Chapter 6

Applications and Performance of mpiJava

We have implemented several SPMD style parallel applications using `mpiJava`. These applications were included in mpiJava software package. In this chapter we discuss two parallel Monte Carlo simulations which can be used for education purpose and experimental performance of mpiJava with a real application.

In first section, we briefly introduce a Java version of parallel random number generator which is used in two Monte Carlo spin model simulations, Ising Model and Potts Model. Then we discuss the `mpiJava` version of Monte Carlo simulation of Ising model implemented with two different algorithms, metropolis and cluster algorithms. This Ising model simulation can be used as a full application Java benchmarks with comparable native MPI-C and MPI-Fortran codes for performance comparison. This is present in the last section. We also present graphical Potts model simulation in Section 6.3. The Potts model is a generalized extension of the Ising model to an arbitrary number, Q, of discrete states, however, it has a much richer phase structure, which makes it a richer graphical visualization.

6.1 Parallel Random Number generator

Random number generators are widely used for simulations in computational science and engineering application. For some Java Grande applications, such Monte Carlo simulations, it is crucial that the random number generator have good randomness properties. This is especially true in large-scale simulations on parallel supercomputers, which consume huge quantities of random numbers, and require parallel algorithms for random number generation. The techniques for testing the quality of parallel random number are presented in [32].

Random number generators use iterative deterministic algorithms for producing for a sequence X_i of pseudo-random numbers that approximate a truly random sequence. Probably the most commonly-used algorithms used for sequential random number generators are Linear Congruential Generators (LCG). LCGs produce a sequence X_i of random integers using the relation

$$X_i = (a * X_{i-1} + b) \mod M, \tag{6.1}$$

where a is known as the multiplier, M is the modulus, and b is an additive constant. If $b = 0$, this termed a Multiplicative Linear Congruential Generators (MLCG). The parameters (a, b, M) must be chosen carefully to ensure a large period and good uniformity and randomness properties. Standard random number generators return a floating point number x_i in the interval [0,1), which can be obtained by dividing X_i by M.

Many different parallel random number generators have been proposed [10, 31], but most of them use the same basic concept, which is to parallelize a sequential generator by taking the elements of the sequence of pseudo-random numbers it generates and distributing them among the processors in some way. One of the method to achieve this goal is to split the sequence into non-overlapping contiguous sections, each of which is generated by a different processor. This also utilizes the fact that this type of generator allows an arbitrary element of the sequence to be calculated. One can therefore simply divide the period of the generator by the number of processors, and jump ahead in the sequence by this amount for each processor. Alternatively, the length of each section of numbers could be chosen much larger than could be used by any processor. If the length of the sections is L, then processor P would generate the sequence

$$X_{PL}, X_{PL+1}, X_{PL+2}, \ldots,$$

This method also requires the ability to jump ahead in the sequence by a given amount.

Another simple method for parallelizing random number generators is for processor P of an N processor machine to generate the sub-sequence

$$X_P, X_{P+N}, X_{P+2N}, \ldots,$$

so that the sequence is spread across processors in the same way as a deck of cards is dealt in turn to players in a card game. This is known as the leapfrog technique, since each processor leap-frogs by N in the sequence. In order to use this method we need to be able to easily jump ahead in the sequence. Linear congruential generators are simple enough that it is possible to specify analytically an arbitrary element in the sequence. By recursively applying equation 6.1, it is easy to show that the element X_{P+N} is given by

$$\begin{aligned} X_{P+N} &= (A * X_P + B) \mod M \qquad (6.2) \\ A &= a^N \\ B &= \sum_{k=0}^{N-1} a^k = (a^N - 1)/(a - 1) \end{aligned}$$

Thus moving through the sequence with stride N, rather than 1, is simply accomplished by specifying a different multiplier and additive constant for the generator.

```
// Parallel MLCG program

class Prand {
  final static int MULT = 1103515245;
  final static int ADD  = 12345;
  final static int MASK = 0x7fffffff;
  final static double twoto31 = 2147483648.0;
  static long AAA,BBB,randx;

  // Initialize random number generator.
  static void randinit(int seed) {
    AAA = 1;
    BBB = 0;
    for (int proc=0;proc<Total_procs;++proc) {
      AAA = (MULT * AAA) & MASK;
      BBB = (MULT * BBB + ADD) & MASK;
      if (proc == My_procnum) randx = (AAA*seed + BBB) & MASK;
    }
  }

  // Return a random double in [0, 1.0).
  static double rand()  {
    double retvalue = randx / twoto31;
    randx = (AAA*randx+BBB)&MASK;
    return(retvalue);
  }
}
```

Figure 6.1: Parallel Multiplicative Linear Congruential Generators (MLCG).

In mpiJava Potts model and Ising model simulations, we use parallel MLCG using leapfrog algorithm as shown in Figure 6.1. The leapfrog method is consequently very easy to implement on parallel machine. The idea is to take a standard linear congruential algorithm, and have every processor compute the *Pth* iterate of that algorithm (where P is the number of processors.) [48]. If the seeds are set up properly (staggered) the processors leapfrog over one another, and it is just as good as having used the basic algorithm on a sequential machine.

6.2 The Monte Carlo Simulations of Spin Model

Parallel computers have been very successfully applied to the Monte Carlo simulation of spin models. Here we present overview of the Potts spin models and then outline two different methods, metropolis and cluster algorithms that known for Monte Carlo simulation of many spin model.

6.2.1 The Potts Model

The Potts model [84, 96] encompasses a number of problems in statistical physics and lattice theory. It generalizes of the Ising model so that each spin can have more than two components, and has been a subject of increasingly intense research interest in recent years. It includes the ice-rule vertex and bond percolation models as special cases.

The Q-state Potts model consists of a lattice of spins σ_i, which can take Q different values, and whose Hamiltonian is

$$H = K \sum_{(i,j)} \delta_{\sigma_i \sigma_j}, \tag{6.3}$$

where the spins take on the values $\sigma_i = 0, 1, 2, ..., Q-1$, the sum is over nearest neighbor pairs on sites on a lattice. For Q=2, this is equivalent to the Ising model. The Potts model is thus a simple extension of the Ising model; however, it has a much richer phase structure, which makes it an important testing ground for new theories and algorithms in the study of critical phenomena.

6.2.2 The Metropolis Algorithm

The Metropolis algorithm [39] was invented by Metropolis et al in 1953 for generating an arbitrary probability distribution. This algorithm has been the most successful

1. Select one spin to flip.
2. Compute the change in energy resulting from this flip.
3. Calculate the probability $P = exp(-\beta * \Delta E)$ for this flip.
4. Generate a random number r between zero and one.
5. If $\Delta E \leq 0$ or $r < P$, flip the spin to one of values $0, 1, ..., Q - 1$.
6. Repeat step 1 to 5 over all sites.
7. Visualize the updated lattice configuration.
8. Analyze the resulting configuration.

Figure 6.2: The Main Procedure of Potts Model Simulation using Metropolis Algorithm: One starts with an initial configuration of spins and repeats these procedures.

and influential of all the areas of Monte Carlo Method.

In the Metropolis algorithm, new states or configurations of the system are found by systematically moving through all the lattice sites. A new configuration ϕ' is generated by updating a single variable in the old configuration ϕ and calculating the change in energy

$$\Delta E = E(\phi') - E(\phi). \tag{6.4}$$

If $\Delta E \leq 0$, the change is accepted; if $\Delta E > 0$ the change is accepted with probability $\exp(-\beta * \Delta E)$, where β is inverse temperature. In practice, on a computer, this is done by generating a pseudo-random number r in the interval [0,1) with uniform probability distribution and accepting the change if $r < \exp(-\Delta E)$. These procedures of Metropolis algorithm are shown in Figure 6.2. If the variable is updated by a large amount then the change in energy will be large and the probability of the update being accepted will be small. Conversely, if the variable is changed by

