pattern [10]. By running a patternlet, viewing its output, and comparing that output to the (minimalist) source code that produced the output, students can see the essence of the pattern in a way that minimizes cognitive load.

[3] illustrated OpenMPI's Java bindings by showing 6 of the 25 OpenMPI+Java patternlets. Each patternlet is ideal for introducing students to a parallel design pattern, as it provides working syntax that implements and shows the behavior of the pattern. But for students to see *why* a particular pattern is useful, they need to see it being used to solve a significant problem. Such problems are called *exemplars* [4].

This paper differs from [3] by: (i) presenting an exemplar problem, (ii) solving it using MPI in both C and Java, and (iii) comparing those solutions' performances. The presentation of the C and Java solutions and the comparison of their relative performances are the primary contributions of the paper.

II. Quinn's Circuit-Solver Problem

In Quinn's classic PDC text [14], he presents a 16-bit circuit and then poses the problem of writing a program that outputs:

a. All of the 16-bit inputs that cause the circuit to output the bit 1; and
b. A count of the number of inputs that cause the circuit to output the bit 1. (This provides an easy way to check the correctness of the computation.)

In keeping with good pedagogical practice, Quinn provided a sequential program for readers to use as scaffolding. This program uses a brute-force approach: a for loop iterates through the integers $0..2^{16}-1$, and for each integer, checks the circuit for that value. One can easily create a parallel version by converting that sequential loop into a parallel loop, making this an exemplar problem for the *parallel loop* design pattern.

When Quinn wrote his text, sequentially iterating through all 2^{16} possible inputs took long enough to motivate the use of parallelism. But CPUs have improved since then; modern CPUs can solve the 16-bit problem using Quinn's sequential code in a few seconds, reducing the motivation for parallelization. To remotivate his students, this author developed a 32-bit version of the problem; its circuit is shown in Figure 1:

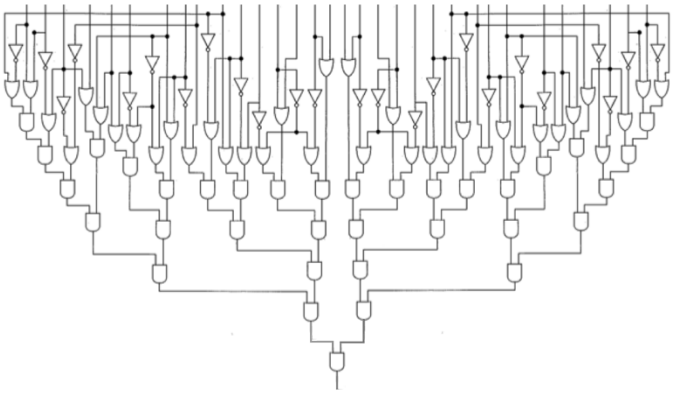

Fig. 1. A 32-bit Version of Quinn's Circuit-Solver Problem

A. A Sequential C Program

Fig. 2 presents the **`main()`** function of a sequential C program that solves the problem for the 32-bit circuit in Fig. 1:

```
int main (int argc, char *argv[]) {
   int  id    = 0;        // process id
   int  count = 0;        // number of solutions

   printf ("\nChecking the circuit...\n";

   for (long i = 0; i <= UINT_MAX; ++i) {
      count += checkCircuit(id, i);
   }

   printf("\n%d solutions found.\n", count);

   return 0;
}
```

Fig. 2. The main() Function of *circuitSolver.c* (Sequential Version)

The for loop in Fig. 2 iterates through the range of values $0..2^{32}-1$, invoking **`checkCircuit()`** on each value.

