Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java

Alan Kaminsky
Rochester Institute of Technology
Department of Computer Science
Rochester, NY 14623 USA
ark@cs.rit.edu

Abstract

Parallel Java is a parallel programming API whose goals are (1) to support both shared memory (thread-based) parallel programming and cluster (message-based) parallel programming in a single unified API, allowing one to write parallel programs combining both paradigms; (2) to provide the same capabilities as OpenMP and MPI in an object oriented, 100% Java API; and (3) to be easily deployed and run in a heterogeneous computing environment of single-core CPUs, multi-core CPUs, and clusters thereof. This paper describes Parallel Java's features and architecture; compares and contrasts Parallel Java to other Java-based parallel middleware libraries; and reports performance measurements of Parallel Java programs.

1. Introduction

Three trends are converging to move parallel computing out of its traditional niche of scientific computations programmed in Fortran or C. First, parallel computing is becoming of interest in other domains that need massive computational power, such as graphics, animation, data mining, and informatics; but applications in these domains tend to be written in newer languages like Java. Second, Java is becoming the main programming language students learn; recognizing this trend, the ACM Java Task Force has recently released a collection of resources for teaching introductory programming in Java [1]. Third, even desktop personal computers are now using multicore CPU chips. In other words, today's desktop PCs are shared memory multiprocessor (SMP) parallel computers, and desktop applications will need to use SMP parallel programming techniques to take full advantage of the PC's hardware. Thus, APIs are needed for parallel programming in Java in domains other than scientific computing.

In the scientific computing arena, parallel programs are generally written either for SMPs or for clusters. Reinforcing this dichotomy are separate standard libraries—OpenMP [16] for thread-based shared memory parallel programming on multi-CPU SMP machines, MPI [13] for process-based message passing parallel programming on clusters of single-CPU machines. However, it will soon be impossible to build a cluster of single-CPU machines, since every machine will come with a multicore CPU chip or chips. While parallel programs for such "hybrid SMP cluster" machines can be written using the process-based message passing paradigm, sending messages between different processes' address spaces on the same SMP machine often yields poorer performance than simply sharing the same address space among several threads. A hybrid SMP cluster parallel program should use the shared memory paradigm for parallelism within each SMP machine and should use the message passing paradigm for parallelism between the cluster machines [17]. Yet there are no standard libraries, let alone standard Java libraries, that combine the shared memory and message passing paradigms in a single API.

Parallel Java (PJ) [9, 10] was developed in response to these trends. With features inspired both by OpenMP and by MPI, PJ is a unified shared memory and message passing parallel programming library written in 100% Java. Using the same PJ API one can write parallel programs in Java for SMP machines, clusters, and hybrid SMP clusters. PJ also includes its own middleware for managing a queue of PJ jobs on a cluster and launching processes on the cluster machines.

This paper is organized as follows. Section 2 describes the features of the PJ API for SMP parallel programming, cluster parallel programming, and hybrid SMP cluster parallel programming. Section 3 describes the architecture of the PJ middleware. Section 4 compares and contrasts PJ

1-4244-0910-1/07/$20.00 ©2007 IEEE.

to other Java-based parallel middleware libraries. Section 5 reports performance measurements of PJ programs. Section 6 concludes with status and future plans.

2. Parallel Java API

To illustrate the PJ API, we will use Floyd's algorithm for finding all shortest paths in an N-node graph. The input is an $N \times N$ distance matrix **D**, where D_{rc} is the distance from node r to node c if the nodes are adjacent or is ∞ otherwise. On output, D_{rc} is the length of the shortest path from node r to node c if there is a path between the nodes or is ∞ otherwise. Floyd's algorithm is:

```
for $i = 0$ to $N-1$
    for $r = 0$ to $N-1$
        for $c = 0$ to $N-1$
            $D_{rc}$ = min ($D_{rc}$, $D_{ri} + D_{ic}$)
```

2.1. SMP Programming

In a parallel version of Floyd's algorithm designed to run on an SMP parallel computer, the distance matrix will be located in shared memory. The outer loop on i cannot be parallelized because of sequential dependencies from one iteration to the next. The nested inner loops on r and c, however, can be parallelized. Here is the computational core of the PJ SMP program (the complete program is available in the PJ distribution [10]).

```
static double[][] d;
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      for (int ii = 0; ii < n; ++ ii)
         {
         final int i = ii;
         execute (0, n-1, new IntegerForLoop()
            {
            public void run (int first, int last)
               {
               for (int r = first; r <= last; ++ r)
                  {
                  for (int c = 0; c < n; ++ c)
                     {
                     d[r][c] = Math.min (d[r][c],
                        d[r][i] + d[i][c]);
                     }
                  }
               }
            });
         }
      }
   });
```

