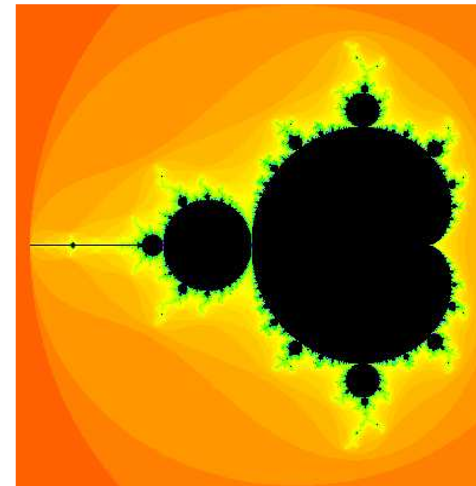
PARALLEL JAVA

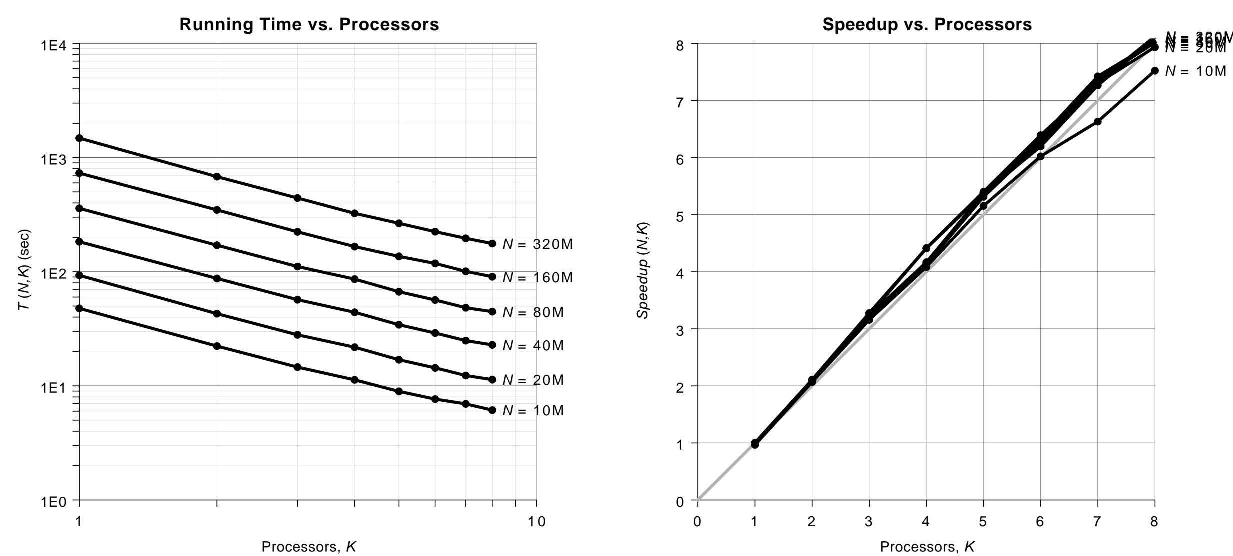
Alan Kaminsky
Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology

A LIBRARY FOR SMP, CLUSTER, AND HYBRID PARALLEL PROGRAMMING IN 100% JAVA

SMP PARALLEL PROGRAMMING WITH PJ



- Problem: Compute an image of the **Mandelbrot Set**
- Massively parallel problem; each pixel can be computed independently of all other pixels
- Load balancing required; pixels in the Mandelbrot Set (black pixels) require much more computation than pixels not in the Mandelbrot Set (colored pixels)
- Performance measurements on an 8-processor SMP machine with two Sun UltraSPARC-IV dual-core CPU chips, 1.35 GHz clock, 16 GB main memory, Sun JDK 1.5
- N = number of pixels
- K = number of processors



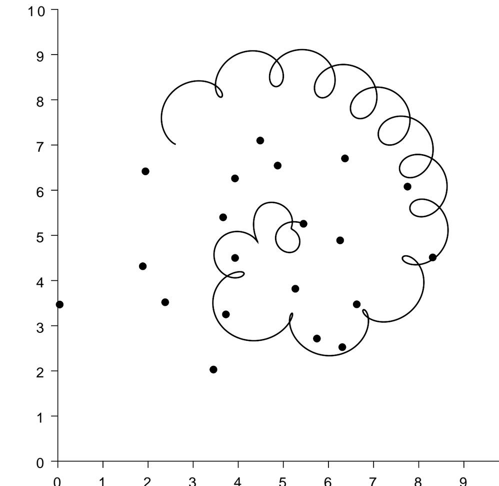
PJ's SMP parallel programming features are inspired by OpenMP