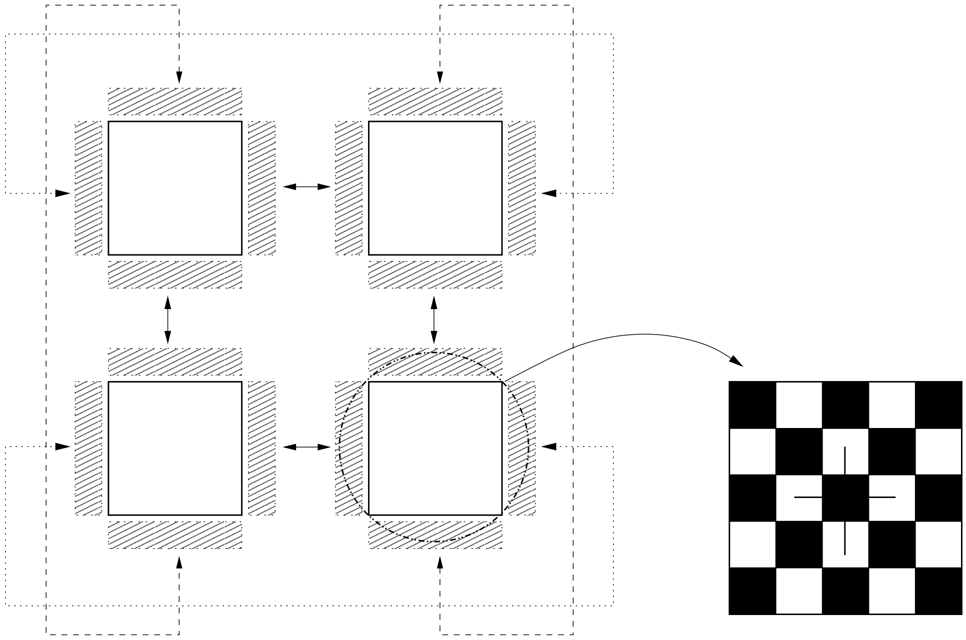

Figure 6.3: Checkerboard Partition and Blocked Communication.

a small amount then the changes will usually be accepted but their small size will make exploration of phase space rather slow.

The Metropolis algorithm for a spin model is well suited to parallelism, since it is regular – so can use standard data decomposition of the lattice onto the processors and get good load balance – and local – the update of a site depends only on its nearest neighbors, so only local communication is required. On SIMD computers, we can not update a site and its neighbors at the same time, since they are dependent each other, and we would violate detailed balance. The detailed balance is a sufficient condition for equilibrium probability distribution and guarantee a valid Monte Carlo procedure. Therefore each site must updated separately. However we can partition the lattice into a checkerboard of red and black sites, and update all the black sited in parallel(at once), since they are all independent, and then followed by all red sites in parallel. This method is well known as *red/black* or *checkerboard* update.

On MIMD computers, we do a checkerboard partitioning of the lattice to do the data communication as large blocks of spins rather than single spins, by passing all the edge data to neighboring processors after each red or black update. This checkerboard partitioning and blocked communication is illustrated in Figure 6.3. This blocked communication greatly reduces the latency time for communication, thereby improving efficiency. It is also possible to overlap communications with computation, by computing edge values first, and sending them while the interior points are being computed.

6.2.3 The Swendsen-Wang Cluster Algorithm

The previous Metropolis algorithm perform poorly because the updates are *local*, that is, one spin at a time is updated. Cluster algorithms use a clever way of finding large clusters of sites which can be updated at once.

Cluster algorithms were proposed by Swendsen and Wang [87] in 1987, for Potts spin models. They have since been generalized to other models, and other algorithms. In the Swendsen-Wang algorithm, clusters of spins are created by introducing bonds between neighboring spins with probability

$$P(\sigma_i, \sigma_j) = \delta_{\sigma_i \sigma_j}(1 - \exp^{-\beta}) \tag{6.5}$$

if the two spins are the same, and zero if they are not. All such clusters are generated and then updated by choosing a random new spin value for each cluster and assigning it to all the spins in that cluster.

A slightly different cluster algorithm has been proposed by Wolff [**?**]. In the Swendsen-Wang algorithm, we generated many clusters and then flip these clusters. In Wolff algorithm, a spin is chosen at random and a single cluster constructed around it, using same bond probabilities as for the Swendsen-Wang algorithm. All the spins in this cluster are then flipped, i.e, changed to a random new spin different

1. Go through each four nearest neighbor connection of the lattice, set up a bond between the two neighboring sites i and j with probability $p = 1 - \exp^{-\beta}$ only if the spins are the same, otherwise zero.

2. Identify clusters as a set of sites connected by bonds. Two sites are said to be in the same cluster if there is a connected path of bonds joining them. Every site has to belong to one of the clusters.

3. After the clusters are created, each cluster is assigned a new Ising spin chosen with equal probability between +1 and -1.

4. Repeat step 1 to 3 for the next iteration.

Figure 6.4: The Main Procedure of Sequential Swendsen-Wang Algorithm.

from the old one. These parallel Wolff cluster algorithms were developed in several different languages including C code for the CM-5 using CMMD [88] which was presented in [12].

Cluster algorithms have in common the problem of identifying and labeling the connected clusters of spins. This is very similar to an important problem in image processing, that of identifying and labeling the connected components in a binary or multi-colored image composed of an array of pixels. The only real difference is that in the spin model case, neighboring sites of the same spin have a certain *probability* of being in the same cluster, while for neighboring pixels of the same color that probability is one. Unfortunately this is a large enough difference so that some algorithms which work in image analysis will not work, or require substantial changes, for spin models. In our Ising model, we have used a commonly used cluster identification algorithm invented by Hoshen and Kopelman.

6.2.3.1 Hoshen and Kopelman

Hoshen and Kopelman [61] developed a sequential cluster identification algorithm which gives each cluster a unique label and counts the number of sites it contains. Each site i belonging to cluster α is assigned a cluster label m_t^α. The cluster α may initially be assigned several different cluster labels. These are given as a set of natural numbers $\{m_1^\alpha, m_2^\alpha, \ldots, m_s^\alpha, \ldots, m_t^\alpha, \ldots\}$. In this set only one number is regarded as the *proper* cluster label which we designate as m_s^α. This is the smallest number in the set.

A connection between the label m_t^α at any site, and the proper cluster cluster label m_s^α, is provided by an array N as $\{N(m_1^\alpha), N(m_2^\alpha), \ldots, N(m_s^\alpha), \ldots, N(m_t^\alpha), \ldots\}$. This is constructed so that $N(m_s^\alpha)$ is the only positive integer member of the set and denotes the number of sites in the cluster, while the remaining $N(m_t^\alpha)$ provide the links between the $N(m_t^\alpha)$ and the proper cluster label $N(m_s^\alpha)$. To be specific, if a site with label m_t^α is bonded to a site which has the proper cluster label m_s^α then

$$N(m_t^\alpha) = -m_s^\alpha \tag{6.6}$$

However, if a site with label m_p^α is not directly bonded to a site with the proper cluster label, but is connected to a site with label m_t^α then

$$N(m_p^\alpha) = -m_t^\alpha \tag{6.7}$$

Therefor to get the proper cluster label for this site we have to go through m_t^α using 6.7 then 6.6. Similarly for higher-order indirections, we just need to iterate $-m \leftarrow N(m)$ until we reach a positive value of $N(m)$, which means that m is the proper cluster label and $N(m)$ the current number of sites in the cluster. Fortunately in most cases this hierarchy consists to one or two levels(one or two equations) only [61].

In practice the algorithm works as follows. We sweep through the sites of the lattice looking at neighbors in the negative directions(there are d of them for a d-dimensional lattice). If site i has no connected neighbors(or none of its neighbors have been labeled) then it is assigned a new label $S_i = m_t^\alpha$, and $N(m_t^\alpha) = 1$. If there is only one connected neighbor, at site n with label $S_n = m_r^\alpha$, say, then i gets the same label as $n : S_i = m_r^\alpha$, and $N(m_t^\alpha) = N(m_t^\alpha)+1$. The good feature of this cluster labeling technique becomes apparent when site i links two or more previously labeled cluster fragments into a single cluster, that is, when it is bonded to two or more of its neighbors. No site belonging to any of these cluster fragments is relabeled (so that once a site is labeled it retains this label throughout the labeling process)—instead the readjustments occur within the $N(m_t^\alpha)$. The number of readjusted $N(m_t^\alpha)$'s for a site is equal to the number of coalescing cluster fragments at that site. Let us assume that the connected sites belong to clusters $\alpha, \beta, \gamma, \ldots$ which have proper cluster labels $m_s^\alpha, m_s^\beta, m_s^\gamma, \ldots$ with m_s^α being the smallest. These clusters coalesce at i to from a single larger cluster so we set label $S_i = m_s^\alpha$ and readjust

$$\begin{aligned} N(m_s^\alpha) &= N(m_s^\alpha) + N(m_s^\beta) + N(m_s^\gamma) + \ldots + 1 \\ N(m_s^\beta) &= -m_s^\alpha \\ N(m_s^\gamma) &= -m_s^\alpha \\ &\ldots \end{aligned} \tag{6.8}$$

It is this continual readjustment which keeps the hierarchy of indirections small. As a final stage in the algorithm we can pass through the lattice a second time setting the label at each site to be the proper cluster label m_s^α. This makes it easier to pick out the clusters for updating. An illustration of this algorithm for the 4x4 lattice used in Figure **??**.