Fig. 3 presents a C implementation of **checkCircuit()**:

```
#define SIZE 32

// Extract bit i from int value n
#define EXTRACT_BIT(i, n) ( (n & (1<<i) ) ? 1 : 0)

int checkCircuit(int id, long value) {
   int v[SIZE];  // Each v[i] is one of the 32 bits

   for (long i = 0; i < SIZE; i++) {
     v[i] = EXTRACT_BIT(i, value);
   }

   if ( ( (v[0] || v[1])
       && (!v[1] || !v[3]) && (v[2] || v[3])
       && (!v[3] || !v[4]) && (v[4] || !v[5])
       && (v[5] || !v[6]) && (v[5] || v[6])
       && (v[6] || !v[15]) && (v[7] || !v[8])
       && (!v[7] || !v[13]) && (v[8] || v[9])
       && (v[8] || !v[9]) && (!v[9] || !v[10])
       && (v[9] || v[11]) && (v[10] || v[11])
       && (v[12] || v[13]) && (v[13] || !v[14])
       && (v[14] || v[15]) )
       && ( (v[16] || v[17]) && (!v[17] || !v[19])
       && (v[18] || v[19])
       && (!v[19] || !v[20]) && (v[20] || !v[21])
       && (v[21] || !v[22]) && (v[21] || v[22])
       && (v[22] || !v[31]) && (v[23] || !v[24])
       && (!v[23] || !v[29]) && (v[24] || v[25])
       && (v[24] || !v[25]) && (!v[25] || !v[26])
       && (v[25] || v[27]) && (v[26] || v[27])
       && (v[28] || v[29]) && (v[29] || !v[30])
       && (v[30] || v[31]) ) )
   {
      printf("%d) %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d\
%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d \n",
              id, v[31], v[30], v[29], v[28],
                  v[27], v[26], v[25], v[24],
                  v[23], v[22], v[21], v[20],
                  v[19], v[18], v[17], v[16],
                  v[15], v[14], v[13], v[12],
                  v[11], v[10], v[9], v[8],
                  v[7], v[6], v[5], v[4],
                  v[3], v[2], v[1], v[0] );
      fflush (stdout);
      return 1;
   } else {
      return 0;
   }
}
```

Fig. 3. The checkCircuit() Function of *circuitSolver.c*

The **checkCircuit()** function first uses the **EXTRACT_BIT()** macro to create **v**, a bit-vector of length 32 representing the parameter **value**. The if statement's condition encodes the Boolean logic of the circuit shown in Figure 1; if that condition is true, the function outputs the process **id** and the bit-vector **v**, and then returns 1; otherwise, it returns 0. Out of the 2^{32} different 32-bit inputs, just 81 of the values cause this circuit to output 1.

To compute the speedup requires a baseline time. Since the for loop in Fig. 2 is the "hot spot" in the computation, the baseline time can be computed by wrapping that loop in calls to **MPI_Wtime()** and then calculating the difference of the calls. The **MPI_Wtime()** function must be called between calls to **MPI_Init()** and **MPI_Finalize()**, as shown in Fig. 4:

```
#include <mpi.h>

int main (int argc, char *argv[]) {
   int  id    = 0;           // process id
   int  count = 0;         // number of solutions

   printf ("\nChecking the circuit...\n", id);

   MPI_Init(&argc, &argv);

   double startTime = MPI_Wtime();

   for (long i = 0; i <= UINT_MAX; ++i) {
      count += checkCircuit (id, i);
   }

   double totalTime = MPI_Wtime() - startTime;

   printf("\n%d solutions in time %f secs.\n",
            count, totalTime);

   MPI_Finalize();

   return 0;
}
```

Fig. 4. The main() Method of *circuitSolver.c* (Timed Sequential Version)

The program in Fig. 4 can then be compiled by entering:

```
mpicc -Wall -std=c99 circuitSolver.c \
      -o circuitSolver
```

This creates a binary executable program and stores it in a file named **circuitSolver** that can be run by entering:

```
mpirun -np 1 ./circuitSolver
```

Running the program produces output like the following:

```
Checking the circuit...
0) 10011001111101011001100111110101
0) 10011001111101011001100111110110
0) 10011001111101011001100111110111
0) 10011001111101011001101111110101
0) 10011001111101011001101111110110
... 75 similar lines omitted ...
0) 10011101111101111001110111110111

81 solutions found in 136.579118 secs.
```

For this paper, we compiled and ran all of the programs using MPI installed over: (i) Apple gcc (clang 13.0) on a 2022 MacBook Pro with Apple's 10-core M1 Pro CPU (whose 8 performance cores run at 3.2 GHz), and (ii) gcc 7.1 on a Linux workstation with a 3.6 GHz 8-core Intel i7 CPU. The results we present in Section III were similar on both platforms; in this paper we report the times from the MacBook Pro, since it has more cores.

