Appendix A
Clash of the Titans: C vs. Java

► Part I. Preliminaries
► Part II. Tightly Coupled Multicore
► Part III. Loosely Coupled Cluster
► Part IV. GPU Acceleration
► Part V. Map-Reduce
▼ Appendices
  Appendix A. Clash of the Titans: C vs. Java
Many folks who do high performance computing (HPC)—the kind of large-scale, scientific and engineering applications that have to run on parallel computers to get answers in a reasonable amount of time—feel that Java is not an appropriate language for HPC work. Java is for stuff like enterprise data processing, web services, and smartphone apps. For heavy-duty HPC work you need a heavy-duty language, like C.

I disagree. Let me explain why.

One objection to the use of Java for HPC is that widely-used parallel programming APIs—such as OpenMP for multicore parallel programming, MPI for cluster parallel programming, and CUDA or OpenCL for GPU parallel programming—do not support Java. Rather, they support Fortran, C, and C++. However, this doesn’t mean that you can’t do HPC in Java. The Parallel Java 2 Library (PJ2) is a counterexample. While this book’s focus is on teaching parallel programming, not on developing HPC applications, several of the programs we’ve studied did have an HPC flavor. For example:

• The $\pi$ estimating programs in Chapters 8, 22, 30, and 31 use an HPC technique called Monte Carlo integration to estimate the integral

$$\int_{0}^{1} \sqrt{1-x^2} \, d\,x$$

—that is, the area under the circle quadrant in Figure 4.1.

• The zombie programs in Chapters 12, 25, 32, and 33 solve an $N$-body problem. This kind of computation pops up in many areas of HPC. Star cluster and galaxy simulations, comet orbit predictions, and molecular modeling all use $N$-body techniques.

Another objection is that supercomputer centers, like those at the U.S. national laboratories, have already installed the Fortran, C, and C++ oriented compilers and libraries, but not the Java compiler, Java virtual machine, or PJ2. But that’s just an argument about convenience. A user can easily install the JDK and PJ2 in her account on the supercomputer and proceed to work in Java. A fellow professor did exactly that on the Lonestar and Ranger clusters in the Texas Advanced Computing Center* at the University of Texas at Austin, and used the Parallel Java Library (PJ2’s predecessor) to develop and run parallel molecular dynamics programs in Java.

Another widely-voiced objection is that Fortran, C, and C++ programs are fast, but Java programs are slow. Therefore, HPC programs, which have to wring the fastest possible performance out of the parallel computer hardware, should be written in those languages, not Java. This would be a serious objection if it were true. However, in my experience, most folks who claim

* https://www.tacc.utexas.edu/
#include <stdlib.h>
#include <stdio.h>
#include <sys/timeb.h>
#include "Prng.h"

// Print a usage message and exit.
static void usage()
{
    fprintf (stderr, "Usage: PiSeq <seed> <N>\n");
    fprintf (stderr, "<seed> = Random seed\n");
    fprintf (stderr, "<N> = Number of random points\n");
    exit (1);
}

// Print an error message and exit.
static void error (char *msg)
{
    fprintf (stderr, "PiSeq: %s\n", msg);
    usage();
}

// Main program.
int main
(int argc,
 char **argv)
{
    unsigned long long int seed, N, count, i;
    double x, y;
    prng_t prng;
    struct timeb t1, t2;

    ftime (&t1);

    // Validate command line arguments.
    if (argc != 3) usage();
    if (sscanf (argv[1], "%llu", &seed) != 1)
        error ("<seed> illegal");
    if (sscanf (argv[2], "%llu", &N) != 1)
        error ("<N> illegal");

    // Generate n random points in the unit square, count how many
    // are in the unit circle.
    count = 0;
    prng_setSeed (&prng, seed);
    for (i = 0; i < N; ++ i)
    {
        x = prng_nextDouble (&prng);
        y = prng_nextDouble (&prng);
        if (x*x + y*y <= 1.0) ++ count;
    }

    // Print results.
    printf ("pi = 4*%llu/%llu = %.9f\n", count, N, 4.0*count/N);
    ftime (&t2);
    printf ("%ld msec\n", (t2.time - t1.time)*1000 + t2.millitm - t1.millitm);
}

Listing A.1. PiSeq.c
that C is faster than Java are just repeating hearsay; they have never bothered to substantiate the claim with actual performance data.

So let’s try to substantiate the claim that C is faster than Java. I’m going to do a head-to-head comparison between the same multicore parallel program written in C with OpenMP and in Java with PJ2—a veritable “Clash of the Titans.” I’ll use the \( \pi \) estimating program from Chapter 8.

