Chapter 5 Parallel Computation in Physics

Due to fundamental speed limitation of single-processor computers, parallel computers emerge as new direction for scientific computations. Parallel computers offer much more computer power than single CPU computers and many more difficult problems can be solved on them. Parallel programming is more difficult, because that we are accustomed to serial computer and because nature of the problem does not permit a simple decomposition for parallel computation. Fortunately, many problems in physics can be parallelized efficiently. In this Chapter, we'll give some examples. For a general introduction to parallel programming, Ref. 1 is a nice textbook to look for. Ref. 2 emphasizes applications in Physics.

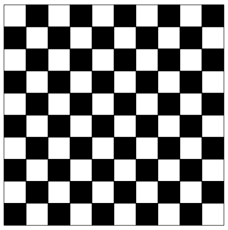
1. Replication Algorithms

A trial type of parallelism is that each processor runs exactly the same program with different input parameters. This is most suitable for distributed memory computers. For example, in Monte Carlo simulation, we can start the simulation with different random number sequence on each processor and generate more data points with many processors. Or each processor works on a different value of the model parameter, e.g. temperature T, so that we can get some quantity vs. parameter quickly. Since there is no communication between processors, except perhaps at the end of the run to collect data, this is in fact the most efficient parallel algorithm. If a computation takes time T on a single processor, the same amount of computation can be finished in time T/p, where p is the number processors. Thus, the speedup is p. The speedup is define as

$$Speedup = \frac{\text{Execution time on one single processor}}{\text{Execution time on a parallel computer with } p \text{ processors}}. \quad (1)$$

2. Multi-spin Coding of Ising Model Simulation

Parallelism has been used before the advent of parallel computers. In the simulation of Ising model, in combination with vectorization on supercomputers, people used bitwise operations to speedup the computation. There are several ways of using the bitwise operations. The basic idea is that the Ising spin can be represented by bits, taking 0 and 1 corresponding to spin up and down. On workstations, each computer word (integer) takes 32 bits, thus each word can be used to store 32 Ising spins. The calculation of the product $\sigma_i \cdot \sigma_j$ can be accomplished by the bitwise exclusive OR. In C we can write `i^j`. This single operation can be used to replace 32 multiplications. To utilize this feature, one can either simulate 32 copies of the same model, or use a checkerboard method to update the spin. We imagine that the spins are on the squares of a checkerboard, shown below. It is a valid algorithm if we update all the spins on the white squares at the same time and then update all the spins on the black squares. Note that the neighbors of a white square is all black and vice versa.



3. Geometrically Parallel Algorithms

In geometrically parallel algorithms we divide the whole region into small domains, each of the domain is handled by one processor. Many problems in physics and mathematics can be expressed as local rules with short-range interactions. For example, the updating of the spins in Ising model depends only on the states of the nearest-neighbor spins and solving Laplace equation or diffusion equation can be done locally. In these situations, efficient parallel algorithms can be devised by partitioning the region.

3–1. Parallel Metropolis algorithm for Ising model

We'll use two-dimensional Ising model again to illustrate the concept of geometrically parallel algorithms. There are three commonly used partition of the spins: (1) checkerboard, where each processor has just one spin. If the sites are denoted by (i,j), the black or white squares are distinguished by $i+j$ even or odd. All the even sites are updated in parallel, followed by the updating of the odd sites. This kind of algorithms is suitable for massively parallel computer with lot of processors like Connection Machine, but is inefficient on workstation clusters. (2) Divide the $L \times L$ system into small squares of some size. Each processor is assigned to one square. Connections between processors are needed when updating the four boundary sites. (3) the last one is to divide the $L \times L$ region into strips of $L \times (L/p)$, where p is the number of processors, and L should be a multiple of p.

Let us discuss method (3) in some detail. If we update the spins in a strip by choosing the site at random or sequentially without special care, we might simultaneously update two spins who are nearest neighbors at the boundary, while our calculation of the energy change is based on a single spin flip. This violates the detailed balance condition, and we get wrong results. To overcome this problem, we'll update the spin row by row, making sure that we always use the new spin values for updating. Each processor works cooperatively (rather than independently) with other processors. In other words, the processors will synchronize. This synchronization is easily achieved on distributed memory parallel computer with a message-passing mechanism.