The program creates a ParallelTeam object which contains a number of hidden threads. The number of threads can be specified on the Java command line; the default is one thread for each CPU in the SMP machine. The program then creates a ParallelRegion object and tells the parallel team to execute the parallel region. Each team thread calls the parallel region's `run()` method, and the Java virtual machine (JVM) schedules the threads to run simultaneously, each on its own processor. On each iteration of the outer loop on i, each thread creates an IntegerForLoop object and executes it. This object provides a work-sharing parallel loop over the index range 0 through $N-1$. The parallel loop objects partition the full index range among themselves and cause each team thread to perform a different subset of the iterations on r. Thus, the team threads simultaneously compute different portions of the distance matrix `d`, a shared static variable, resulting in a parallel speedup.

PJ has work-sharing parallel loop classes to iterate over an index range (`int` or `long`) using a static, dynamic, or self-guided loop schedule (these loop schedules are the same as in OpenMP). The loop schedule can be specified in the source code at compile time or as a command line flag at run time. With a static loop schedule (the default), the complete index range (0 to $N-1$ in this example) is divided into K equal-sized chunks, where K is the number of parallel team threads. Each team thread calls the IntegerForLoop object's `run()` method with the lower and upper bounds of a different chunk as the `first` and `last` arguments. In other words, each thread's subset of the loop iterations is fixed before the iterations start; this can lead to an unbalanced load and reduced performance if different iterations do different amounts of work. A dynamic or self-guided loop schedule can yield a balanced load and improved performance in such a situation. With a dynamic loop schedule, the complete index range is divided into chunks where each chunk consists of a specified number of iterations (the chunk size). With a self-guided loop schedule, the complete index range is divided into chunks whose sizes are determined automatically; each chunk consists of half the remaining number of iterations divided by K, so that chunks start out larger and become progressively smaller. With a dynamic or a self-guided loop schedule, each team thread then repeatedly gets the next available chunk and calls the IntegerForLoop's `run()` method to execute that chunk's iterations until all chunks have been executed. In this way, some threads can execute fewer long-running chunks while other threads execute more short-running chunks, so that each thread runs for about the same total time yielding a balanced load.

PJ also has parallel loop classes to iterate over the elements of an array, over the objects returned by an Iterator, and over the objects in an Iterable collection. These are analogous to the "for-each" loop added to the Java language in Java 5.

PJ has a work-sharing ParallelSectionGroup class, which contains a number of ParallelSection objects, each of which contains a section of code; each team thread executes a different parallel section simultaneously. PJ supports single sections (executed by only one team thread) and critical sections protected by exclusive or nonexclusive locks.

Variables in a PJ program are either shared or thread-local depending on where they are declared. Shared variables are typically static fields of the main program class, thread-local variables are typically instance fields of the parallel loop class. PJ also supports shared reduction variables with arbitrary user-defined reduction operators.

Space limitations preclude describing all of PJ's features for shared memory parallel programming here. For a complete list, refer to the PJ documentation [9].

2.2. Cluster Programming

In a parallel version of Floyd's algorithm designed to run on a cluster parallel computer, there are K processes, each running a copy of the program on its own processor. Each process has a rank between 0 and $K - 1$. The distance matrix is split among the processes, each process computes a different portion of the matrix, and the pieces are recombined. Here is the core of the PJ cluster program (considerable detail is omitted).

```
static Comm world = Comm.world();
static int rank = world.rank();
static double[][] d;
static double[] d_i;
static DoubleMatrixBuf[] rowSliceBuffers;
static DoubleMatrixBuf rowSliceBuffer;
static DoubleArrayBuf d_i_buffer;
world.scatter (0, rowSliceBuffers, rowSliceBuffer);
for (int i = 0; i < n; ++ i)
   {
   world.broadcast (i_root, d_i_buffer);
   for (int r = 0; r < rlen; ++ r)
      {
      for (int c = 0; c < n; ++ c)
         {
         d[r][c] = Math.min (d[r][c], d[r][i] + d_i[c]);
         }
      }
   }
world.gather (0, rowSliceBuffer, rowSliceBuffers);
```

d holds the distance matrix; the complete matrix in process rank 0, one slice of the matrix in the other processes. d_i holds row i of the distance matrix, received from the process in charge of row i. rowSliceBuffers is an array of buffer objects, one buffer for each process. A buffer is a "view" of a group of data items stored in some other variable, allowing those data items to be sent or received in messages. Each buffer in rowSliceBuffers refers to a different slice of the rows and all of the columns of the matrix d. rowSliceBuffer is the buffer corresponding to this process's rank. d_i_buffer is a buffer referring to the array d_i. (The code for initializing these variables and buffers is omitted.)