Listing A.1 is the sequential C program, PiSeq.c. It is the same as the PiSeq.java program (Listing 8.1), except it uses C idioms. Several variables are declared as the C type unsigned long long int, a 64-bit integer equivalent to the Java type long. The program does input/output with the C functions \texttt{scanf()}, \texttt{printf()}, and \texttt{fprintf()}. The Prng.h header file (line 4) defines a structure \texttt{prng_t} for a pseudorandom number generator (PRNG) as well as several functions to perform operations on a PRNG, including \texttt{prng_setSeed()} (line 45) and \texttt{prng_nextDouble()} (lines 48–49). This PRNG uses the same algorithm as Java class edu.rit.util.Random. The program also measures and reports its own wall clock running time (lines 31, 33, 55–57).

Listing A.2 is the multicore parallel C/OpenMP program, PiSmp.c. It is the same as PiSeq.c, except I added a couple of OpenMP \textit{pragmas} to indicate regions of code to be performed in parallel on multiple cores.

The first pragma (line 45) specifies that the following block of code is to be executed by a team of threads. Each thread executes the statements enclosed between the braces on lines 46 through 55. By default, the number of team threads is the same as the number of cores in the machine; but this can be overridden by calling an OpenMP function or by setting the shell environment variable \texttt{OMP_NUM_THREADS}.

The \texttt{private} clause in the pragma specifies that the variables \texttt{i}, \texttt{x}, \texttt{y}, and \texttt{prng}, which were declared at the top of the program, are to be per-thread variables. Each thread gets its own copies of these variables and can update them without needing to synchronize with the other threads. This is particularly important for the \texttt{prng} variable. Variables not listed in the \texttt{private} clause, including the WORM variables \texttt{seed} and \texttt{N}, are shared by all the threads.

The \texttt{reduction} clause in the pragma specifies that the \texttt{count} variable is a reduction variable, and the reduction operation is summation (+). Outside the parallel region, \texttt{count} refers to the global variable. Inside the parallel region, \texttt{count} refers to the per-thread copy. Line 44 initializes the global \texttt{count} variable to 0; the per-thread copies are then automatically initialized to 0 also. On line 53, each team thread increments its own per-thread \texttt{count} variable. When the program leaves the parallel region on line 55, the per-thread \texttt{count} variables are automatically reduced together using summation, and the result is placed in the global \texttt{count} variable; this then gets printed on line 58.
Appendix A. Clash of the Titans: C vs. Java

Listing A.2. PiSmp.c (part 1)

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/timeb.h>
#include "Prng.h"

// Print a usage message and exit.
static void usage()
{
    fprintf (stderr, "Usage: PiSmp <seed> <N>\n");
    fprintf (stderr, "<seed> = Random seed\n");
    fprintf (stderr, "<N> = Number of random points\n");
    exit (1);
}

// Print an error message and exit.
static void error (char *msg)
{
    fprintf (stderr, "PiSmp: %s\n", msg);
    usage();
}

// Main program.
int main
(int argc,
 char **argv)
{
    unsigned long long int seed, N, count, i;
    double x, y;
    prng_t prng;
    struct timeb t1, t2;
    ftime (&t1);

    // Validate command line arguments.
    if (argc != 3) usage();
    if (sscanf (argv[1], "%llu", &seed) != 1)
        error ("<seed> illegal");
    if (sscanf (argv[2], "%llu", &N) != 1)
        error ("<N> illegal");

    // Generate n random points in the unit square, count how many
    // are in the unit circle.
    count = 0;
    #pragma omp parallel private(i, x, y, prng), reduction(+:count)
    {
        prng_setSeed (&prng, seed + omp_get_thread_num());
        #pragma omp for
        for (i = 0; i < N; ++ i)
        {
            x = prng_nextDouble (&prng);
            y = prng_nextDouble (&prng);
            if (x*x + y*y <= 1.0) ++ count;
        }
    }