6.2.3.2 Parallel Cluster Algorithm

A parallel cluster algorithm involves distributing the lattice onto an array of processors using the usual domain decomposition. Clearly a sequential algorithm can be used to label the clusters on each processors, but we also need a procedure for converting these labels to their correct *global* values. We need to be able to tell many processors, which may be any distance apart, that some of their clusters are actually the same. Thus we need to be able to agree on which of many different local labels for a given cluster should be assigned to be the global cluster label, and to pass this label to all the processors containing a part of that cluster. We will discuss one of the methods called *self-labeling* for tackling this problem.

Self-labeling We shall refer to this algorithm as *self-labeling*, since each site figures out which cluster it is in by itself from local information. We begin by assigning each site i a unique cluster label S_i. In practice this is simply chosen as the position of that site in the lattice. At each step of the algorithm, in parallel, every site looks in turn at each of its neighbors in the positive directions. If it is bonded to a neighboring site n which has a different cluster label S_n, then both S_i and S_n are set to the minimum of the two. This is continued until nothing changes, by which time all the clusters will have been labeled with the minimum initial label of all the sites in the cluster. Note that to check termination of the algorithm involves each processors sending a termination flag(finished or nor finished) to every other processor after each step, which can become very costly for large processor array.

We can improve this method by using faster sequential algorithm, such as Hoshen and Kopelman mentioned in previous section, to label the clusters in the sub-lattice on each processor, and then just use self-labeling on the sites at the edges of each processor to eventually arrive at the global cluster labels. These are

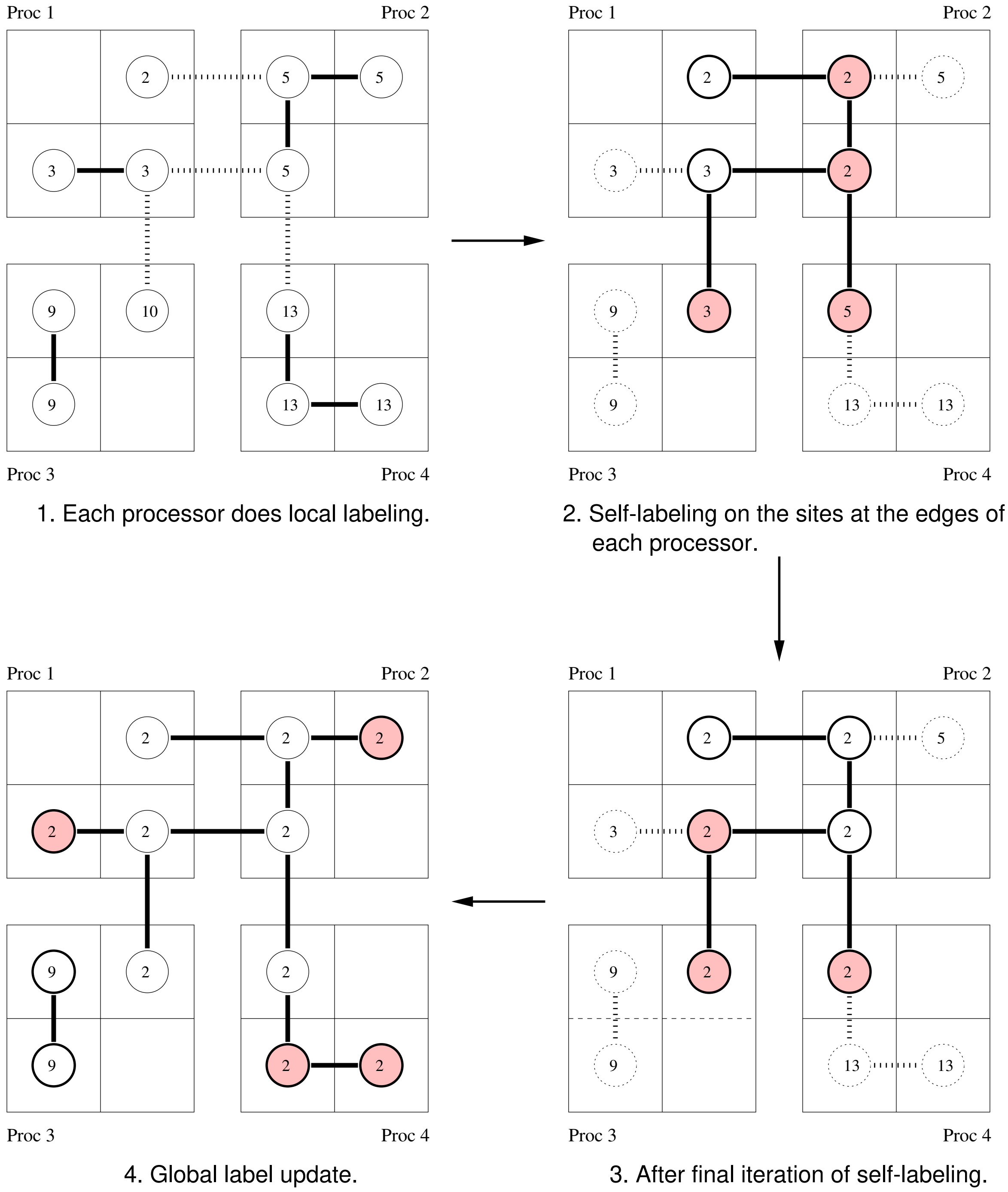

Figure 6.5: MIMD Component Labeling. The bonds are shown as the thick lines.

illustrated in Figure 6.5. The number of steps required to do the self-labeling will depend on the largest cluster, which at the phase transition will generally span the entire lattice. The number of steps self-labeling steps will therefore be of the order of the maximum distance between processors, which for a square array of P processors is just $2\sqrt{P}$. Hence the amount of communication(and calculation) involved in doing the self-labeling, which is proportional to the number of iterations times the perimeter of the sub-lattice, goes like L for an $L \times L$ lattice, whereas the time taken on each processor to do the local cluster labeling goes like the area of the sub-lattice, which is L^2/P. Therefore as long as L is substantially greater than the number of processors we can expect to obtain a reasonable speedup.

6.3 Graphical Simulation

`mpiJava` application that performs a Potts Model Monte Carlo simulation using parallel Metropolis algorithms in two dimensions running on four processors is shown in Figure 6.6. One of the good feature of `mpiJava` is that users have full access to the Java AWT and other Java graphical packages to support GUI and visualize graphical output of application. Visualization of the collected data is a critical element in providing developers or educators with the needed insight into the system under study and education. In this Potts model simulation, the different colors represent the Q different spin states on a lattice. All the options of the simulation are customizable through a GUI. Processor *P0* handles all GUI input and visualization of output data. When users run simulation by clicking start button, input values are broadcasted to other processors and then after each processor generate a new configuration, processor *P0* gathers all updated data to plot the output in order to visualize the results. Measurements of standard quantities such as energy and magnetization are easily done in parallel by just calculate the quantities locally, sum