The program scatters the distance matrix from process 0 to all the processes by calling the scatter() method on the world communicator. rowSliceBuffers specifies the source data items in process 0 to be sent to each process, and rowSliceBuffer specifies the destination for this process's data items. The program then executes Floyd's algorithm. On each outer loop iteration, the process in charge of row i (process rank i_root) broadcasts the contents of row i to all the processes by calling the broadcast() method. Each process then executes the nested inner loop iterations for the process's slice of the rows and all the columns. Finally, the program gathers the distance matrix from all the processes back into process 0 by calling the gather() method.

PJ provides class Comm, a communicator for message passing. The world communicator encompasses all the processes in the program. Class Comm has methods for point-to-point communication operations: send, receive, and send-receive, in blocking and non-blocking versions. Class Comm also has methods for collective communication operations: broadcast, scatter, gather, all-gather, and reduce with arbitrary user-defined reduction operators. PJ provides buffer classes to act as sources or destinations for the messages' data items. By specifying the proper buffer, PJ can send and receive primitive data types as well as Java objects (using object serialization); and PJ can send and receive single variables, arrays, matrices, and portions thereof. User-defined buffer classes to send and receive other data structures can also be written.

For a complete list of PJ's message passing features, refer to the PJ documentation [9].

2.3. Hybrid SMP Cluster Programming

To get a hybrid SMP cluster version of Floyd's algorithm, simply replace the sequential for loop on r in the cluster version with the parallel loop of the SMP version. Slices of the distance matrix are still scattered to the cluster processors, which compute the slices in parallel. Within each processor (SMP machine), all threads share the distance matrix slice, and the threads compute the rows of the slice in parallel. Here is the core of the PJ hybrid SMP cluster program.

```
static Comm world = Comm.world();
static int rank = world.rank();
static double[][] d;
static double[] d_i;
static DoubleMatrixBuf[] rowSliceBuffers;
static DoubleMatrixBuf rowSliceBuffer;
static DoubleArrayBuf d_i_buffer;
world.scatter (0, rowSliceBuffers, rowSliceBuffer);
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      for (int ii = 0; ii < n; ++ ii)
         {
         final int i = ii;
         single (new ParallelSection()
            {
            public void run() throws Exception
               {
               world.broadcast (i_root, d_i_buffer);
               }
            });
         execute (0, rlen-1, new IntegerForLoop()
            {
            public void run (int first, int last)
               {
               for (int r = first; r <= last; ++ r)
                  {
                  for (int c = 0; c < n; ++ c)
                     {
                     d[r][c] = Math.min (d[r][c],
                        d[r][i] + d_i[c]);
                     }
                  }
               }
            });
         }
      }
   });
world.gather (0, rowSliceBuffer, rowSliceBuffers);
```

After scattering the initial distance matrix to all processes, the program creates a ParallelTeam object with a number of threads. The threads execute a ParallelRegion object's `run()` method simultaneously. The `run()` method contains the triply-nested loop of Floyd's algorithm. As in the cluster version of the program, on each iteration of the outer loop on i, the contents of row i must be broadcast from the process in charge of row i to all the processes. However, this time multiple threads are executing the `run()` method; but only one of the threads must do the broadcast, and the remaining threads must wait for the broadcast to finish. This is accomplished by creating a new ParallelSection object and passing it to the ParallelRegion's `single()` method. The `single()` method causes one thread to call the ParallelSection's `run()` method and perform the broadcast, and the `single()` method causes the other threads to wait until the first thread returns from the ParallelSection's `run()` method. (In OpenMP parlance, this is a "single section.") Once the broadcast is complete, all the threads proceed to execute the middle loop on r in parallel, using an IntegerForLoop as in the shared memory version of the program. At the conclusion of the triply-nested loop, the program gathers the distance matrix into process 0 as in the cluster version.

3. Parallel Java Architecture

Figure 1 shows the architecture of PJ running on a cluster parallel computer with one frontend processor and multiple backend processors connected by a high-speed network. A Job Launcher Daemon runs on each backend processor, and a Job Scheduler Daemon runs on the frontend processor. (All PJ processes are written in Java.) The Job Scheduler keeps track of each backend processor's status and maintains a queue of PJ jobs; a web interface displays the cluster status.