As can be seen above, the program took over 2 minutes to identify the 81 solutions to the circuit. This lengthy time provides strong motivation to explore a parallel solution.

B. A Parallel OpenMPI+C Program

With a baseline time for the sequential version, the next task is to 'parallelize' the program in Fig. 4 by converting its sequential loop into a parallel loop. Fig. 5 shows a relatively simple way to do this, using the "slices" version of the *parallel loop* pattern, with some of the key differences from Fig. 4 highlighted in blue:

```
int main (int argc, char *argv[]) {
   int  id         = 0;  // process id
   int  numProcs   = 0;  // number of processes
   int  localCount = 0;  // solutions for process p
   int  totalCount = 0;  // total solutions

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

   if (id == 0) {
      printf("\nChecking using %d processes...\n",
                numProcs);
   }
   MPI_Barrier(MPI_COMM_WORLD);

   double startTime = MPI_Wtime();

   for (long i = id; i <= UINT_MAX; i += numProcs) {
      localCount += checkCircuit (id, i);
   }

   MPI_Reduce(&localCount, &totalCount, 1,
              MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

   double totalTime = MPI_Wtime() - startTime;

   if (id == 0) {
      printf("\n%d solutions found in %f secs.\n\n",
                total_count, totalTime);
   }

   MPI_Finalize();

   return 0;
}
```

Fig. 5. The main() Function of *circuitSolver.c* (Parallel Version)

Fig. 5 uses the "slices" version of the *parallel loop* pattern rather than the "equal chunks" *parallel loop* because: (i) "slices" is much simpler, (ii) "slices" may balance the workloads better if some circuits take longer to check than others, and (iii) "equal chunks" offers no performance advantage for this problem.

In Fig. 5, each **`printf()`** is guarded by an if statement that ensures only process 0 performs that output operation. However, we do not guard the **`printf()`** in function **`checkCircuit()`** (see Fig. 3), so each process outputs the solutions it discovers. The identification of the 81 solutions is thus spread across different processes, making their ordering non-deterministic, but the output produced by this parallel program has a similar overall structure to that of the sequential program:

```
Checking using 8 processes...
7) 10011001111101011001100111110111
7) 10011001111101011001101111110111
7) 10011001111101011001110111110111
6) 10011001111101011001100111110110
6) 10011001111101011001101111110110
... 75 similar lines omitted ...
5) 10011011111101111001110111110101

81 solutions found in 19.904344 secs.
```

This problem is almost embarrassingly parallel; the only inter-process communication in Fig. 5 occurs after the parallel loop, where **`MPI_Reduce()`** is used to combine the process's **`localCount`** values into the **`totalCount`**. As a result, this program exhibits nearly linear scaling as we increase the number of processes, as we will see in Section III.

C. A Sequential Java Program

Converting the sequential C code in Fig. 2 from C to Java is fairly straightforward, as can be seen in Fig. 6:

```
public class CircuitSolver {
 public static void main (String args[]) {
   int  id             = 0;           // process id
   int  count          = 0;           // solutions
   final long UINT_MAX = 4294967295L; // 2^32 - 1

   System.out.printf("\nChecking the circuit...\n");

   for (long i = 0; i <= UINT_MAX; ++i) {
      count += checkCircuit(id, i);
   }

   System.out.printf("%d solutions found.\n", count);
 }
 // ... method checkCircuit() omitted to save space
}
```

Fig. 6. The main() Method of *CircuitSolver.java* (Sequential Version)

Unlike C, Java has no **`unsigned`** primitive type, so it has no predefined constant **`UINT_MAX`**. We therefore define our own **`UINT_MAX`** using the 'hardwired' value 4,294,967,295 (2^{32}-1).

Fig. 6 omits the definitions of the **`checkCircuit()`** method, as converting the C function in Fig. 3 into a Java method is fairly straightforward.