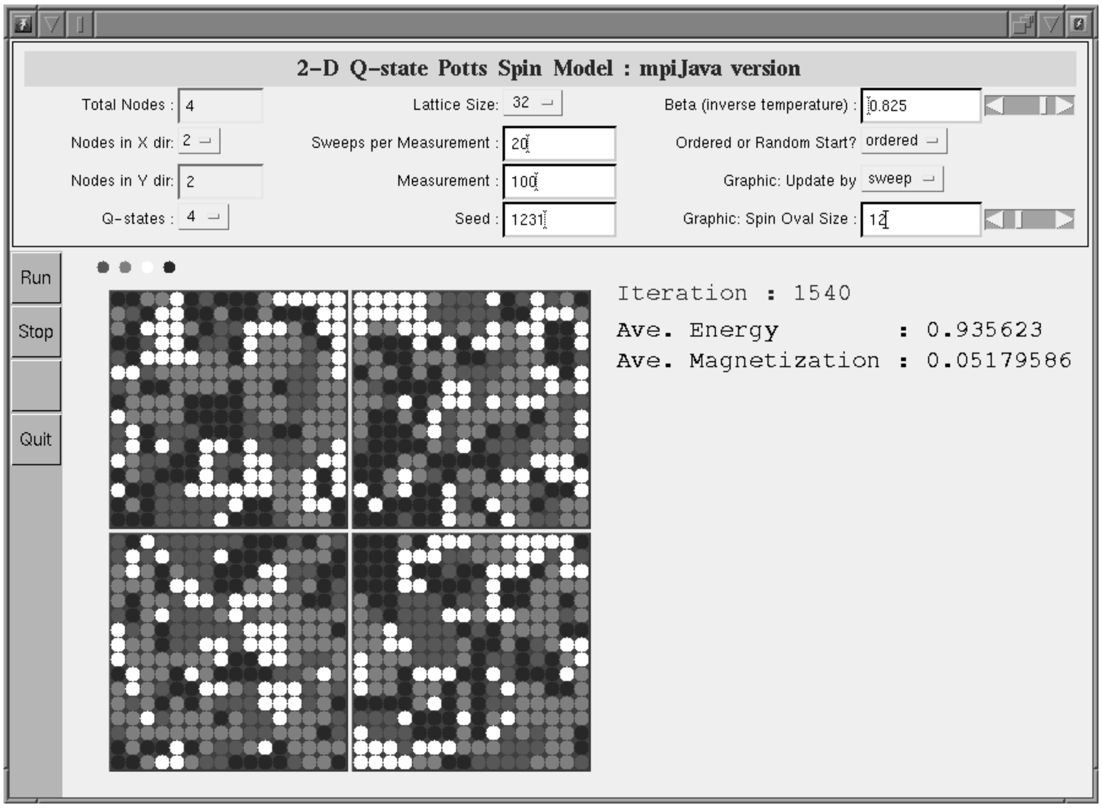

Figure 6.6: The Potts Model Monte Carlo Simulation.

over sites in each processor, and then combine the partial sums on each processor using MPI.Allreduce.

Users can also choose data distribution in row, column or block wise through the number of nodes in X direction options after the number of running processors is given at the starting time by execution command as follow;

```
prunjava 4 Potts
```

The `prunjava` script is a wrapper for the MPICH `mpirun` script. The first argument is the number of processors on which the program will be executed. The number of nodes in Y direction is set automatically by total number of running nodes and the given number of nodes in X direction. A list of available host computers may be given in an MPICH-style 'machines' file in the local directory.

From this simulation, one is able to determine the average magnetization, the average energy, and the specific heat of the system and numerous other quantities which are experimentally observable.

6.4 Ising Model Evaluation

In this section we evaluate the performance of applications written in mpiJava. We have developed both sequential and parallel message passing programs to compare the performance of mpiJava using Monte Carlo simulation of the two dimensional Ising model. The sequential program were written in C, F77 and Java and message passing programs were written in MPI-C, MPI-F77 and mpiJava. This Ising model test can also be used for testing the quality of parallel random number generators [32].

The two different methods, Metropolis and Swendsen-Wang cluster algorithms are used with a standard block domain decomposition. As in Potts model, the red/black updating scheme is used in parallel Metropolis Ising model simulation [48].

The system environment was as follows:

- CPU: Four dual processor (UltraSparc 200 MHz) Solaris workstations with 256 MBytes of DRAM. Each of the two processors/system has 1MB cache.
- OS: Solaris 2.5.1
- Network: 10BaseT Ethernet and SunATM-155/MFiber
- Java VM: Sun's JIT
- Java Compiler: Sun JDK1.2.2
- MPI packages: mpich1.2.0 and Sun HPC MPI 2.0

For comparison, we have completed experiments for three different versions of out programming environment—sequential codes, parallel codes with mpich and Sun HPC MPI. For the sequential F77 and C codes, we use Sun WorkShop FORTRAN 77 5.0 and C 5.0 compilers and Sun JDK1.2.2 for sequential Java code. For the parallel mpiJava codes, we use mpich 1.2.0 and Sun HPC MPI 2.0 on 1, 2, 4 and 8 nodes. The mpich 1.2.0 was compiled with `-comm=shared` communication option so that to permit the use of multiple communication of both shared memory and TCP/IP, since each of four our Solaris machines has dual processors. Sun HPC MPI uses Sun ATM network which is much faster in communication than p4 device of mpich. For better performance, all sequential and parallel Fortran, C and Java codes were compiled using `-O` optimization option and all sequential Java and mpiJava codes executed with JIT enabled.

We performed Monte Carlo Ising simulations on simple 2-D lattices with linear size of L=32 to 1024 and periodic boundary conditions. To measure the total execution time, we performed 20 iterations to thermalize the system and at least 2000 to Monte Carlo sweeps. The inverse temperature β was taken to be the critical inverse temperature $\beta_c = \log(1+\sqrt{2})/2 \approx 0.4406868$ for the $2-d$ Ising model. Timings were measured using `MPI_Wtime` for parallel and shell built-in `time` command for sequential codes. We have repeated the benchmarks several times when there was little network activity and on quiet machines.

The Metropolis timing results for sequential and parallel tests for different lattice sizes are shown in Figure 6.7 through 6.12 and Swendsen-Wang in Figure 6.13 through 6.18. The results demonstrate as follows;

- **mpich vs sunHPC**

  In both Metropolis and Swendsen-Wang(SW), for small lattice size $L \leq 64$ on node $N \geq 2$, the performance of the mpiJava benchmarks is better with Sun HPC MPI than mpich due to faster communication device. On the other hand, for the larger latice size $L \geq 512$, the performance difference is much smaller between sunHPC and mpich, since the main overhead is in the computations rather than communications as the size of sub-lattice increases.

- **mpiJava vs serial codes**

  The speedup obtained for a variety of lattice sizes using the mpiJava, as compared with sequential Java, are shown in Figure 6.19 and 6.20. On the largest lattice size plotted(1024×1024), mpiJava with sunHPC provides a speedup up to 2.4(Metropolis) and 4.6(SW) over sequential F77 and, 2.1(Metropolis) and 3.7(SW) over sequential C.

- **mpiJava vs MPI F77 and C**

  It is not surprising that mpiJava has lower performance than MPI F77 or C since the performance of Java floating point arithmetic is slower than the performance of native C or Fortran code [69]. In addition, mpiJava uses wrapper routines to interface between the Java and the C routines that introduce additional overhead. The sequential and parallel Java/F77 and Java/C run time ratios of Metropolis and Swendsen-Wang are shown in Table 6.1 and 6.2 respectively. It shows that the performance difference between mpiJava and MPI F77 or C decreases as the number of processors increases. For Swendsen-Wang benchmarks, the MPI F77 and C codes are only about twice as fast as mpiJava version for the largest lattice size $L = 1024$.

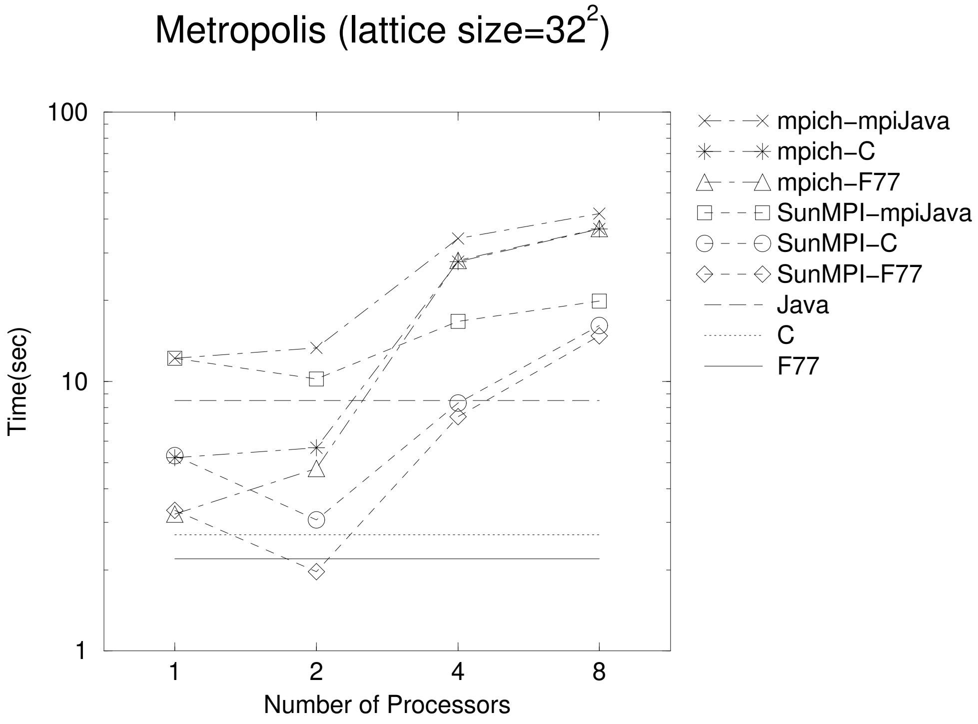

Figure 6.7: Metropolis Performance with Lattice Size 32^2.

