PARALLEL PROGRAMMING IN JAVA

Alan Kaminsky
Associate Professor
Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
102 Lomb Memorial Drive
Rochester, NY 14623
ark@cs.rit.edu
http://www.cs.rit.edu/~ark/

Presented at the CCSCNE 2007 Conference
April 20, 2007

Revised 22-Oct-2007
SCHEDULE AND TOPICS

9:00am    Introduction and goals
          Parallel computing architectures and middleware standards
          Shared memory multiprocessor (SMP) parallel programming

10:00am   Hands-On Exercise 1

            Break

10:30am   Cluster parallel programming

11:00am   Hands-On Exercise 2

            Break

11:30am   Further examples of parallel programs
          Hybrid SMP cluster parallel programming

12:00pm   Done

GOALS OF THE WORKSHOP

● To convince you that all computing students need to learn parallel programming.

● To demonstrate that it is possible to write high-performance parallel programs in Java—
       —Without needing to write in C or Fortran.
       —Without needing to write low-level threading code or networking code.

● To introduce Parallel Java (PJ)—
       —An API and middleware for parallel programming in 100% Java.
PARALLEL JAVA RESOURCES

- Alan Kaminsky. *Building Parallel Programs: SMPs, Clusters, and Java.*
  To be published by Thomson Course Technology, tentative publication date January 2009.
  Draft chapters:  http://www.cs.rit.edu/~ark/bpp/


BASIC PARALLEL PROGRAMMING BOOKS


- Kenneth A. Berman and Jerome L. Paul. *Algorithms: Sequential, Parallel, and Distributed.*

- Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming.*
  Addison-Wesley, 2005.


ADVANCED PARALLEL PROGRAMMING BOOKS


OpenMP RESOURCES


MPI RESOURCES

PARALLEL COMPUTER ARCHITECTURES

Shared memory multiprocessor (SMP) parallel computers
—Standard programming API: OpenMP

Cluster parallel computers
—Standard programming API: Message Passing Interface (MPI)

Hybrid SMP cluster parallel computers
—Standard programming API: ???

The RIT CS Department's hybrid SMP cluster parallel computer
● Frontend computer — tardis.cs.rit.edu — UltraSPARC-Ile CPU, 650 MHz clock, 512 MB main memory
● 10 backend computers — dr00 through dr09 — each with two AMD Opteron 2218 dual-core CPUs, four processors, 2.6 GHz clock, 8 GB main memory
● 1-Gbps switched Ethernet backend interconnection network
● Aggregate 104 GHz clock speed, 80 GB main memory
PROBLEM: COMPUTING $\pi$

Monte Carlo technique for computing an approximate value of $\pi$:

- The area of the unit square is 1
- The area of the circle quadrant is $\pi/4 = 0.78540$
- Generate a large number of points at random in the unit square
- Count how many of them fall within the circle quadrant, i.e. distance from origin $\leq 1$
- The fraction of the points within the circle quadrant gives an approximation for $\pi/4$
- 4 times this fraction gives an approximation for $\pi$
public class PiSeq
{
    // Prevent construction.
    private PiSeq()
    {
    }
    import edu.rit.pj.Comm;
    import edu.rit.util.Random;
    /**
     * Class PiSeq is a sequential program that calculates an approximate value for
     * $\pi$; using a Monte Carlo technique. The program generates a number of random
     * points in the unit square (0,0) to (1,1) and counts how many of them lie
     * within a circle of radius 1 centered at the origin. The fraction of the
     * points within the circle is approximately $\pi/4$.
     * <p>
     * Usage: java PiSeq <I>seed</I> <I>N</I>
     * <BR><I>seed</I> = Random seed
     * <BR><I>N</I> = Number of random points
     * <p>
     * The computation is performed sequentially in a single processor. The program
     * uses class edu.rit.util.Random for its pseudorandom number generator. The
     * program measures the computation's running time. This establishes a benchmark
     * for measuring the computation's running time on a parallel processor.
     * *
     * @author Alan Kaminsky
     * @version 21-Dec-2006
     */
// Program shared variables.

// Command line arguments.
static long seed;
static long N;

// Pseudorandom number generator.
static Random prng;

// Number of points within the unit circle.
static long count;

// Main program.