- Parallel thread teams; parallel code regions; nested parallel teams and regions
- Work-sharing parallel for loops with static, dynamic, self-guided, and user-programmable scheduling
- Parallel section groups; critical sections; single sections; barrier actions
- Parallel reduction of primitive types and objects using predefined and user-programmable operators
- Thread local variables; shared variables; multiple thread safe classes for primitive and array types

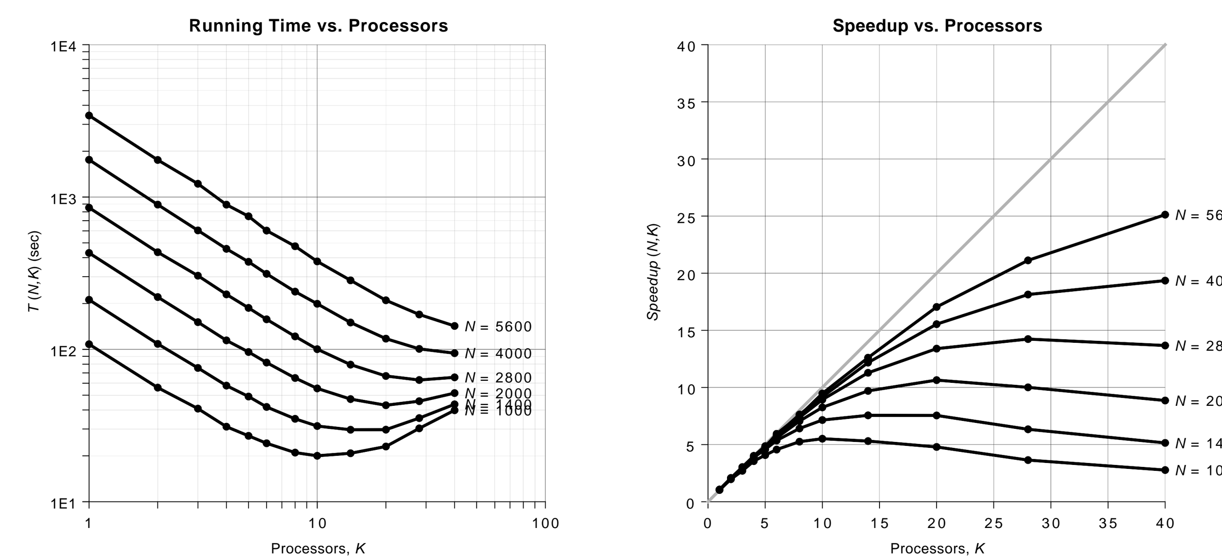
```
public static void main
   (String[] args)
   throws Exception
   {
   matrix = new int [height] [width];
   huetable = new int [maxiter+1];

   // Other initialization code omitted.

   new ParallelTeam().execute (new ParallelRegion()
      {
      public void run() throws Exception
         {
         execute (0, height-1, new IntegerForLoop()
            {
            public IntegerSchedule schedule()
               {
               return IntegerSchedule.guided();
               }
            public void run (int first, int last)
               {
               for (int r = first; r <= last; ++ r)
                  {
                  int[] matrix_r = matrix[r];
                  double y = ycenter + (yoffset - r) / resolution;

                  for (int c = 0; c < width; ++ c)
                     {
                     double x = xcenter + (xoffset + c) / resolution;

                     // Iterate until convergence.
                     int i = 0;
                     double aold = 0.0;
                     double bold = 0.0;
                     double a = 0.0;
                     double b = 0.0;
                     double zmagsqr = 0.0;
                     while (i < maxiter && zmagsqr <= 4.0)
                        {
                        ++ i;
                        a = aold*aold - bold*bold + x;
                        b = 2.0*aold*bold + y;
                        zmagsqr = a*a + b*b;
                        aold = a;
                        bold = b;
                        }

                     // Record number of iterations for pixel.
                     matrix_r[c] = huetable[i];
                     }
                  }
               }
            });
         }
      });

   // Code to write image file omitted.
   }
```