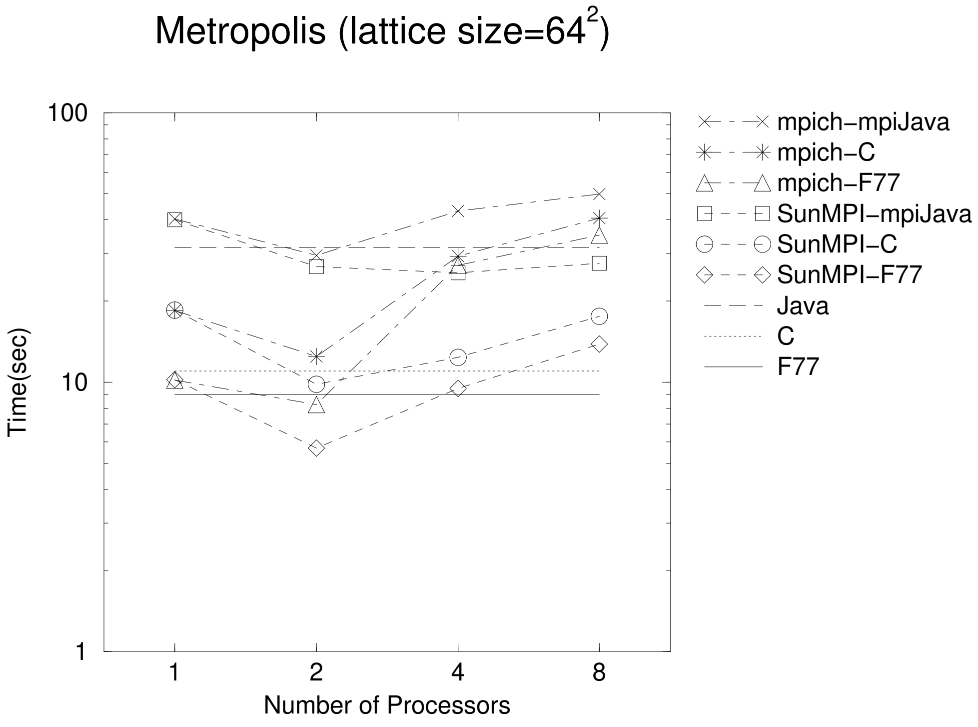

Figure 6.8: Metropolis Performance with Lattice Size 64^2.

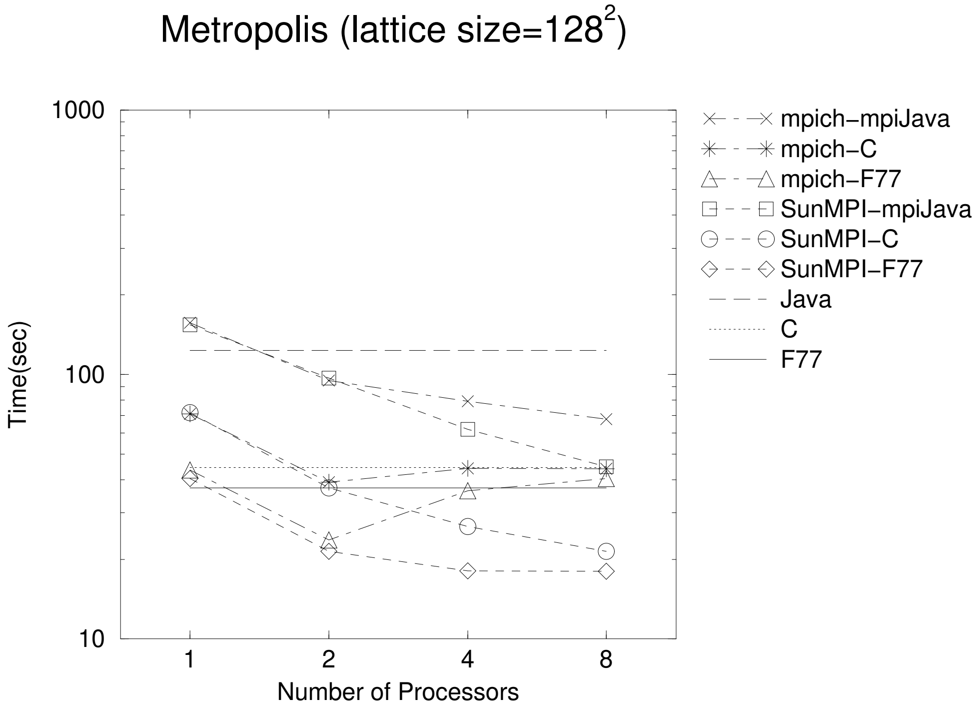

Figure 6.9: Metropolis Performance with Lattice Size 128^2.

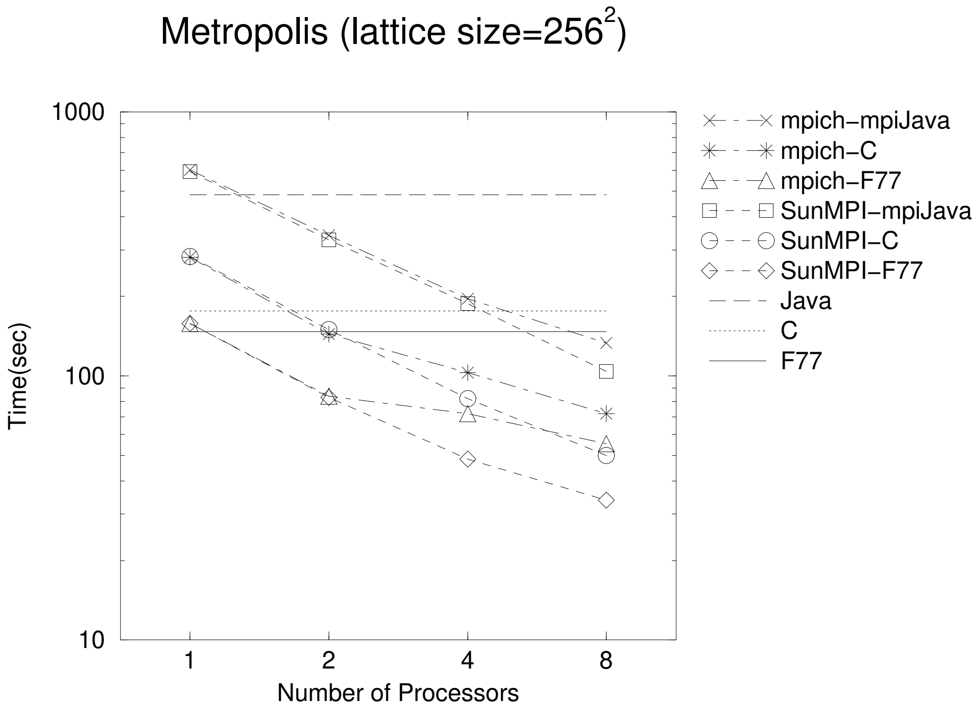

Figure 6.10: Metropolis Performance with Lattice Size 256^2.