/**
 * Main program.
 */
public static void main
(String[] args)
throws Exception
{
Comm.init (args);

// Start timing.
long time = -System.currentTimeMillis();

// Validate command line arguments.
if (args.length != 2) usage();
seed = Long.parseLong (args[0]);
N = Long.parseLong (args[1]);

// Set up PRNG.
prng = Random.getInstance (seed);

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

// Stop timing.
time += System.currentTimeMillis();

// Print results.
System.out.println
("pi = 4 * " + count + "/ " + N + " = " + (4.0 * count / N));
System.out.println (time + " msec");
}

// Hidden operations.

/**
 * Print a usage message and exit.
 */
private static void usage()
{
System.err.println ("Usage: java PiSeq <seed> <N>");
}
System.err.println("<seed> = Random seed");
System.err.println("<N> = Number of random points");
System.exit(1);
import edu.rit.pj.Comm;
import edu.rit.pj.ParallelTeam;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.reduction.SharedLong;
import edu.rit.util.LongRange;
import edu.rit.util.Random;

/**
 * Class PiSmp is an SMP parallel program that calculates an approximate value
 * for π; using a Monte Carlo technique. The program generates a number of
 * random points in the unit square (0,0) to (1,1) and counts how many of them
 * lie within a circle of radius 1 centered at the origin. The fraction of the
 * points within the circle is approximately π/4.
 * <p>
 * Usage: java -Dpj.nt=<I>K</I> PiSmp <I>seed</I> <I>N</I>
 * <BR><I>K</I> = Number of parallel threads
 * <BR><I>seed</I> = Random seed
 * <BR><I>N</I> = Number of random points
 * <p>
 * The computation is performed in parallel in multiple threads. The program
 * uses class edu.rit.util.Random for its pseudorandom number generator. To
 * improve performance, each thread has its own pseudorandom number generator,
 * and the program uses the reduction pattern to determine the count. The
 * program uses the "sequence splitting" technique with the pseudorandom number
 * generators to yield results identical to the sequential version. The program
 * measures the computation's running time.
 * <p>
 * @author Alan Kaminsky
 * @version 21-Dec-2006
 */
public class PiSmp
{
    private PiSmp()
    {
    }

    // Program shared variables.

    // Command line arguments.
    static long seed;
    static long N;

    // Number of points within the unit circle.
    static SharedLong count;

    // Main program.

    /**
     * Main program.
     */
    public static void main(String[] args)
        throws Exception
    {
        Comm.init(args);

        // Start timing.
        long time = -System.currentTimeMillis();

        // Validate command line arguments.
        if (args.length != 2) usage();
        seed = Long.parseLong(args[0]);
        N = Long.parseLong(args[1]);

        // Generate n random points in the unit square, count how many are in
        // the unit circle.
        count = new SharedLong(0L);
        new ParallelTeam().execute(new ParallelRegion()
        {
            public void run()
            {
                // Set up per-thread PRNG and counter.
                Random prng_thread = Random.getInstance(seed);
                long count_thread = 0L;

                // Determine range of iterations for this thread.
                LongRange range_thread =
                    new LongRange(0, N-1)
                        .subrange(getThreadCount(), getThreadIndex());
                long N_thread = range_thread.length();

                // Skip PRNG ahead over the random numbers the lower-indexed
                // threads will generate.
                prng_thread.skip(2 * range_thread.lb());

                // Generate random points.
                for (long i = 0L; i < N_thread; ++i)
                {
double x = prng_thread.nextDouble();
double y = prng_thread.nextDouble();
if (x*x + y*y <= 1.0) ++ count_thread;
}

// Reduce per-thread counts into shared count.
count.addAndGet (count_thread);
}
}

// Stop timing.
time += System.currentTimeMillis();

// Print results.
System.out.println
("pi = 4 * " + count + " / " + N + " = " +
(4.0 * count.doubleValue() / N));
System.out.println (time + " msec");
}

// Hidden operations.
/**
 * Print a usage message and exit.
 */
private static void usage()
{
    System.err.println ("Usage: java -Dpj.nt=<K> PiSmp <seed> <N>");
    System.err.println ("<K> = Number of parallel threads");
    System.err.println ("<seed> = Random seed");
    System.err.println ("<N> = Number of random points");
    System.exit (1);
}
}
HANDS-ON EXERCISE 1A
π SMP PROGRAM PERFORMANCE