To time this Java computation, we can use the **`MPI`** class's **`Init()`**, **`wtime()`**, and **`Finalize()`** methods, as shown in Fig. 7:

```
public static void main (String args[])
                              throws MPIException {
   MPI.Init(args);
   int  id             = 0;           // process id
   int  count          = 0;           // solutions
   final long UINT_MAX = 4294967295L; // 2^32 – 1

   System.out.printf("\nChecking the circuit...\n");

   double startTime = MPI.wtime();

   for (long i = 0; i <= UINT_MAX; ++i) {
      count += checkCircuit(id, i);
   }

   double totalTime = MPI.wtime() - startTime;

   String fmt = "\n%d solutions found in %f secs.\n";
   System.out.printf(fmt, count, totalTime);

   MPI.Finalize();
 }
```

Fig. 7. The main() Method of *CircuitSolver.java* (Timed Sequential Version)

The program in Fig. 7 can then be compiled using **`mpijavac`**:

```
mpijavac CircuitSolver.java
```

This creates a Java bytecode program and stores it in the file *CircuitSolver.class*. That file can then be run using **`mpirun`**:

```
mpirun -np 1 java CircuitSolver
```

This launches the Java Virtual Machine (JVM), which then runs the bytecode program in the *CircuitSolver.class* file. The output is identical to the sequential C version, except for the time, which we found very surprising, as we discuss in Section III.

D. A Parallel OpenMPI+Java Program

With a baseline time for one process, the next task is to convert the sequential program in Fig. 7 into a parallel program using OpenMPI's Java bindings. Fig. 8 shows the result, with the key changes highlighted in blue:

```
public static void main (String args[])
                              throws MPIException {

   MPI.Init(args);
   Comm comm           = MPI.COMM_WORLD;
   int  id             = comm.getRank();
   int  numProcs       = comm.getSize();
   int  localCount     = 0;
   int  totalCount     = 0;
   final long UINT_MAX = 4294967295L;

   if (id == 0) {
      System.out.printf("\nChecking the circuit "
                        + "with %d processes...\n",
                        numProcs);
   }
   comm.barrier();

   double startTime = MPI.wtime();

   for (long i = id; i <= UINT_MAX; i += numProcs) {
      localCount += checkCircuit(id, i);
   }

   IntBuffer localCountBuffer = MPI.newIntBuffer(1);
   localCountBuffer.put(localCount);

   IntBuffer totalCountBuffer = MPI.newIntBuffer(1);

   comm.reduce(localCountBuffer, totalCountBuffer,
               1, MPI.INT, MPI.SUM, 0);

   totalCount = totalCountBuffer.get(0);

   double totalTime = MPI.wtime() - startTime;

   String fmt = "\n%d solutions found in %f secs.\n";
   if (id == 0) {
      System.out.printf(fmt, totalCount, totalTime);
   }

   MPI.Finalize();
}
```

Fig. 8. The main() Method of *CircuitSolver.java* (Parallel Version)

An `MPI.Init()` call launches a multi-process computation of ***N*** processes, using the command-line argument `-np` ***N*** of `mpirun`. That call also creates an object named `MPI.COMM_WORLD` that can be thought of as the set of those ***N*** processes. In OpenMPI's Java bindings, `MPI.COMM_WORLD` is an instance of a class named `Comm`, and `getRank()` and `getSize()` are `Comm` class methods. Each process uses these methods to discover: (i) its process id and (ii) how many processes are performing the computation. These values are then used to implement the same "slices" *parallel loop* pattern used in Fig. 5. OpenMPI's Java bindings for the reduction operation use strongly typed buffer objects, so after the parallel loop completes, we build capacity-1 buffers for the `localCount` and `totalCount` values. We then use those buffers in the reduction operation, which OpenMPI's Java bindings provide via a `Comm` class method named `reduce()`. Following the call to `reduce()`, we retrieve the `totalCount` value from its buffer for subsequent reporting.