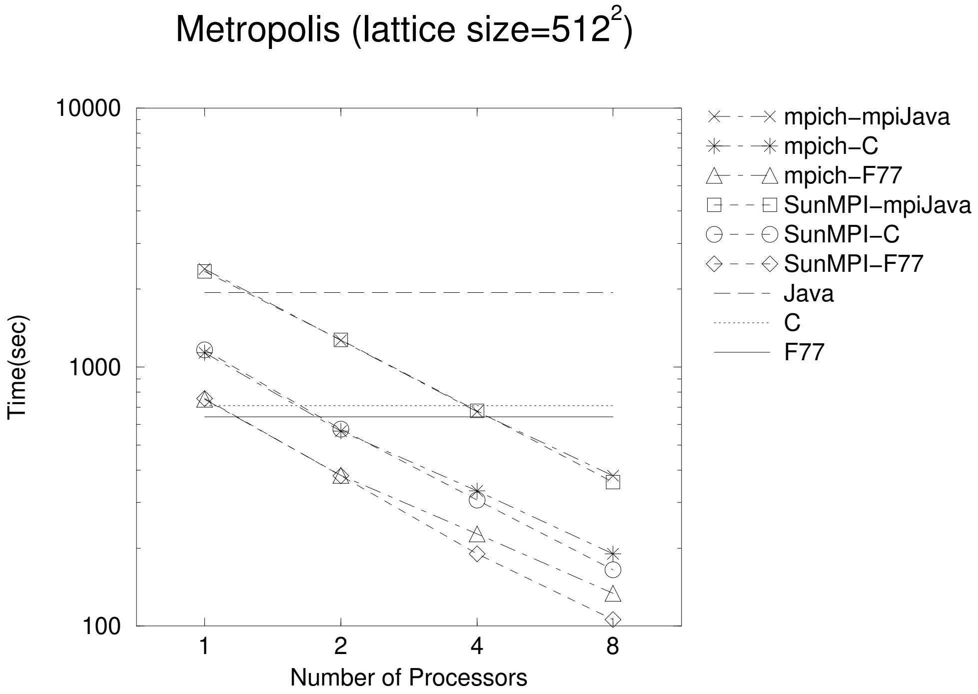

Figure 6.11: Metropolis Performance with Lattice Size 512^2.

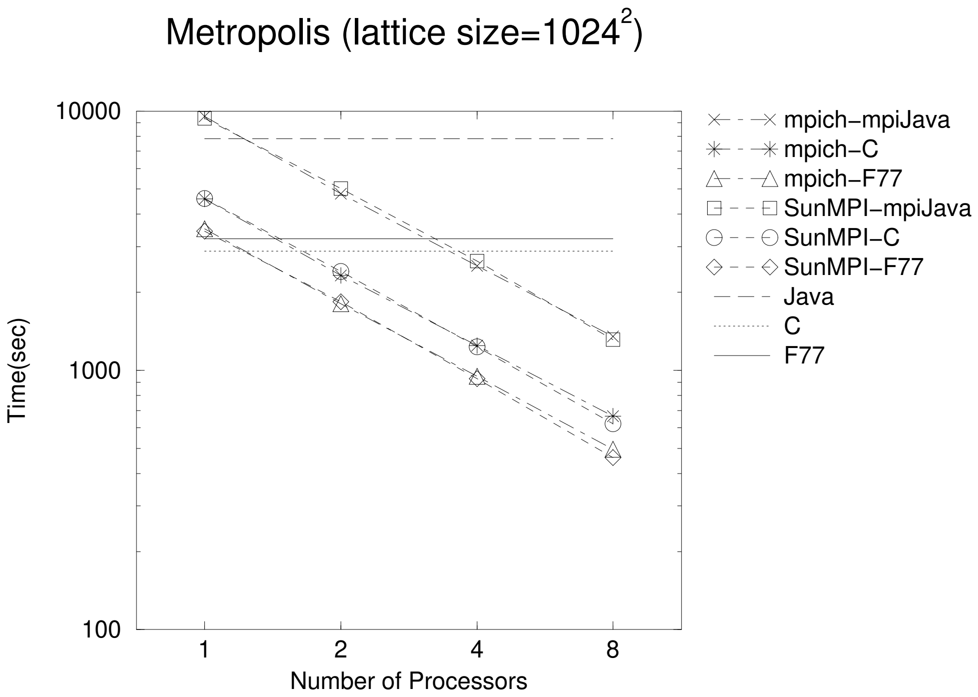

Figure 6.12: Metropolis Performance with Lattice Size 1024^2.

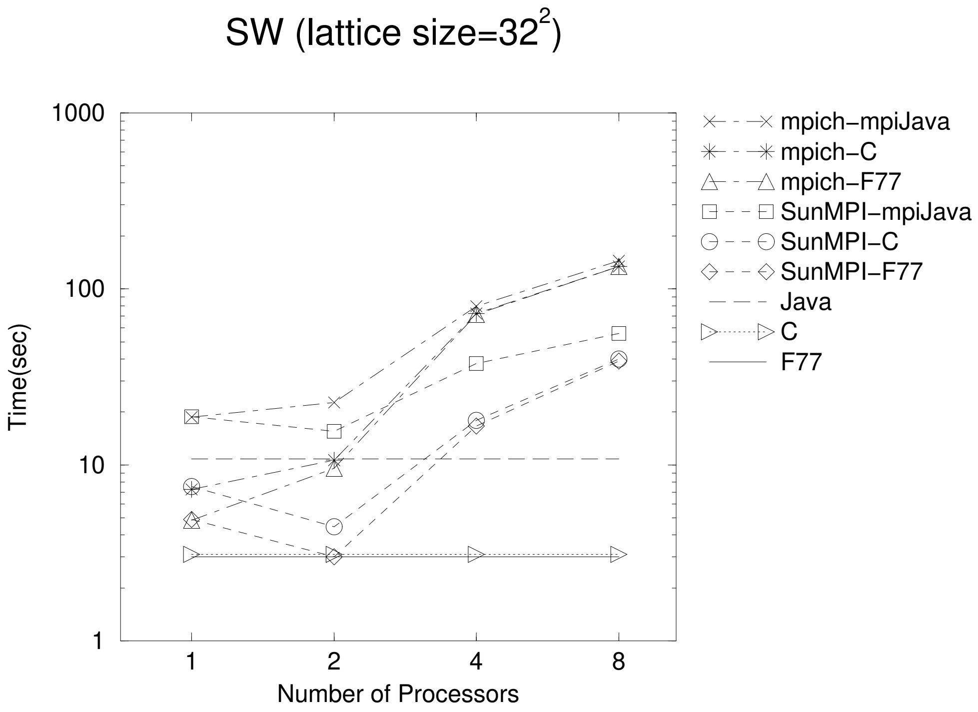

Figure 6.13: Swendsen-Wang Performance with Lattice Size 32^2.

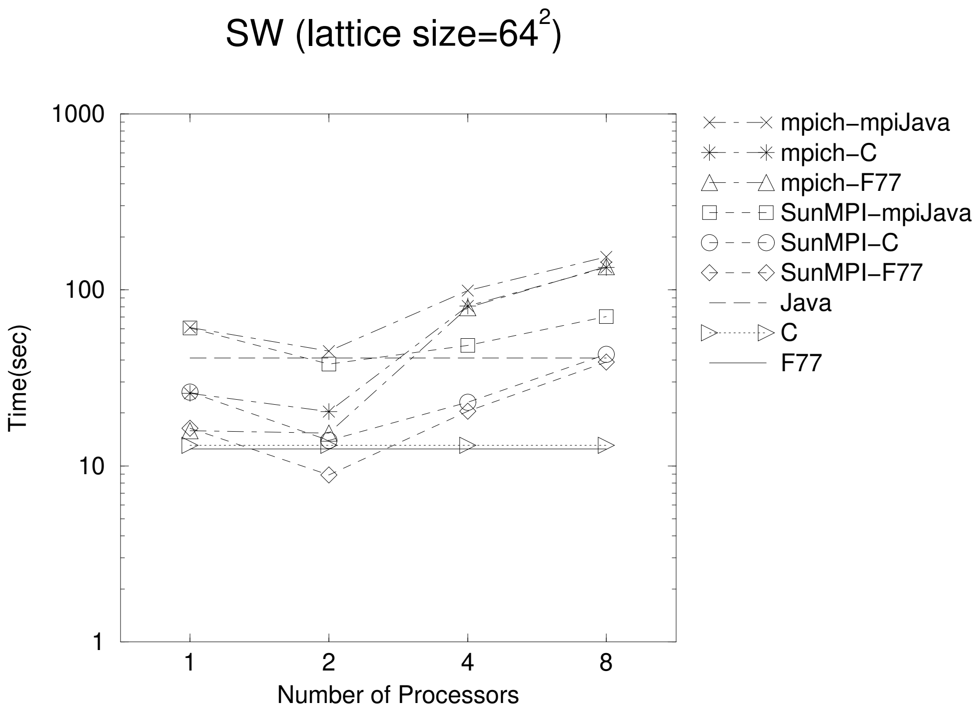

Figure 6.14: Swendsen-Wang Performance with Lattice Size 64^2.