1. Log into the Macintosh workstation if necessary. Username: student Password: student
3. Start a terminal window.
4. Remote login to the Tardis parallel computer. Password: 1ebd_13bf_3c70_4195
   ssh paraconf@tardis.cs.rit.edu
5. Change into your working directory. Replace dir with your email name.
   cd dir
6. Run the sequential π program. The first argument is the random seed. The second argument
   is the number of iterations. Try different seeds. Try different numbers of iterations. Notice
   the running times.
   java PiSeq 142857 10000000
7. Run the SMP parallel π program. The number after -Dpj.nt= is the number of parallel
   threads. You can specify from 1 to 4 threads. Try different numbers of iterations and
   different numbers of threads. Notice the running times.
   java -Dpj.nt=1 PiSmp 142857 10000000
8. Pick a number of iterations that yields a running time of about 30 seconds for the sequential
   program. Run the sequential program three times and record the running times on the next
   page. Run the parallel program three times each with 1, 2, 3, and 4 threads and record the
   running times. Calculate and record the median running time, speedup, and efficiency for
   each trial.
   Speedup = (Median sequential time) / (Median parallel time)
   Efficiency = (Speedup) / (Number of threads)
9. What do you notice about the speedup and efficiency as the number of threads increases?
10. Do the same measurements as Step 8 for larger numbers of iterations. What do you notice
    about the speedup and efficiency as the number of iterations increases?
### HANDS-ON EXERCISE 1A (cont.)

Random seed: ________________  Number of iterations: ________________

<table>
<thead>
<tr>
<th># of threads</th>
<th>T1 (msec)</th>
<th>T2 (msec)</th>
<th>T3 (msec)</th>
<th>Median T</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Random seed: ________________  Number of iterations: ________________

<table>
<thead>
<tr>
<th># of threads</th>
<th>T1 (msec)</th>
<th>T2 (msec)</th>
<th>T3 (msec)</th>
<th>Median T</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Random seed: ________________  Number of iterations: ________________

<table>
<thead>
<tr>
<th># of threads</th>
<th>T1 (msec)</th>
<th>T2 (msec)</th>
<th>T3 (msec)</th>
<th>Median T</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A three-dimensional random walk is defined as follows. A particle is initially positioned at (0, 0, 0) in the X-Y-Z coordinate space. The particle does a sequence of \(N\) steps. At each step, the particle chooses one of the six directions left, right, ahead, back, up or down at random, then moves one unit in that direction. Specifically, if the particle is at \((x, y, z)\):

With probability 1/6 the particle moves left to \((x-1, y, z)\);
With probability 1/6 the particle moves right to \((x+1, y, z)\);
With probability 1/6 the particle moves back to \((x, y-1, z)\);
With probability 1/6 the particle moves ahead to \((x, y+1, z)\);
With probability 1/6 the particle moves down to \((x, y, z-1)\);
With probability 1/6 the particle moves up to \((x, y, z+1)\).

Write a sequential program and an SMP parallel program to calculate the particle's final position. The program's command line arguments are the random seed and the number of steps \(N\). The program prints the particle's final position \((x, y, z)\) as well as the distance of the particle from the origin.

1. Log into the Macintosh workstation if necessary. Username: student Password: student
3. Start a terminal window.
4. Remote login to the Tardis parallel computer. Password: lebd_13bf_3c70_4195
   ssh paraconf@tardis.cs.rit.edu
5. Change into your working directory. Replace dir with your email name.
   cd dir
6. Use your favorite terminal editor to write the Java source file for the sequential program. Suggested class name: WalkSeq File name: WalkSeq.java
   vi WalkSeq.java
7. Compile the sequential random walk program.
   javac WalkSeq.java
HANDS-ON EXERCISE 1B (cont.)