To run a PJ job on the cluster, the user logs into the frontend processor and uses the `java` command to launch the PJ program. This JVM becomes the job frontend process. The job frontend connects to the Job Scheduler. (All interprocess communication in PJ uses TCP sockets.) The Job Scheduler waits until the requisite number of backend processors are idle, then tells the job frontend which backend processors to use. The job frontend connects to the Job Launcher on each backend processor and tells the Job Launcher to spawn a JVM. These JVMs become the job backend processes. The job backends connect to the job frontend, obtain the program's class files and command line arguments, and call the static `main()` method of the main program class. The job frontend relays the job's standard input, standard output, and standard error streams between the job backends and the user's terminal.

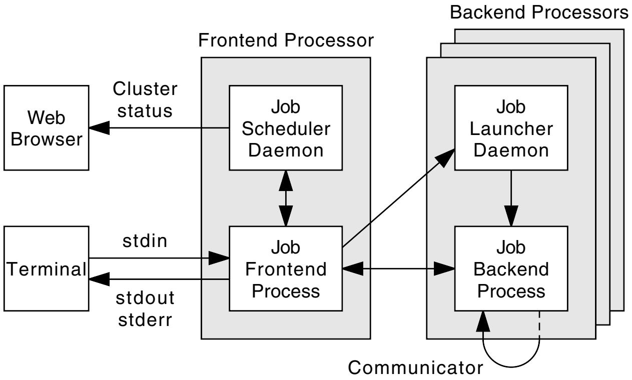

Figure 1. Parallel Java architecture

As the PJ program runs in the job backend processes, the job backends set up connections among themselves for message passing via the world communicator. Message passing is implemented using Java New I/O (NIO) direct byte buffers and socket channels. Message data items of a primitive type are transferred between the program variables and the communication byte buffers using the NIO primitive data type bulk `get()` and `put()` methods; message data items of a non-primitive type (objects) are transferred between the program variables and the communication byte buffers using Java object serialization. However, PJ uses multi-threaded blocking I/O ("Old" I/O) to send to and receive from the socket channels, rather than single-threaded selector-based I/O; timing measurements show that blocking I/O gives better performance than selector-based I/O.

Since PJ is written in 100% Java, it will run on any platform that supports Java Development Kit (JDK) 1.5.0. (PJ has been tested on Solaris and on Linux.) Installing PJ on a cluster is easy; simply run the Job Scheduler and Job Launcher daemons. Even if the cluster consists of heterogeneous machines with different CPUs and operating systems, PJ's message passing will still work due to Java's platform independence. Installing PJ on an SMP machine is even easier, as no daemons are needed; PJ programs just run like any other Java programs.

4. Related Work

JOMP [6, 7] is a Java API for thread-based SMP parallel programming. Patterned after OpenMP, JOMP uses compiler directives to insert parallel programming constructs into a regular program. In a Java program, the JOMP directives take the form of comments beginning with `//omp`. The JOMP program is run through a precompiler which processes the directives and produces the actual Java program, which is then compiled and executed. JOMP supports most features of OpenMP, including work-sharing parallel loops and parallel sections, shared variables, thread local variables, and reduction variables.

PJ, like JOMP, supports most features of OpenMP; but unlike JOMP, PJ's parallel constructs are obtained by instantiating library classes rather than by inserting precompiler directives. Thus, PJ needs no extra precompilation step. Further, since no intermediate source file is needed, debugging tools can debug PJ programs directly. In addition, Java programmers are accustomed to writing object oriented programs by instantiating classes rather than by adding precompiler directives.

Many Java versions of the MPI standard have been developed; mpiJava [5, 14] and MPJ [4] are typical. mpiJava is implemented as a thin Java native interface (JNI) wrapper on top of a native MPI library. MPJ is a 100% Java implementation of the MPI standard. Both mpiJava and MPJ adhere closely to MPI's Fortran and C oriented API; for example, each provides a Comm class with the same method names and arguments as MPI's message passing subroutines.

Like mpiJava and MPJ, PJ provides MPI-like functionality for message passing parallel programming. Also like MPJ, PJ is a 100% Java implementation that does not use a native MPI library. However, while providing MPI's *functionality,* PJ does not attempt to adhere closely to MPI's *API.* This considerably simplifies PJ's message passing methods. For example, mpiJava has two different methods for scattering, each with many arguments:

```
public void Scatter
   (Object sendbuf, int sendoffset, int sendcount,
    Datatype sendtype, Object recvbuf, int recvoffset,
    int recvcount, Datatype recvtype, int root);
public void Scatterv
   (Object sendbuf, int sendoffset, int[] sendcount,
    int[] displs, Datatype sendtype, Object recvbuf,
    int recvoffset, int recvcount, Datatype recvtype,
    int root);
```

In contrast, PJ has one method for scattering with only three arguments:

```
public void scatter
   (int root, Buf[] srcarray, Buf dst);
```

By encapsulating in a separate buffer object (class Buf) all the information about where to retrieve or store data items, PJ's one scatter method provides the same functionality as MPI's (and mpiJava's) multiple scatter routines. Again, Java programmers are accustomed to hiding these sorts of details in separate objects. Section 5 compares the performance of message passing in PJ versus mpiJava.