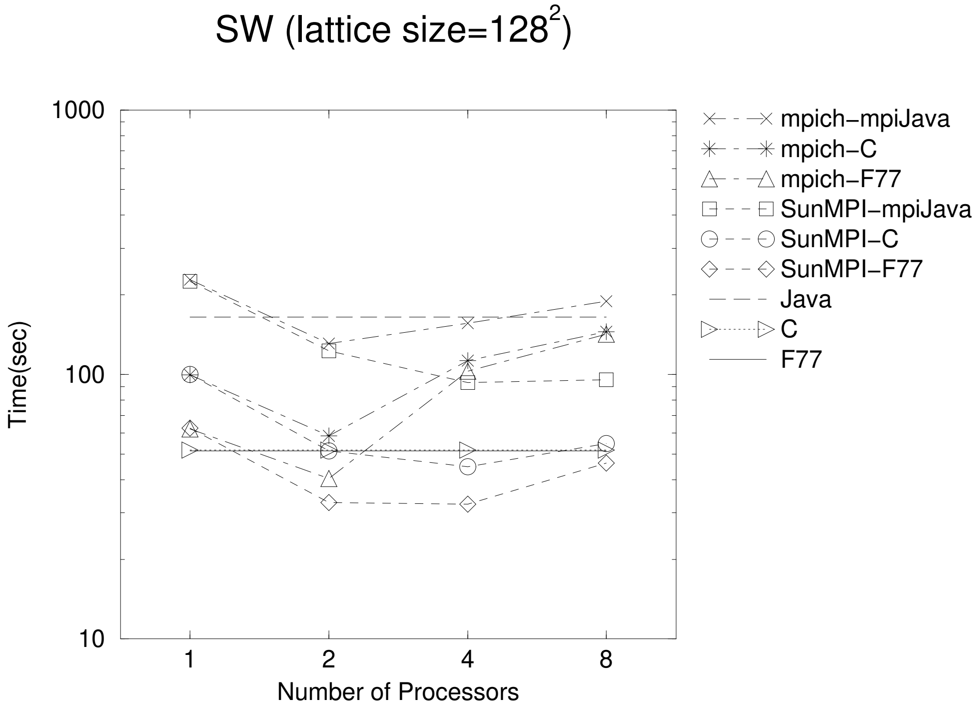

Figure 6.15: Swendsen-Wang Performance with Lattice Size 128^2.

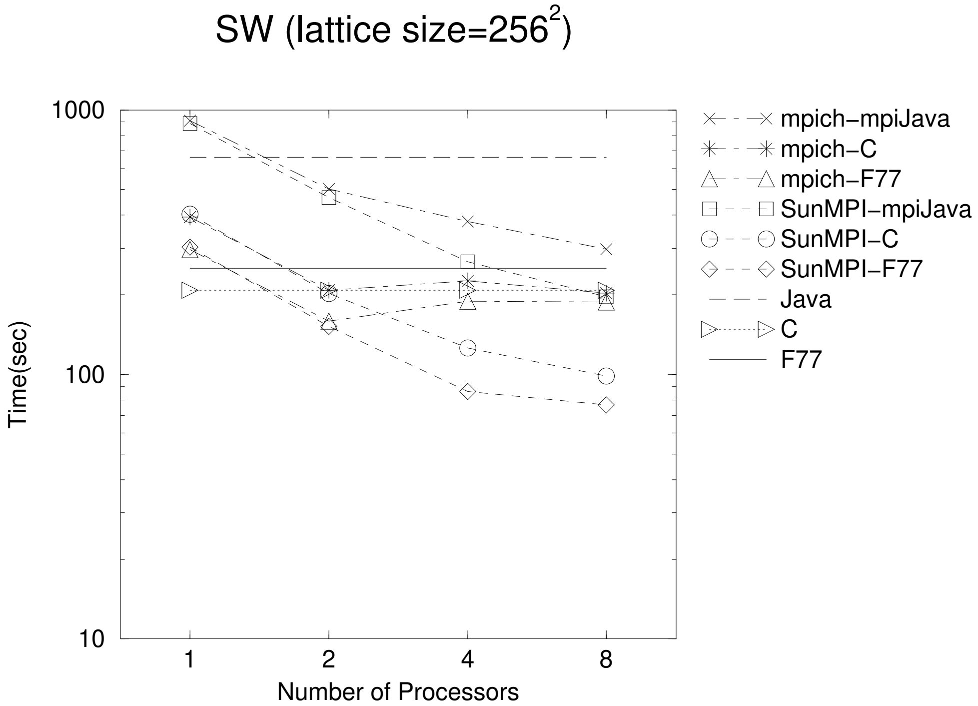

Figure 6.16: Swendsen-Wang Performance with Lattice Size 256^2.

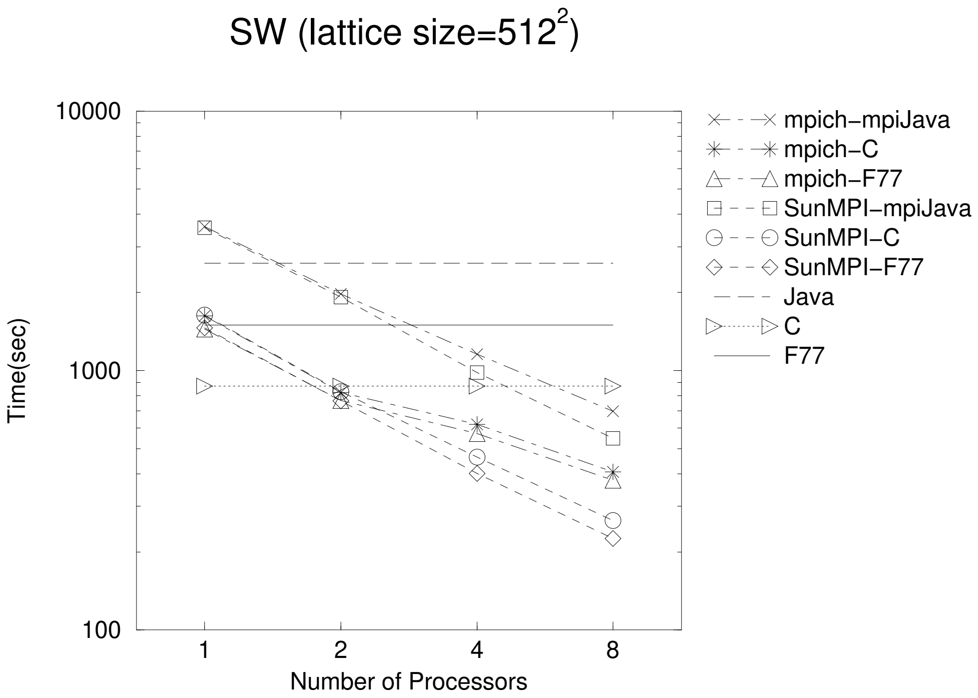

Figure 6.17: Swendsen-Wang Performance with Lattice Size 512^2.

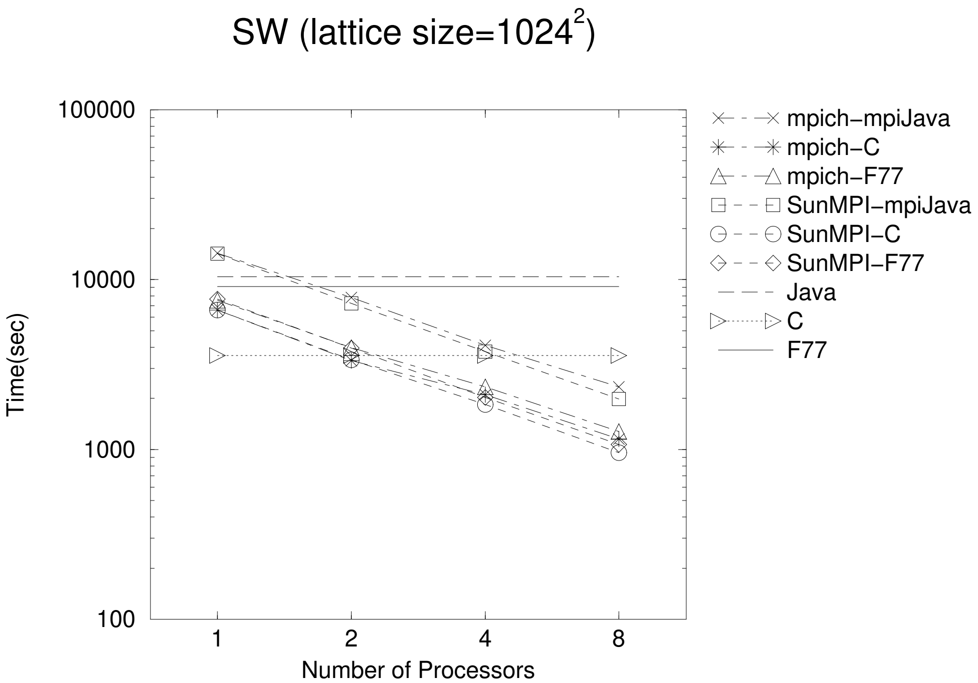

Figure 6.18: Swendsen-Wang Performance with Lattice Size 1024^2.