Parallel thread team
Region of code to be executed in parallel
Work-sharing parallel for loop
Self-guided loop schedule for automatic load balancing
Parallel for loop body
Thread local variables (declared inside parallel region)
Shared variables (declared outside parallel region)

CLUSTER PARALLEL PROGRAMMING WITH PJ



- Problem: **2-D Electromagnetic *N*-bodies Problem**
- Compute the motion of N antiprotons due to:
  - Repulsive forces from other antiprotons
  - Magnetic force from perpendicular magnetic field
- Antiproton positions must be communicated to all processors after each time step
- Computation time is $O(N^2/K)$
- Communication time is $O(N+K)$
- Performance measurements on a 10-node hybrid SMP cluster machine; each node with two AMD Opteron 2218 dual-core CPU chips, 2.6 GHz clock, 8 GB main memory, Sun JDK 1.5
- N = number of antiprotons
- K = number of processors



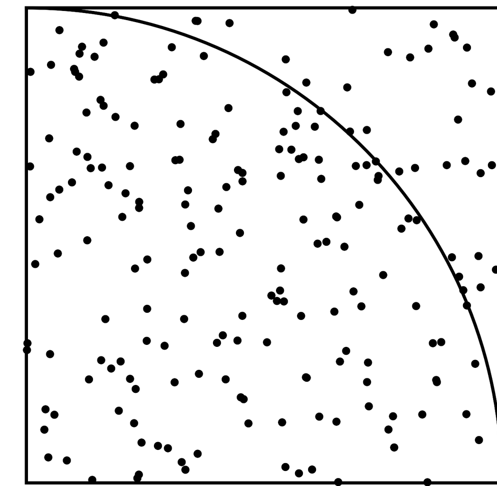
PJ's cluster parallel programming features are inspired by MPI

- PJ middleware automatically runs a program on multiple processors of the cluster
- Message passing of primitive types and object types (Java Object Serialization)
- Message passing of arrays and matrices, or arbitrary portions thereof
- Point-to-point communication: send, receive, send-receive; blocking and non-blocking versions
- Collective communication: broadcast, flood, scatter, gather, all-gather, reduce, all-reduce, barrier

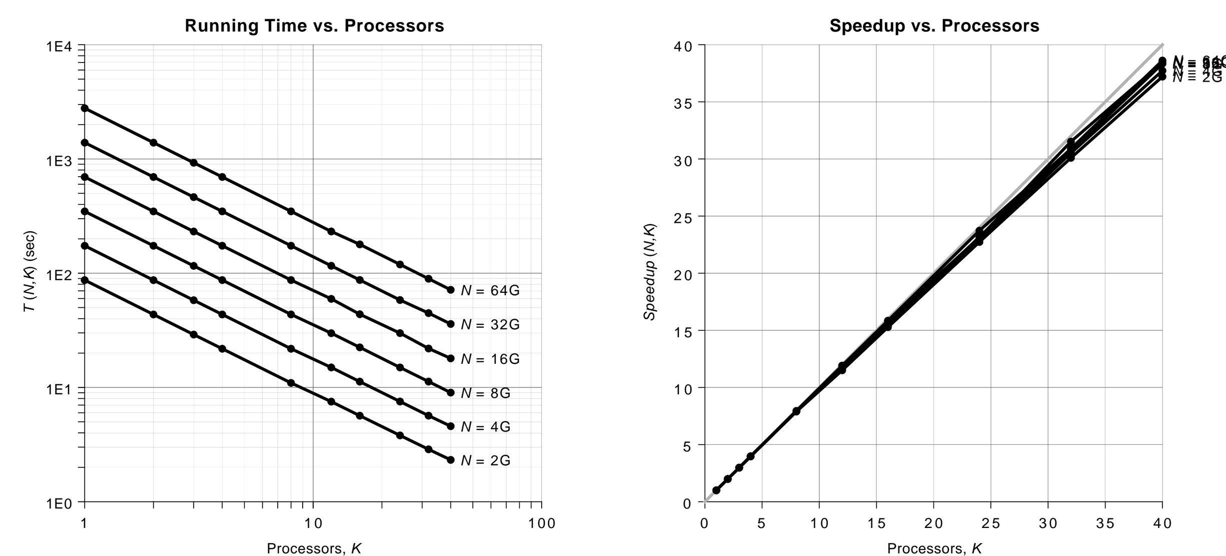
```
public static void main
   (String[] args)
   throws Exception
   {
   // Initialize world communicator.
   Comm.init (args);
   world = Comm.world();
   size = world.size();
   rank = world.rank();

   // Set up antiproton slices.
   slices = new Range (0, N-1) .subranges (size);
   mySlice = slices[rank];
   myLb = mySlice.lb();
   myLen = mySlice.length();

   // Initialize position vector array with all antiprotons.
   p = new Vector2D [N];

   // Initialize acceleration and velocity vector arrays with a slice of
   // antiprotons.
   a = new Vector2D [myLen];
   v = new Vector2D [myLen];

   // Set up position array communication buffers.
   buffers = Vector2D.doubleSliceBuffers (p, slices);
   myBuffer = buffers[rank];

   // Other initialization code omitted.

   // Do <snaps> snapshots.
   for (int s = 0; s < snaps; ++ s)
      {
      // Advance time by <steps> steps.
      for (int t = 0; t < steps; ++ t)
         {
         // Compute accelerations (forces) on antiprotons in this slice.
         computeAcceleration();

         // Update positions and velocities by one time step.
         step();

         // All-gather the new antiproton positions.
         world.allGather (myBuffer, buffers);
         }

      // Compute total momentum of antiprotons in this slice.
      computeTotalMomentum();

      // Code to write snapshot of antiproton positions omitted.
      }
   }
```