CCJ [15] is a Java library of MPI-like collective communication operations. CCJ provides the barrier, broadcast, scatter, gather, all-gather, reduce, and all-reduce operations (but not point-to-point communication operations like send, receive, and send-receive). Rather than implementing its own low-level communication protocol, CCJ is implemented on top of Java remote method invocation (RMI); this allows CCJ to transfer arbitrary serializable objects, not just primitive data types as in MPI. Furthermore, CCJ can scatter and gather portions of arbitrary collection objects among a group of parallel processes, as long as the collections implement CCJ's DividableDataObject interface; CCJ is not limited to scattering and gathering arrays as in MPI. For example, CCJ can scatter and gather the nodes in a linked list data structure, or the elements of a sparse matrix data structure. However, since CCJ uses RMI, CCJ's performance depends critically on the RMI layer's performance; the authors implemented CCJ using the Manta high performance Java RMI system [12] rather than the RMI layer in the standard Java platform.

PJ provides functionality similar to CCJ. PJ has MPI-like collective communication operations (although all-reduce and barrier are not implemented yet) as well as point-to-point communication operations. PJ can transfer arbitrary serializable objects. PJ can scatter and gather portions of arbitrary collection objects; this is done by defining an appropriate subclass of class Buf, the class PJ uses to retrieve or store the data elements being sent or received in a message. Unlike CCJ, PJ uses its own low-level communication protocol that is implemented using the high performance NIO direct byte buffers and socket channels in the standard Java platform. (NIO was not part of the standard Java platform in 2001 when CCJ was developed.) Also, PJ includes classes for OpenMP-like shared memory parallel programming; CCJ provides only message passing operations.

Object-Oriented SPMD (OO-SPMD) [3], an extension of the ProActive library [2], introduces an interesting, albeit unconventional, paradigm for parallel programming in Java—a paradigm based on *active objects* rather than message passing. Active objects run concurrently on the processor nodes; the active objects are associated together to form a group; asynchronous method calls are performed on the group, causing each object in the group to execute the methods; data is transferred between the active objects as arguments of the method calls. In contrast, PJ follows the conventional thread-based shared memory paradigm of OpenMP and the process-based message passing paradigm of MPI. Also, like OpenMP and MPI, PJ is targeted primar-