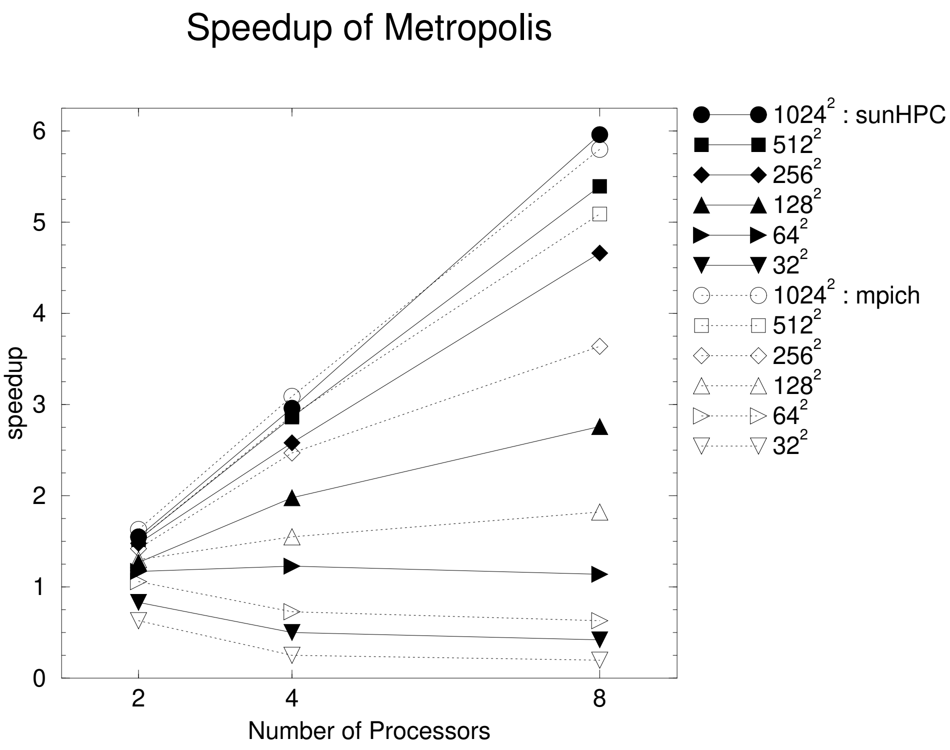

Figure 6.19: Speedup of Metropolis by using mpiJava as compared with serial Java.

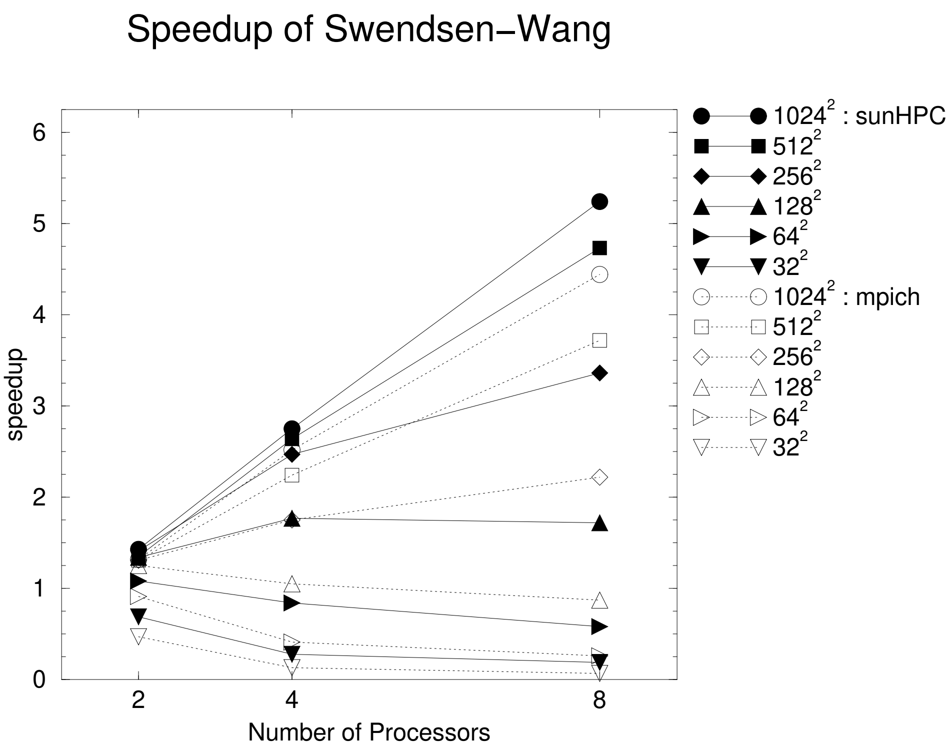

Figure 6.20: Speedup of Swendsen-Wang by using mpiJava as compared with serial Java.

	sunHPC		mpich		Sequential	
Nodes	Java/F77	Java/C	Java/F77	Java/C	Java/F77	Java/C
$L = 32^2$						
2	5.17	3.33	2.80	2.34	3.86	3.14
4	2.25	2.00	1.20	1.21		
8	1.34	1.23	1.13	1.14		
$L = 64^2$						
2	4.71	2.74	3.57	2.37	3.51	2.87
4	2.69	2.06	1.59	1.47		
8	1.99	1.57	1.42	1.22		
$L = 128^2$						
2	4.52	2.60	4.00	2.42	3.33	2.74
4	3.43	2.33	2.18	1.79		
8	2.48	2.08	1.67	1.54		
$L = 256^2$						
2	3.94	2.18	4.05	2.36	3.30	2.75
4	3.88	2.29	2.72	1.90		
8	3.07	2.08	2.40	1.84		
$L = 512^2$						
2	3.33	2.20	3.31	2.23	3.01	2.73
4	3.56	2.21	2.96	2.02		
8	3.38	2.17	2.83	2.00		
$L = 1024^2$						
2	2.73	2.08	2.66	2.07	2.43	2.71
4	2.84	2.14	2.67	2.03		
8	2.85	2.11	2.71	2.02		

Table 6.1: Metropolis...

	sunHPC		mpich		Sequential	
Nodes	Java/F77	Java/C	Java/F77	Java/C	Java/F77	Java/C
$L = 32^2$						
2	5.16	3.49	2.37	2.13	3.60	3.48
4	2.27	2.10	1.11	1.10		
8	1.43	1.39	1.08	1.08		
$L = 64^2$						
2	4.27	2.71	2.92	2.21	3.29	3.13
4	2.37	2.11	1.24	1.22		
8	1.80	1.64	1.14	1.15		
$L = 128^2$						
2	3.75	2.39	3.24	2.24	3.21	3.18
4	2.88	2.08	1.51	1.38		
8	2.07	1.74	1.33	1.30		
$L = 256^2$						
2	3.07	2.31	3.16	2.41	2.63	3.18
4	3.10	2.12	2.00	1.67		
8	2.57	1.99	1.59	1.47		
$L = 512^2$						
2	2.52	2.32	2.57	2.39	1.73	2.98
4	2.45	2.12	2.03	1.86		
8	2.44	2.08	1.84	1.71		
$L = 1024^2$						
2	1.83	2.15	1.98	2.35	1.14	2.90
4	1.87	2.04	1.76	1.96		
8	1.85	2.07	1.84	2.03		

Table 6.2: SW.....