III. Performance Assessment

The author has taught high performance computing for more than two decades, and has for many years used Quinn's Circuit-Solver problem to introduce students to MPI, C, and the simpler "slices" *parallel loop* pattern. The problem is accessible to most students; the multi-minute runtime of the sequential version, plus the near-linear scaling lets parallel coimputing novices viscerally experience the benefits of parallel execution.

The author implemented the Java solution in Figs. 6-8 to see how much of a performance penalty one would incur for using Java instead of C. After all, the command:

```
mpirun -np N java CircuitSolver
```

launches ***N*** JVMs, each running the Java bytecode program in *CircuitSolver.class*. How could a JVM interpreting a bytecode program possibly be competitive with a native-binary executable program running directly on the hardware?

Imagine the author's surprise—and consternation—to find that not only was Java competitive; it was significantly faster! On the author's M1-equipped MacBook Pro, running the C version with one process took about 137 secs to find the 81 solutions; the Java corresponding version took about 60 secs. From these baseline times, each version's execution times decreased smoothly as more processes were used, until the M1's cores were oversubscribed, as can be seen in Fig. 9:

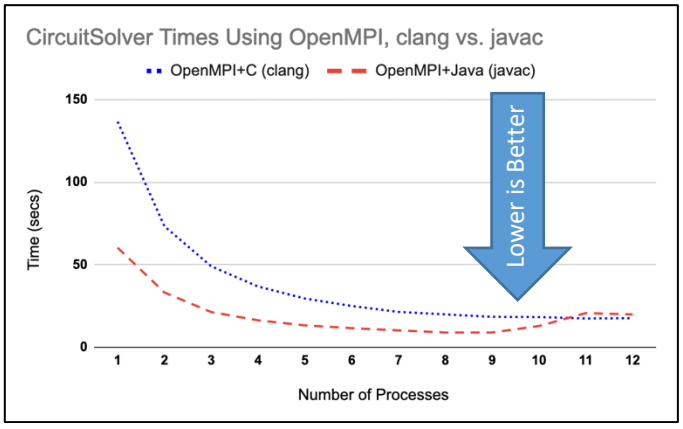

Fig. 9. Runtimes of *circuitSolver.c* and *CircuitSolver.java*

The author was rather vexed by this outcome, as it was the *exact opposite* of what was expected. This result was so counter to his intuition, he re-ran each program multiple times, both on his laptop and on an Intel i7-equipped Linux workstation. While the precise execution times were different, the pattern was the same on each platform: the OpenMPI+Java version was more than twice as fast as the OpenMPI+C version.

After spending time pondering how this could occur, the author recalled that the JVM incorporates "HotSpot" technology that at runtime, identifies *hot spots*—portions of code that are being executed repeatedly—such as the for loop in Fig. 6. When it identifies such a hot spot, the JVM performs a just-in-time (JIT) compilation to build a highly optimized version of that hot-spot, and then seamlessly switches to run that optimized code.

The key phrase in that sentence is "*highly optimized*"—it led to a hypothesis: that the difference in performance between the OpenMPI+C and the OpenMPI+Java versions was the result of *differing levels of optimization*.

More precisely, recall from Section II that we built the OpenMPI+C version with the following command:

```
mpicc -Wall -std=c99 circuitSolver.c \
      -o circuitSolver
```

Since this command does not specify any optimization switches, it is invoking the C compiler that underlies **`mpicc`** and having it use its default (i.e., minimal) optimization settings.

By contrast, once the JVM has identified the hot spot in *CircuitSolver.class* and compiled a native version of it, it switches to that JIT-compiled, highly optimized native code. Our hypothesis is that the performance difference seen in Fig. 9 stems from running an unoptimized binary (the C version) against a JIT-optimized binary (the Java version).