8. Run the sequential random walk program. The first argument is the random seed. The second argument is the number of iterations. Try different seeds. Try different numbers of iterations. Notice the running times.

```java
java WalkSeq 142857 10000000
```

9. Use your favorite terminal editor to write the Java source file for the SMP parallel program. Suggested class name: WalkSmp  File name: WalkSmp.java

```bash
vi WalkSmp.java
```

10. Compile the parallel random walk program.

```bash
javac WalkSmp.java
```

11. Run the parallel random walk program. The number after -Dpj.nt= is the number of parallel threads. You can specify from 1 to 4 threads. Try different numbers of iterations and different numbers of threads. Notice the running times.

```java
java -Dpj.nt=1 WalkSmp 142857 10000000
```

12. Do you see a speedup as you increase the number of threads?

13. What happens to the speedup as you increase the number of iterations?
import edu.rit.mp.buf.LongItemBuf;
import edu.rit.pj.Comm;
import edu.rit.pj.reduction.LongOp;
import edu.rit.util.LongRange;
import edu.rit.util.Random;

/**
 * Class PiClu is a cluster parallel program that calculates an approximate
 * value for π using a Monte Carlo technique. The program generates a number
 * of random points in the unit square (0,0) to (1,1) and counts how many of
 * them lie within a circle of radius 1 centered at the origin. The fraction of
 * the points within the circle is approximately \( \frac{\pi}{4} \).
 * 
 * Usage: java -Dpj.np=<I>K</I> PiClu <I>seed</I> <I>N</I>
 * 
 * The computation is performed in parallel in multiple threads. The program
 * uses class edu.rit.util.Random for its pseudorandom number generator. To
 * improve performance, each process has its own pseudorandom number generator,
 * and the program uses the reduction pattern to determine the count. The
 * program uses the “sequence splitting” technique with the pseudorandom number
 * generators to yield results identical to the sequential version. The program
 * measures the computation’s running time.
 * 
 * @author Alan Kaminsky
 * @version 27-Jun-2007
 */
public class PiClu
{

// Prevent construction.
private PiClu()
{
}

// Program shared variables.

// World communicator.
static Comm world;
static int size;
static int rank;

// Command line arguments.
static long seed;
static long N;

// Pseudorandom number generator.
static Random prng;

// Number of points within the unit circle.
static long count;

// Main program.

/** 
* Main program.
*/
public static void main(String[] args)
throws Exception
{
    // Start timing.
    long time = -System.currentTimeMillis();

    // Initialize middleware.
    Comm.init(args);
    world = Comm.world();
    size = world.size();
    rank = world.rank();

    // Validate command line arguments.
    if (args.length != 2) usage();
    seed = Long.parseLong(args[0]);
    N = Long.parseLong(args[1]);

    // Determine range of iterations for this thread.
    LongRange range = new LongRange(0, N-1).subrange(size, rank);
    long my_N = range.length();

    // Set up PRNG and skip ahead over the random numbers the lower-ranked
    // processes will generate.
    prng = Random.getInstance(seed);
    prng.skip(2 * range.lb());

    // Generate random points in the unit square, count how many are in the
    // unit circle.
    count = 0L;
for (long i = 0L; i < my_N; ++ i)
    {
        double x = prng.nextDouble();
        double y = prng.nextDouble();
        if (x*x + y*y <= 1.0) ++ count;
    }

    // Reduce all processes' counts together into process 0.
LongItemBuf buf = new LongItemBuf();
buf.item = count;
world.reduce (0, buf, LongOp.SUM);
count = buf.item;

    // Stop timing.
time += System.currentTimeMillis();

    // Print results.
System.out.println (time + " msec total " + rank);
if (rank == 0)
    {
        System.out.println
            ("pi = 4 * " + count + " / " + N + " = " +
            (4.0 * count / N));
    }
}

// Hidden operations.
/**
 * Print a usage message and exit.
 */
private static void usage()
{
    System.err.println ("Usage: java -Dpj.np=<K> PiClu <seed> <N>\n";
    System.err.println ("<K> = Number of parallel processes\n";
    System.err.println ("<seed> = Random seed\n";
    System.err.println ("<N> = Number of random points\n";
    System.exit (1);
}
HANDS-ON EXERCISE 2A
π CLUSTER PROGRAM PERFORMANCE