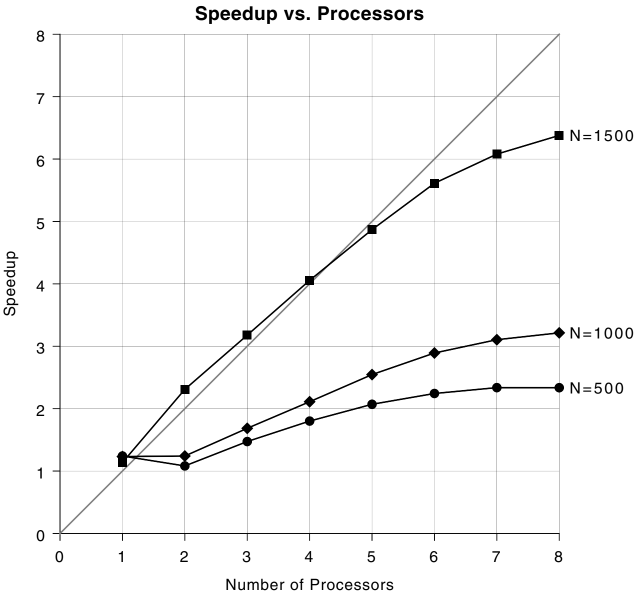

Figure 2. Floyd SMP program speedup

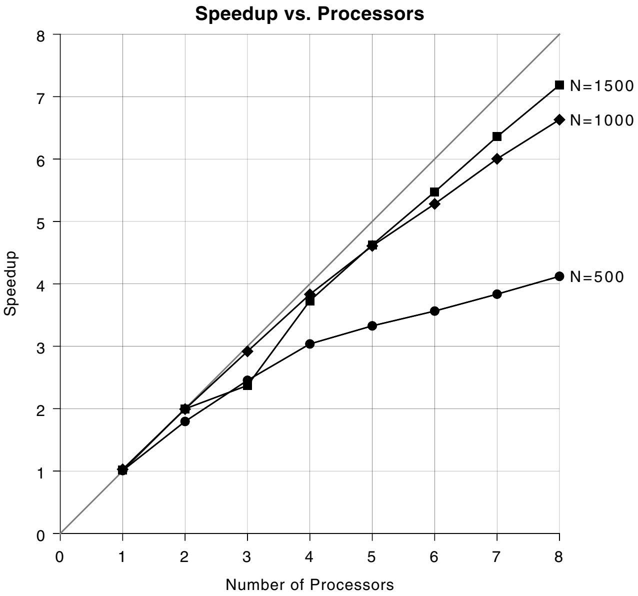

Figure 3. Floyd cluster program speedup

ily at programming *tightly coupled* parallel machines such as SMP machines and clusters with a dedicated high speed network; OO-SPMD is targeted more towards programming *loosely coupled* parallel machines such as computing grids.

Smith and Bull [17] describe how to write "mixed mode" applications using both MPI and OpenMP, what we are calling hybrid SMP cluster programs. They also survey several mixed mode projects (though none in Java). They point out that because MPI implementations are not necessarily multiple thread safe, MPI calls in an OpenMP program must be made in sequential code, such as single sections or critical sections. This introduces additional thread synchronization overhead that is not required by the application logic. In contrast, PJ's message passing constructs are multiple thread safe by design and require no extra thread synchronization when combined with PJ's shared memory parallel programming constructs. (The single section in the hybrid SMP cluster PJ program presented earlier is required by the application logic.)

Taboada et al. [18] describe the design of Java Fast Sockets, a high performance socket implementation in Java for high speed cluster interconnection networks. They identified Java NIO direct byte buffers with their primitive data type bulk `get()` and `put()` methods, NIO socket channels, and non-blocking selectors as important features for high performance Java sockets. While PJ's message passing does use NIO direct byte buffers and socket channels, PJ does not use selectors. During development, both a multi-threaded blocking version and a single-threaded selector-based version of PJ's message passing classes were developed, and the former version yielded significantly better performance measurements. (Specifically, the time to send a message was about 50% longer with the latter version.)

5. Parallel Java Performance

Figure 2 plots the measured speedup of the SMP version of the PJ program for Floyd's algorithm with respect to a sequential version of the program for 500, 1,000, and 1,500-node graphs. The two smaller problem sizes show poor speedups due to the relatively large sequential overhead of synchronizing the parallel team threads (a barrier wait) at the end of every outer loop iteration. With a 1,500-node graph there is relatively less sequential overhead, hence better speedups. (The tests were run on a Sun Microsystems UltraSPARC-IV 8-CPU SMP machine using Sun's JDK 1.5.0. Each timing measurement was the median of seven program runs.)

Figure 3 plots the measured speedup of the cluster version of the PJ program for Floyd's algorithm with respect to a sequential version of the program. Again, the smaller problem sizes show poorer speedups due to the relatively larger sequential overhead of the scatter, broadcast, and gather message passing operations. (The tests were run on a cluster of Sun Microsystems UltraSPARC-IIi Solaris workstations using Sun's JDK 1.5.0. Each timing measurement was the median of seven program runs.)

Floyd's algorithm exhibits poor scalability for smaller problem sizes because it is not a massively parallel problem; on every outer loop iteration, there has to occur a thread synchronization (SMP version) or a message broadcast (cluster version). As an example of a massively parallel problem,