To test this hypothesis, we rebuilt the OpenMPI+C version using the following modified command:

```
mpicc -Wall -std=c99 -O3 circuitSolver.c \
      -o circuitSolver
```

The **`-O3`** switch tells the compiler underlying **`mpicc`** (i.e., gcc or clang) to generate highly optimized code. We then re-ran our computations; Fig. 10 shows the results:

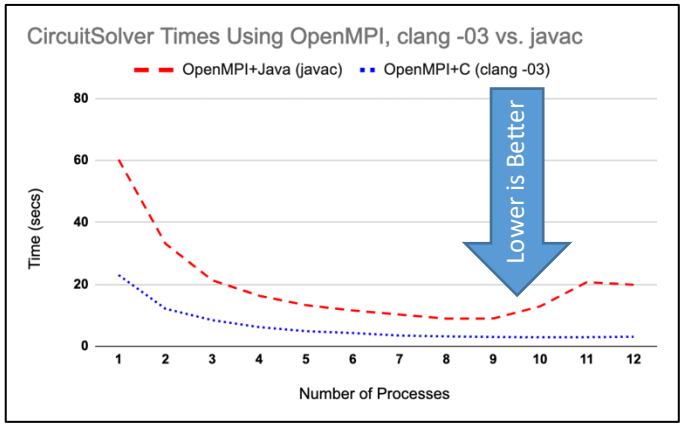

Fig. 10. Runtimes of *circuitSolver.c* (-O3 Optimized) and *CircuitSolver.java*

Comparing Fig. 10 to Fig. 9, we see that the highly optimized C version has now leapfrogged the HotSpot-optimized Java version. For example, where the single-process Java version takes about 60 secs to find the 81 instances (the same as before), the highly optimized C version has gone from about 137 secs to about 23 secs. From their baselines, both versions exhibit good scaling as we increase the number of processes, though we see diminishing returns after all of the M1 chip's eight performance cores are in use. (Apple's M1 Pro CPU has two 1.2 GHz efficiency cores and eight 3.2 GHz performance cores; MacOS initially schedules a process on the efficiency cores; if it is computationally intensive, the system migrates that process to a performance core.) The difference in performance between the C and Java versions narrows as more processes are used, until the chip's eight performance cores are oversubscribed. Beyond that point, Java's performance degrades much more than C's.

For the sake of completeness, we rebuilt and reran the OpenMPI+C version twice more: once with **`-O2`** optimization and then again with **`-O1`** optimization. The results of the **`-O2`** optimization were very similar to Fig. 10, but the results of the **`-O1`** optimization were quite different, as seen in Fig. 11:

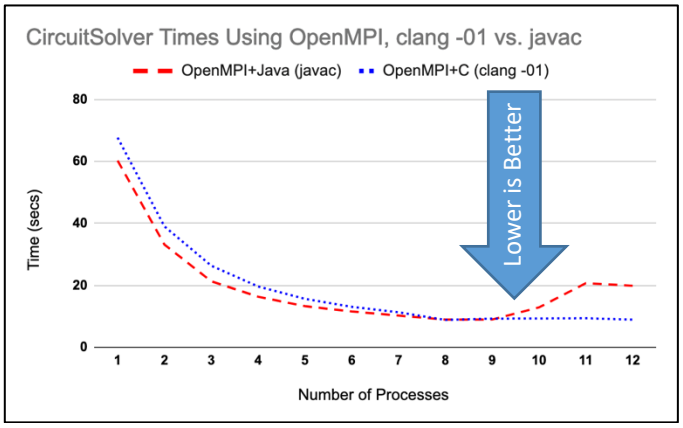

Fig. 11. Runtimes of *circuitSolver.c* (-O1 Optimized) and *CircuitSolver.java*

Fig. 11 shows that prior to the M1 chip's performance cores becoming fully saturated, the Java version slightly outperforms and tracks closely with the **`-O1`** optimized C version.

From Figures 10 and 11, it is evident that HotSpot is doing more optimization than **`-O1`** level gcc optimization, but either: (i) it is doing less than **`-O2`** or **`-O3`**, or (ii) is optimizing at a **`-O2`** or **`-O3`** level but some of the performance-gain being offset or lost elsewhere in the execution. The latter is a strong possibility; Fig. 12 shows the timeline of the Java execution:

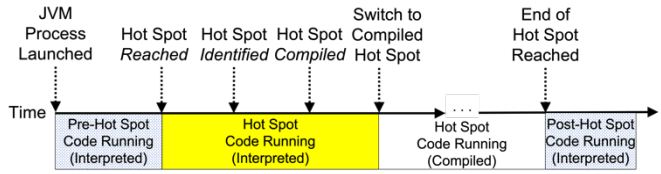

Fig. 12. Execution Timeline of HotSpot-Optimized *CircuitSolver.java*

This program has just one hot spot, represented by the second and third boxes in Fig. 12. (The first and last boxes represent executions of untimed, interpreted, non-optimized bytecode.) The second box represents timed, interpreted, non-optimized bytecode; the third box is timed, optimized binary native code. Since interpreted, bytecode runs much slower than highly optimized native code, the time taken to (i) recognize the hot spot, (ii) compile it, and (iii) switch to the compiled binary contribute to the hot spot's overall execution time. The time spent in the second box thus reduces the performance gains achieved by the third box. We have no tools to find the time spent in the second box, but it should be of a fixed length, so the longer the hot spot's overall time, the higher the percentage of time will be spent in that third (better-performing) box.

Java's HotSpot technology thus has a significant positive effect: its optimizations make Java faster than equivalent C code optimized at the **`-O1`** level; C must be optimized at higher levels to beat it. This implies that our hypothesis is correct: different optimization levels are causing the differences seen in Fig. 9.

IV. Conclusions

Historically, instructors wanting to use MPI to teach their students PDC concepts were limited to languages named in the MPI standard: Fortran, C, and C++. This is no longer the case, now that OpenMPI has added Java bindings to their implementation of the MPI standard.

In this paper, we have presented an exemplar problem, shown C and Java sequential solutions to that problem, and shown how to turn those sequential solutions into parallel solutions using OpenMP's C and Java bindings. Comparing the performance of those parallel solutions, we have seen that OpenMPI+Java performs surprisingly well: for a CPU-bound computation like our exemplar, the OpenMPI+Java solution performs better than the OpenMPI+C solution compiled at the default or `-O1` optimization levels. While a highly optimized C version outperforms the Java version, we have also seen that the performance gap between the two versions narrows as more parallel processes are used. The Java version exhibits speedup, efficiency, and scalability that are similar to the C version, making it a suitable language for teaching these concepts.

We intentionally chose a CPU-bound computation to compare the performance of OpenMPI+C and OpenMPI+Java, because a computation that exercises the CPU's arithmetic logic unit lets us directly compare the two languages' number-crunching capabilities. We expected that Java's performance would be far inferior to that of C, but we were wrong. Thanks to its HotSpot technology, Java performed much better than unoptimized or lightly optimized C, but was surpassed by highly optimized C, most likely due to a portion of the Java bytecode program being interpreted in the JVM. The HotSpot technology is included in the JVMs from both Oracle Java and OpenJDK Java; the work presented in this paper used OpenJDK Java 17.

The only inter-process communication needed to solve this problem was the reduction operation. With respect to MPI's communication operations (e.g., send, receive, broadcast, reduce, scatter, gather, etc.), we hypothesize that OpenMPI's C and Java bindings both use the same underlying core functionality—the language bindings should essentially be APIs to access that core functionality. If this is indeed the case, then a given MPI communication operation should exhibit similar performance regardless of the language used to invoke it. We hope to test this hypothesis in a future paper.

Finally, the clean syntax of OpenMPI's Java bindings combined with the JVM's HotSpot technology make Java a reasonable choice as a language for teaching PDC / HPC topics such as speedup, efficiency, and scalability. For instructors or departments with Java expertise, OpenMPI+Java provides a very good software platform for giving students practical, hands-on learning experiences to improve their understanding of abstract PDC concepts.

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, "Teaching Parallel and Distributed Computing Concepts Using OpenMPI and Java", *2021 IEEE 28th International Conference on High Performance Computing, Data and Analytics Workshop* (HiPCW), Dec. 2021, pp. 4-11. . DOI=10.1109/HiPCW54834.2021.00008.

[4] 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.

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

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

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

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

[9] 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.

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

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

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

[13] 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/.

[14] M. Quinn, *Parallel Programming In C With MPI And OpenMP*, McGraw-Hill, 2003.

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

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