    // Print results.
    printf ("pi = 4*%llu/%llu = %.9f\n", count, N, 4.0*count/N);
```

Listing A.2. PiSmp.c (part 1)
On line 47, each team thread initializes its own PRNG with a different seed, namely the seed command line argument plus the thread’s rank in the team. The OpenMP function `omp_get_thread_num()` returns the rank. This ensures that each thread’s PRNG generates a different sequence of random numbers.

The second pragma (line 48) specifies that the iterations of the immediately following for loop are to be partitioned among the threads in the team, each thread performing a different subset of the loop iterations. In other words, the for loop is to be a work sharing parallel loop. By default, the parallel for loop uses a fixed schedule. (OpenMP calls it a “static” schedule, but static already has another meaning in Java.)

I compiled the two C programs with the Linux commands below. Like many modern C compilers, gcc recognizes OpenMP pragmas and generates the appropriate multithreaded machine code. The -O3 flag tells the compiler to optimize the machine code for high performance.

$ gcc -O3 -o PiSeq PiSeq.c
$ gcc -O3 -fopenmp -o PiSmp PiSmp.c

I ran the PiSeq and PiSmp C programs on one of the 12-core nodes of the tardis cluster and measured their running times, as I did for the PiSeq and PiSmp Java programs. I used the same problem sizes as I did for the weak scaling study in Chapter 14. Table A.1 (at the end of the appendix) lists the C programs’ performance metrics. Figure A.1 plots the C programs’ running times (from Table A.1) and the Java programs’ running times (from Table 14.2) for identical problem sizes.

So are these C programs faster than their Java equivalents? No, they are not. Figure A.1 clearly shows that when I run the C and the Java programs—which are as close to the same as I can make them—on identical inputs, the C programs’ running times are longer than the Java programs’ running times. How much longer? Examining the running time data, the C programs take 38 to 40 percent more time than the Java programs. The point is worth emphasizing:

| The C programs are 38 to 40 percent slower than the Java programs. |

This experiment refutes the conventional wisdom about Java performance.

Can we conclude, then, that Java code is actually faster than C code? No, we cannot. Too many factors affect a parallel program’s performance: the parallel computing hardware; the programming language; the parallel APIs, libraries, and middleware; the compiler; the optimizations performed by the compiler (either the C compiler, or the JVM’s just-in-time compiler); and the
nature of the computation itself—the software design, the data structures, the algorithms. I have seen some parallel programs written in C outperform their Java equivalents by a small percentage. I have seen many parallel programs written in Java outperform their C equivalents, sometimes running as much as 40 percent faster. I have concluded that when it comes to performance, no one is justified in making any blanket statement like "C is faster than Java"—or the reverse.

Why, then, is the myth that "C is faster than Java" so persistent? I think the answer lies in the mists of ancient history. Java came out in 1995, about the same time as the initial versions of MPI (1994) and OpenMP (1997). At that time, JVMs interpreted the Java bytecodes. Interpreted code does indeed run a lot slower than compiled code, and so Java got a reputation for being slow. But within a few years, JVMs with just-in-time compilers started to appear. Now, more than two decades later, a modern JVM’s just-in-time compiler converts the Java bytecodes to highly optimized machine code, which it then executes directly on the CPU. As the example in this appendix demonstrates, it is possible for the dynamic run time optimizations performed by the JVM’s just-in-time compiler to yield machine code that outperforms stati-
cally compiled and optimized C code—despite the overhead of the JVM doing the compilation at run time. But the damage to Java’s reputation was already done.

You might object that this data is for only one small parallel program. A different, larger program might run faster in C than in Java. But this is exactly my point. Change any factor—the computation, the language, whatever—and the balance might tip either way. There’s no way to predict ahead of time which language, API, library, software design, or any other factor would yield the fastest performance for any particular program. The only way to find out is to implement the program with different choices of language, API, and so on and to measure the actual performance.