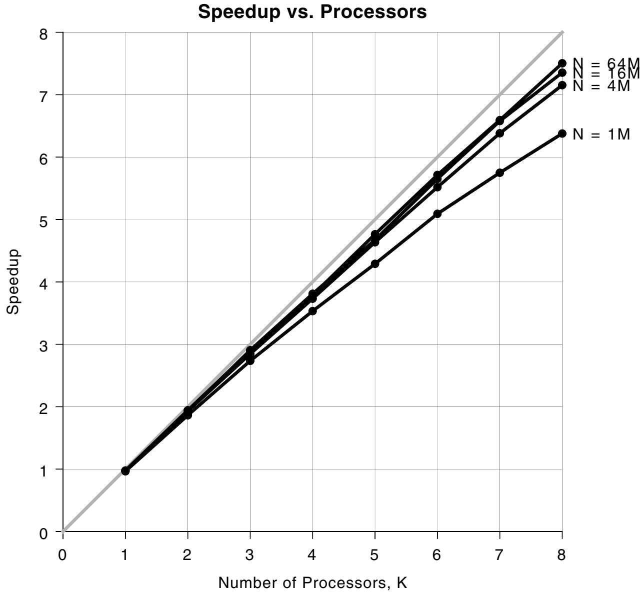

Figure 4. AES key search SMP program speedup

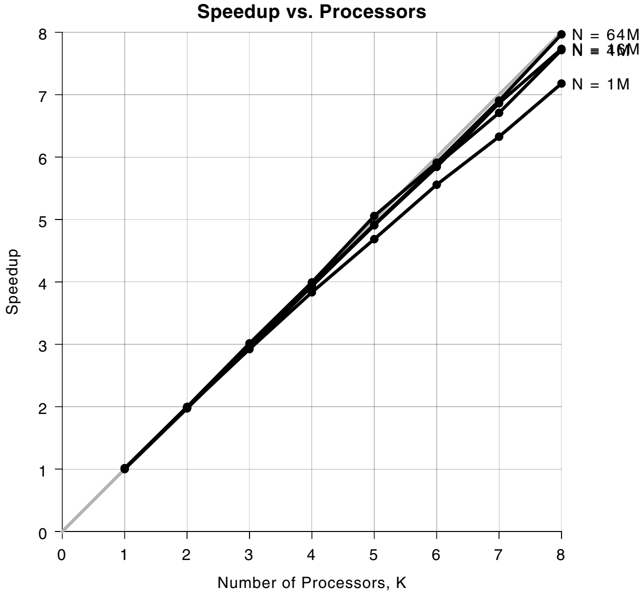

Figure 5. AES key search cluster program speedup

consider a known plaintext attack on a block cipher by an exhaustive search of a portion of the key space. The program's inputs are a plaintext block, the ciphertext block obtained by encrypting the plaintext block with a certain key, the value of the key with some of the bits missing, and n, the number of missing key bits. The program does a search over all $N = 2^n$ values of the missing key bits. For each potential key, the program encrypts the given plaintext with that key and checks whether the result equals the given ciphertext; if so, the correct key has been found. The program uses the Advanced Encryption Standard (AES) block cipher with a 128-bit block size and a 256-bit key size.

Figure 4 plots the measured speedup of the SMP version of the PJ program for block cipher key search with respect to a sequential version of the program for N = 1, 4, 16, and 64 million potential keys searched (n = 20, 22, 24, and 26 missing key bits). Figure 5 plots the measured speedup of the cluster version of the PJ program for block cipher key search with respect to a sequential version of the program for the same inputs. The key search program's scalability for this massively parallel problem is much better than the Floyd's algorithm program's scalability.

Figure 6 compares the performance of message passing in PJ versus mpiJava. A "ping-pong" program sent an N-byte message from one process to another and back again, measured the round trip time, and did 10,000 repetitions. The measured message time T was half the average of the round trip times. The message times for message sizes of N = 100, 200, 500, 1,000, 2,000, 5,000, 10,000, and 20,000 bytes are plotted, with three program runs for each message size. The lines show the linear regression of the data for N = 2,000 to 20,000 bytes. For PJ, the regression formula (correlation = 1.000) is

$$T = (4.53\times10^{-4} + 8.73\times10^{-8}N)\ \text{sec}$$

For mpiJava, the regression formula (correlation = 0.997) is

$$T = (4.61\times10^{-4} + 9.10\times10^{-8}N)\ \text{sec}$$

Thus, PJ's message passing performance is slightly better than mpiJava's, even though PJ is implemented in 100% Java and mpiJava is implemented on top of native MPI. (The tests were run on the aforementioned cluster, using Sun's MPI implementation and Sun's JDK 1.4.2 for mpiJava.)

6. Status and Future Plans

The PJ distribution is available from the author's web site [10]. The shared memory parallel programming features of PJ are largely complete. The message passing operations of PJ are partially complete; so far, the send, receive, send-receive, broadcast, scatter, gather, all-gather, and reduce operations have been implemented.

For the past two years the author has used PJ to teach a parallel programming course taken by upper division undergraduate students and by graduate students. The course covers both SMP parallel programming and cluster parallel programming in Java. Several additional example PJ programs, along with their performance measurements, are available on the course web site [8] and are included in the PJ distribution.