Run copies of this program on cluster nodes; initialize message passing middleware
World communicator
Slice antiproton position array equally among processes
"Buffer" objects define sources and destinations of message data
Collective communication to update all processes with new antiproton positions

HYBRID PARALLEL PROGRAMMING WITH PJ



- Problem: **Monte Carlo Integration**
- Compute the area of one quadrant of a unit circle
  - N random points in unit square, C points inside circle
  - $C/N \approx$ (quadrant's area)/(square's area) $= \pi/4$
  - So $4C/N$ should be $\approx \pi$
- Massively parallel problem
- Hybrid parallel program runs with multiple processes on the cluster nodes and with multiple threads in each process
- Thread local PRNGs for improved performance
- Thread local counters, with a final reduction
- Performance measurements on the same 10-node hybrid SMP cluster machine; Sun JDK 1.5
- N = number of random points
- K = number of processors



OpenMP and MPI lack PJ's hybrid parallel programming abilities

- OpenMP has no routines for message passing
- MPI has no capabilities for executing sections of code in parallel in multiple threads
- Programs using both OpenMP and MPI are not guaranteed to work; MPI need not support multiple threads
- Sending messages between separate processes on the same SMP node often yields poorer performance than sharing memory within the same SMP node
- Hybrid parallel programs should use multithreading within each node and message passing between nodes