If high performance really is your paramount goal, you owe it to yourself—and to your managers and your customers—to choose the parallel programming language and API that actually does yield the smallest running times. But base your choice on measured data from head-to-head comparisons, not on hearsay or conventional wisdom. If Fortran or C or C++ yields better performance for your programs, go for it. But I suspect Java would be a strong competitor, and maybe even a winner.
Table A.1. PiSmp.c weak scaling performance data

<table>
<thead>
<tr>
<th>N(1)</th>
<th>N</th>
<th>K</th>
<th>T (msec)</th>
<th>Sizeup</th>
<th>Effic.</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 billion</td>
<td>4×10^9</td>
<td>seq</td>
<td>107862</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4×10^9</td>
<td>1</td>
<td>108577</td>
<td>0.993</td>
<td>0.993</td>
</tr>
<tr>
<td></td>
<td>8×10^9</td>
<td>2</td>
<td>108655</td>
<td>1.985</td>
<td>0.993</td>
</tr>
<tr>
<td></td>
<td>12×10^9</td>
<td>3</td>
<td>109559</td>
<td>2.954</td>
<td>0.985</td>
</tr>
<tr>
<td></td>
<td>16×10^9</td>
<td>4</td>
<td>108791</td>
<td>3.966</td>
<td>0.991</td>
</tr>
<tr>
<td></td>
<td>24×10^9</td>
<td>6</td>
<td>108771</td>
<td>5.950</td>
<td>0.992</td>
</tr>
<tr>
<td></td>
<td>32×10^9</td>
<td>8</td>
<td>108734</td>
<td>7.936</td>
<td>0.992</td>
</tr>
<tr>
<td></td>
<td>48×10^9</td>
<td>12</td>
<td>108732</td>
<td>11.904</td>
<td>0.992</td>
</tr>
<tr>
<td>8 billion</td>
<td>8×10^9</td>
<td>seq</td>
<td>215703</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8×10^9</td>
<td>1</td>
<td>217148</td>
<td>0.993</td>
<td>0.993</td>
</tr>
<tr>
<td></td>
<td>16×10^9</td>
<td>2</td>
<td>217288</td>
<td>1.985</td>
<td>0.993</td>
</tr>
<tr>
<td></td>
<td>24×10^9</td>
<td>3</td>
<td>219248</td>
<td>2.951</td>
<td>0.984</td>
</tr>
<tr>
<td></td>
<td>32×10^9</td>
<td>4</td>
<td>217518</td>
<td>3.967</td>
<td>0.992</td>
</tr>
<tr>
<td></td>
<td>48×10^9</td>
<td>6</td>
<td>217434</td>
<td>5.952</td>
<td>0.992</td>
</tr>
<tr>
<td></td>
<td>64×10^9</td>
<td>8</td>
<td>217286</td>
<td>7.942</td>
<td>0.993</td>
</tr>
<tr>
<td></td>
<td>96×10^9</td>
<td>12</td>
<td>217424</td>
<td>11.905</td>
<td>0.992</td>
</tr>
<tr>
<td>16 billion</td>
<td>16×10^9</td>
<td>seq</td>
<td>431405</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16×10^9</td>
<td>1</td>
<td>434190</td>
<td>0.994</td>
<td>0.994</td>
</tr>
<tr>
<td></td>
<td>32×10^9</td>
<td>2</td>
<td>434524</td>
<td>1.986</td>
<td>0.993</td>
</tr>
<tr>
<td></td>
<td>48×10^9</td>
<td>3</td>
<td>438157</td>
<td>2.954</td>
<td>0.985</td>
</tr>
<tr>
<td></td>
<td>64×10^9</td>
<td>4</td>
<td>434675</td>
<td>3.970</td>
<td>0.992</td>
</tr>
<tr>
<td></td>
<td>96×10^9</td>
<td>6</td>
<td>434741</td>
<td>5.954</td>
<td>0.992</td>
</tr>
<tr>
<td></td>
<td>128×10^9</td>
<td>8</td>
<td>434695</td>
<td>7.939</td>
<td>0.992</td>
</tr>
<tr>
<td></td>
<td>192×10^9</td>
<td>12</td>
<td>434879</td>
<td>11.904</td>
<td>0.992</td>
</tr>
<tr>
<td>32 billion</td>
<td>32×10^9</td>
<td>seq</td>
<td>862794</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32×10^9</td>
<td>1</td>
<td>868459</td>
<td>0.993</td>
<td>0.993</td>
</tr>
<tr>
<td></td>
<td>64×10^9</td>
<td>2</td>
<td>869302</td>
<td>1.985</td>
<td>0.993</td>
</tr>
<tr>
<td></td>
<td>96×10^9</td>
<td>3</td>
<td>876346</td>
<td>2.954</td>
<td>0.985</td>
</tr>
<tr>
<td></td>
<td>128×10^9</td>
<td>4</td>
<td>870017</td>
<td>3.967</td>
<td>0.992</td>
</tr>
<tr>
<td></td>
<td>192×10^9</td>
<td>6</td>
<td>869757</td>
<td>5.952</td>
<td>0.992</td>
</tr>
<tr>
<td></td>
<td>256×10^9</td>
<td>8</td>
<td>869273</td>
<td>7.940</td>
<td>0.993</td>
</tr>
<tr>
<td></td>
<td>384×10^9</td>
<td>12</td>
<td>869829</td>
<td>11.903</td>
<td>0.992</td>
</tr>
<tr>
<td>64 billion</td>
<td>64×10^9</td>
<td>seq</td>
<td>1725521</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64×10^9</td>
<td>1</td>
<td>1736906</td>
<td>0.993</td>
<td>0.993</td>
</tr>
<tr>
<td></td>
<td>128×10^9</td>
<td>2</td>
<td>1738310</td>
<td>1.985</td>
<td>0.993</td>
</tr>
<tr>
<td></td>
<td>192×10^9</td>
<td>3</td>
<td>1752579</td>
<td>2.954</td>
<td>0.985</td>
</tr>
<tr>
<td></td>
<td>256×10^9</td>
<td>4</td>
<td>1739972</td>
<td>3.967</td>
<td>0.992</td>
</tr>
<tr>
<td></td>
<td>384×10^9</td>
<td>6</td>
<td>1739218</td>
<td>5.953</td>
<td>0.992</td>
</tr>
<tr>
<td></td>
<td>512×10^9</td>
<td>8</td>
<td>1738453</td>
<td>7.940</td>
<td>0.993</td>
</tr>
<tr>
<td></td>
<td>768×10^9</td>
<td>12</td>
<td>1739493</td>
<td>11.904</td>
<td>0.992</td>
</tr>
</tbody>
</table>