1. Log into the Macintosh workstation if necessary. Username: student Password: student
3. Start a terminal window.
4. Remote login to the Tardis parallel computer. Password: lebd_13bf_3c70_4195
   ssh paraconf@tardis.cs.rit.edu
5. Change into your working directory. Replace dir with your email name.
   cd dir
6. Run the sequential π program. The first argument is the random seed. The second argument
   is the number of iterations. Try different seeds. Try different numbers of iterations. Notice
   the running times.
   java PiSeq 142857 10000000
7. Run the cluster parallel π program. The number after -Dpj.np= is the number of parallel
   processes. You can specify from 1 to 10 processes. Try different numbers of iterations and
   different numbers of processes. Notice the running times.
   java -Dpj.np=1 PiClu 142857 10000000
8. Pick a number of iterations that yields a running time of about 30 seconds for the sequential
   program. Run the sequential program three times and record the running times on the next
   page. Run the parallel program three times each with 1, 2, 4, and 8 processes and record the
   running times. Calculate and record the median running time, speedup, and efficiency for
   each trial.
   Speedup = (Median sequential time) / (Median parallel time)
   Efficiency = (Speedup) / (Number of threads)
9. What do you notice about the speedup and efficiency as the number of processes increases?
10. Do the same measurements as Step 8 for larger numbers of iterations. What do you notice
    about the speedup and efficiency as the number of iterations increases?
HANDS-ON EXERCISE 2A (cont.)

Random seed: _______________  Number of iterations: _______________

<table>
<thead>
<tr>
<th># of procs.</th>
<th>T1 (msec)</th>
<th>T2 (msec)</th>
<th>T3 (msec)</th>
<th>Median T</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Random seed: _______________  Number of iterations: _______________

<table>
<thead>
<tr>
<th># of procs.</th>
<th>T1 (msec)</th>
<th>T2 (msec)</th>
<th>T3 (msec)</th>
<th>Median T</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Random seed: _______________  Number of iterations: _______________

<table>
<thead>
<tr>
<th># of procs.</th>
<th>T1 (msec)</th>
<th>T2 (msec)</th>
<th>T3 (msec)</th>
<th>Median T</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
HANDS-ON EXERCISE 2B
WRITE YOUR OWN CLUSTER PROGRAM

A three-dimensional random walk is defined as follows. A particle is initially positioned at (0, 0, 0) in the X-Y-Z coordinate space. The particle does a sequence of \( N \) steps. At each step, the particle chooses one of the six directions left, right, ahead, back, up or down at random, then moves one unit in that direction. Specifically, if the particle is at \((x, y, z)\):

With probability 1/6 the particle moves left to \((x-1, y, z)\);
With probability 1/6 the particle moves right to \((x+1, y, z)\);
With probability 1/6 the particle moves back to \((x, y-1, z)\);
With probability 1/6 the particle moves ahead to \((x, y+1, z)\);
With probability 1/6 the particle moves down to \((x, y, z-1)\);
With probability 1/6 the particle moves up to \((x, y, z+1)\).

Write a sequential program and a cluster parallel program to calculate the particle's final position. The program's command line arguments are the random seed and the number of steps \( N \). The program prints the particle's final position \((x, y, z)\) as well as the distance of the particle from the origin.

1. Log into the Macintosh workstation if necessary. Username: student Password: student
3. Start a terminal window.
4. Remote login to the Tardis parallel computer. Password: lebd_13bf_3c70_4195
   ssh paraconf@tardis.cs.rit.edu
5. Change into your working directory. Replace \( dir \) with your email name.
   \( cd \ dir \)
6. Use your favorite terminal editor to write the Java source file for the sequential program. Suggested class name: WalkSeq File name: WalkSeq.java
   \( vi \ WalkSeq.java \)
7. Compile the sequential random walk program.
   \( javac \ WalkSeq.java \)
HANDS-ON EXERCISE 2B (cont.)

8. Run the sequential random walk program. The first argument is the random seed. The second argument is the number of iterations. Try different seeds. Try different numbers of iterations. Notice the running times.

```java
java WalkSeq 142857 10000000
```

9. Use your favorite terminal editor to write the Java source file for the cluster parallel program. Suggested class name: `WalkClu` File name: `WalkClu.java`

```bash
vi WalkClu.java
```

10. Compile the parallel random walk program.

```bash
javac WalkClu.java
```

11. Run the parallel random walk program. The number after `-Dpj.np=` is the number of parallel processes. You can specify from 1 to 10 processes. Try different numbers of iterations and different numbers of processes. Notice the running times.

```java
java -Dpj.nt=1 WalkClu 142857 10000000
```

12. Do you see a speedup as you increase the number of processes?

13. What happens to the speedup as you increase the number of iterations?