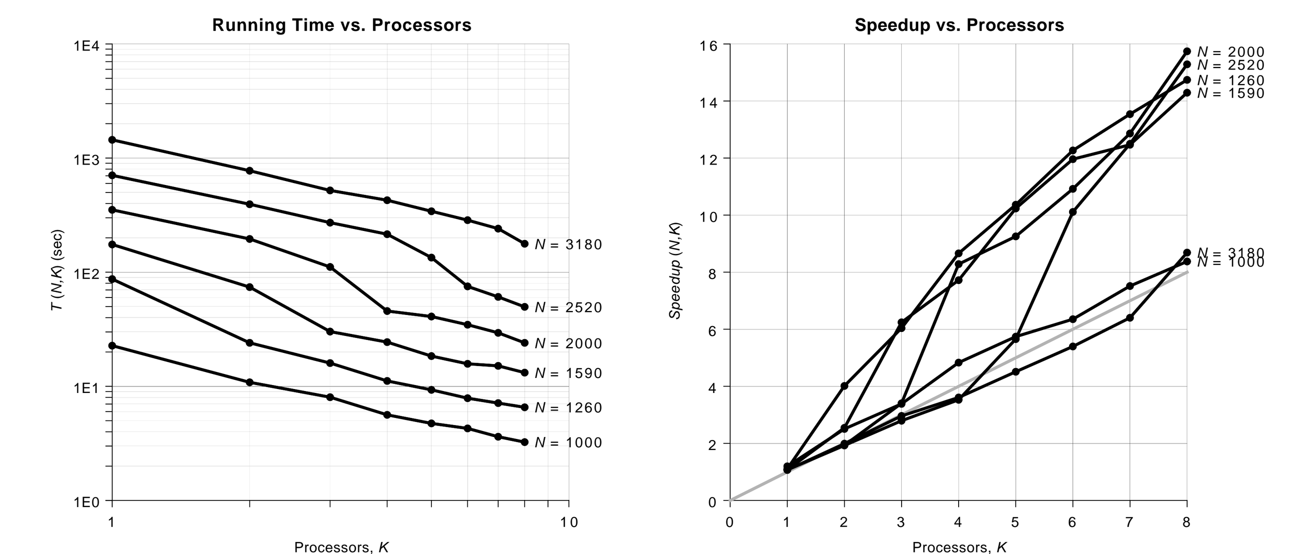
```
public static void main (String[] args) throws Exception
   {
   // Initialize world communicator.
   Comm.init (args);
   world = Comm.world();
   size = world.size();
   rank = world.rank();

   // Other initialization code omitted.

   // Determine subrange of points this process will do.
   LongRange range = new LongRange (0, N-1) .subrange (size, rank);
   lb = range.lb();
   ub = range.ub();

   // Generate subrange of N random points.
   count = new SharedLong (0L);
   new ParallelTeam().execute (new ParallelRegion()
      {
      public void run() throws Exception
         {
         execute (lb, ub, new LongForLoop()
            {
            // Set up per-thread PRNG and counter.
            Random prng_thread = Random.getInstance (seed);
            long count_thread = 0L;

            // Parallel loop body.
            public void run (long first, long last)
               {
               prng_thread.setSeed (seed);
               prng_thread.skip (2 * first);
               for (long i = first; i <= last; ++ 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.
            public void finish()
               {
               count.addAndGet (count_thread);
               }
            });
         }
      });

   // Reduce all processes' shared counts into process 0.
   LongItemBuf buf = LongBuf.buffer (count.longValue());
   world.reduce (0, buf, LongOp.SUM);

   // Print results.
   if (rank == 0) System.out.println ("pi = " + (4.0 * buf.item / N));
   }
```

Run copies of this program on cluster nodes; initialize message passing middleware
World communicator
Slice random points equally among processes
Shared reduction variable
Parallel thread team
Sub-slice random points equally among threads
Thread local variables
Parallel random number generator using "sequence splitting"
Shared memory reduction of thread local counters
Message passing reduction of processes' counters

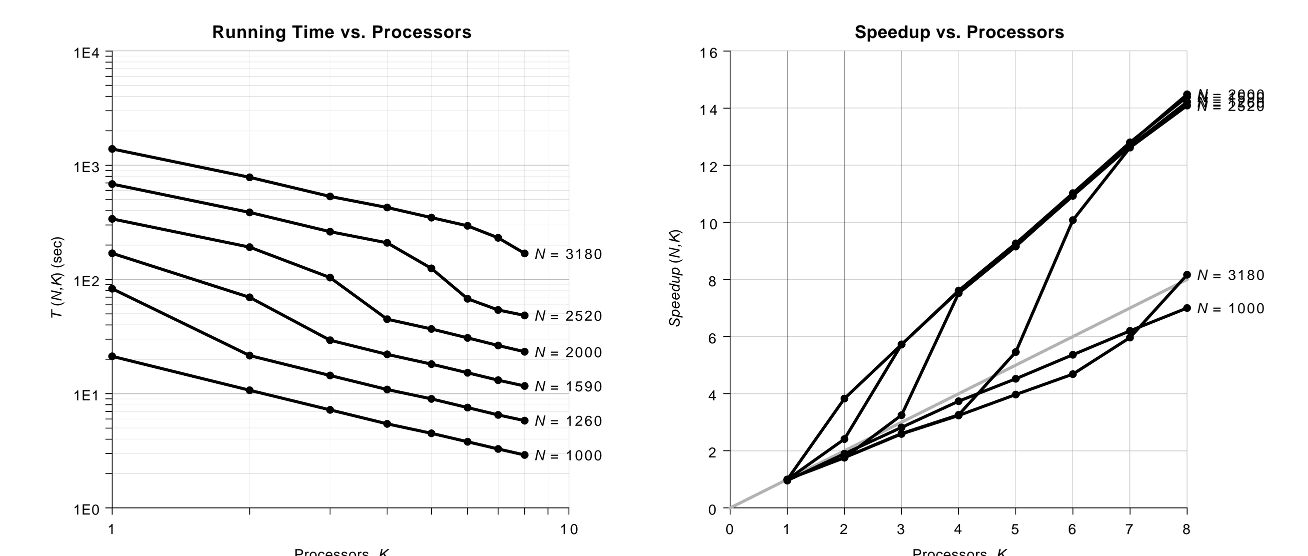
JAVA/PJ PROGRAMS ARE AS FAST AS C/OPENMP