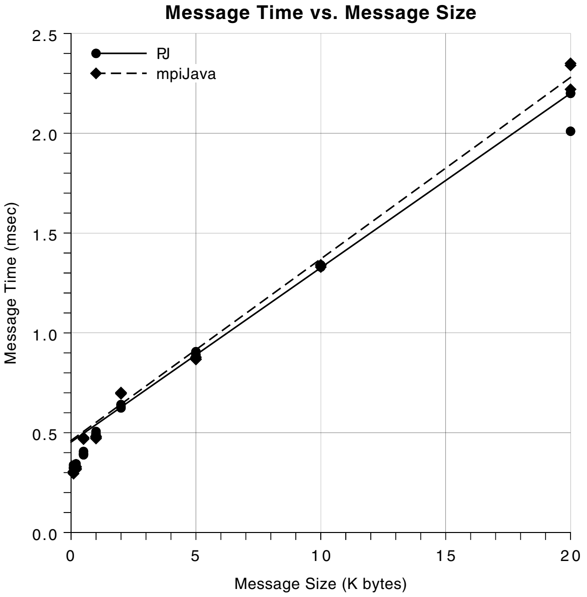

Figure 6. PJ vs. mpiJava message passing performance

Future plans for PJ include improving the performance of PJ's low-level message passing classes; improving the performance of PJ's high-level message passing (communicator) operations; implementing additional message passing operations such as all-reduce, all-to-all, and barrier; and adding parallel file I/O capabilities like those of MPI-2.0.

While the work so far has focused on implementing PJ's functionality, future work will also focus on performance measurements and comparisons. Future plans for PJ include porting standard parallel benchmark codes, such as the SPEC HPC 2002 and OMP Benchmarks, the Pallas MPI Benchmarks, and the NAS Parallel Benchmarks, to PJ and measuring their performance; comparing PJ's performance on standard benchmarks to the performance of other Java-based parallel middleware libraries such as MPJ and mpiJava; and comparing PJ's performance on standard benchmarks to the performance of Fortran-, C-, OpenMP-, and MPI-based versions.

A project in the domain of computational medicine to use parallel computing to speed up the spin relaxometry analysis of magnetic resonance images is underway. An initial parallel program was written in Java using mpiJava [11]. Future plans include rewriting the analysis program using PJ and studying different parallel analysis algorithms.

7. Acknowledgments

I would like to acknowledge my student Luke McOmber, who helped write the first version of PJ in 2005; the students in my Parallel Computing I classes in 2005 and 2006, who have helped me debug and refine the PJ library; and the anonymous referees for suggesting improvements to this paper.

References

[1] ACM Java Task Force. `http://jtf.acm.org`.

[2] L. Baduel, F. Baude, and D. Caromel. Efficient, flexible, and typed group communications in Java. In *2002 Joint ACM-ISCOPE Conference on Java Grande*, pages 28–36, 2002.

[3] L. Baduel, F. Baude, and D. Caromel. Object-oriented SPMD. In *2005 IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, pages 824–831, May 2005.

[4] M. Baker and D. Carpenter. MPJ: A proposed Java message-passing API and environment for high performance computing. In *International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 552–559, May 2000.

[5] M. Baker, D. Carpenter, G. Fox, S. Ko, and S. Lim. mpiJava: An object-oriented Java interface to MPI. In *International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999*, April 1999.

[6] J. Bull, M. Westhead, M. Kambites, and J. Obdržálek. Towards OpenMP for Java. In *Second European Workshop on OpenMP*, pages 98–105, September 2000.

[7] JOMP Home Page. `http://www.epcc.ed.ac.uk/research/jomp/index_1.html`.

[8] A. Kaminsky. Parallel Computing I course web site. `http://www.cs.rit.edu/~ark/531/`.

[9] A. Kaminsky. Parallel Java Documentation. `http://www.cs.rit.edu/~ark/pj/doc/`.

[10] A. Kaminsky. Parallel Java Library. `http://www.cs.rit.edu/~ark/pj.shtml`.

[11] A. Kaminsky and L. McOmber. Solving and MRI spin relaxometry problem with parallel computing. Technical report, Rochester Institute of Technology Department of Computer Science, June 2005. `http://www.cs.rit.edu/~ark/sr/index.shtml`.

[12] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An efficient implementation of Java's remote method invocation. In *7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, 1999.

[13] Message Passing Interface Forum. `http://www.mpi-forum.org`.

[14] mpiJava Home Page. `http://aspen.ucs.indiana.edu/pss/HPJava/mpiJava.html`.

[15] A. Nelisse, T. Kielmann, H. Bal, and J. Maassen. Object-based collective communication in Java. In *2001 Joint ACM-ISCOPE Conference on Java Grande*, pages 11–20, 2001.

[16] OpenMP Home Page. `http://www.openmp.org`.

[17] L. Smith and M. Bull. Development of mixed mode MPI / OpenMP applications. *Scientific Programming*, 9(2–3):83–98, 2001.

[18] G. Taboada, J. Touriño, and R. Doallo. Designing efficient Java communications on clusters. In *International Parallel and Distributed Processing Symposium (IPDPS 2005)*, page 182a, April 2004.