- Problem: **All Shortest Paths in a Graph**
- Input distance matrix: d_{ij} = distance from vertex i to adjacent vertex j, or ∞ if not adjacent
- Output distance matrix: d_{ij} = length of shortest path from vertex i to vertex j, or ∞ if not connected
- Floyd's Algorithm:
  - for $i = 0$ to $N-1$
    - for $r = 0$ to $N-1$
      - for $c = 0$ to $N-1$
        - $d_{rc} = \min(d_{rc}, d_{ri} + d_{ic})$
- Java/PJ program compiled and run using Sun JDK 1.5 with Sun HotSpot just-in-time compiler
- C/OpenMP program compiled using Sun C compiler at highest optimization (cc -xO5 -xopenmp)
- Performance measurements on the same 8-processor SMP machine
- N = number of graph vertices
- K = number of processors

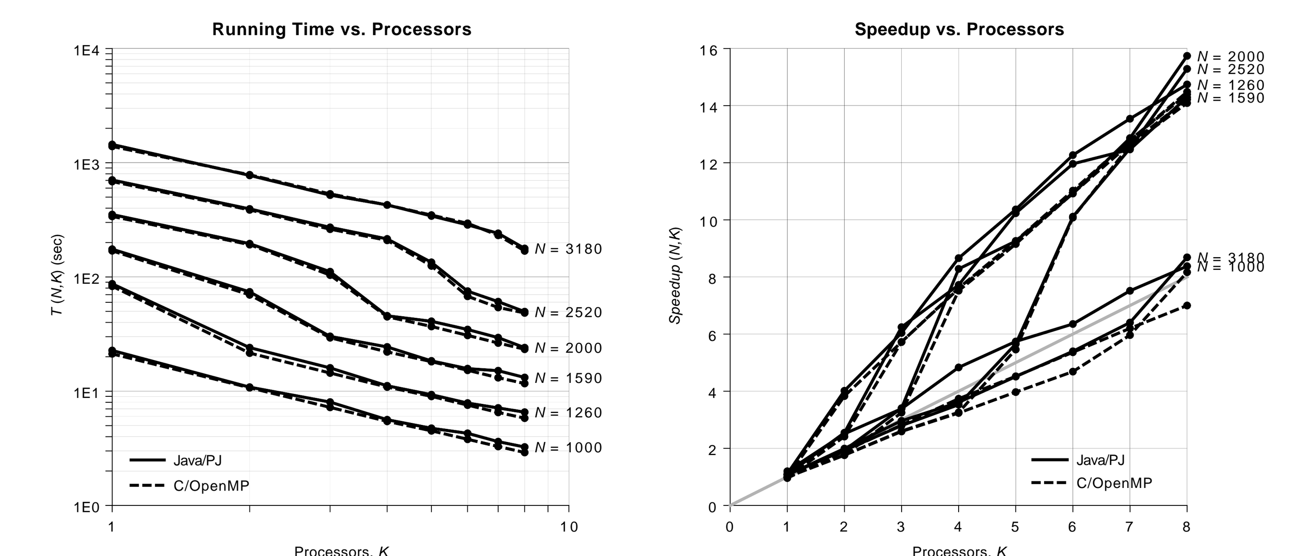
Java/PJ Performance



C/OpenMP Performance



Java/PJ and C/OpenMP Performance, Superimposed



FOR FURTHER INFORMATION

- Alan Kaminsky, Department of Computer Science, Rochester Institute of Technology, ark@cs.rit.edu
- Parallel Java Library download (GNU GPL licensed): http://www.cs.rit.edu/~ark/pj.shtml
- Parallel Java Library documentation (Javadoc): http://www.cs.rit.edu/~ark/pj/doc/index.html
- Parallel Computing I course materials: http://www.cs.rit.edu/~ark/531/
- Parallel Computing II course materials: http://www.cs.rit.edu/~ark/532/

Presented at the 13th SIAM Conference on Parallel Processing for Scientific Computing (PP08), Atlanta, GA, USA, March 12—14, 2008