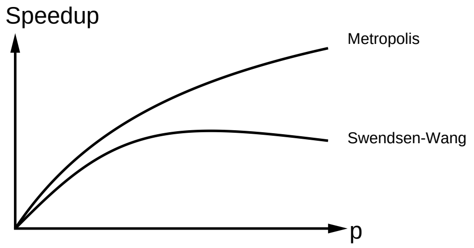
For convenience, we'll duplicate the row of spins in the up and bottom strips. This makes the calculation of energy change easier. Let's say that the width of the strip is $(L/p) = 10$, $L = 100$, then the first processor contains the row 99, 0, 1, 2, ..., 9, 10, and the second processor contain the row 9, 10, 11, 12, ..., 18, 19, 20, and similarly for third and fourth row, etc. We could say that the rows 0 to 9 really belong to the first processor, row 99 and 10 are copies of the bottom and top strips.

We'll update the spins row by row, in a particular order. One Monte Carlo step consists of

1) Send row of spins on bottom boundary downwards, the last strip's bottom row is sent to the top of first strip (cyclically).
2) Receive a row of spins on top boundary cyclically from strip above.
3) Update the top row of spins.
4) Update spins in interior rows.
5) Send spins on top boundary cyclically upwards.
6) Receive spins on bottom boundary cyclically.
7) Update bottom row of spins.

The above steps are performed on all processors. The algorithm above applies to any multiple instructions multiple data (MIMD) machine. To program it on cluster of workstations with PVM, it is convenient to use the master/slave model. The master program spawns the slave processes and distributes the work to the slaves, and collects data, while the slave programs do the Monte Carlo simulation.

Since message passing is very slow on cluster of workstations and on real distributed memory parallel computers, one should minimize communication between processors. As long as the granularity of the computation is sufficiently large compared to the communication delays, the above type of algorithms can achieve good performance. Thus, for relatively large system L (keeping fixed), increasing the number of processors p leads to linear speedup in the beginning. However, as the number of processors increases, the computational work of each processor decreases and communication overhead increases. Eventually the speedup saturates to some value. The qualitative behavior is shown in the following figure. The curve labelled Swendsen-Wang refers to Swendsen-Wang algorithm which we'll discuss in the next few subsections.



We can analyze the performance of the above algorithm in some detail. For a single processor, one Monte Carlo step takes time $T_1 = \tau L^2$, where τ is the time for one single trial. On a parallel computer with p processors, each processor is responsible for L^2/p spins, this takes time $\tau L^2/p$. In addition, the message passing takes time. How much time it takes is highly dependent on the architecture of the communication hardware. Let's assume that the processors are connected by a ring topology, then each exchange of data between the neighboring strips takes same time. This time is proportional to the amount of data transferred, in our case, L, plus a constant for communication overhead. Thus the time it takes for one Monte Carlo step on a parallel computer with ring connection is $T_p = \tau L^2/p + bL + c$. The speedup is then

$$speedup = \frac{T_1}{T_p} = \frac{p}{1 + p(bL + c)/(\tau L^2)}. \tag{2}$$

3–2. SIMD parallel Swendsen-Wang algorithm for Ising model

The single-spin-flip Metropolis algorithm can be parallelized rather efficiently for large systems. The reason for this is that the updating of the spins needs local information only. Only the processors working on neighboring strips need to communicate. Nonlocal algorithms like the Swendsen-Wang algorithm are harder to parallelize due to the fact that all processors need to communicate with each other.

The most time-consuming part of the cluster algorithm is the cluster labelling. On a Single Instruction Multiple Data (SIMD) parallel computer like CM-2 Connection Machine, the labelling can be done by propagation (ref. 3). We start with a different label on each site. Each site then looks to each of its neighbors in turn. This is done in parallel. If the site is connected to its neighbors, and if the neighbor's label is smaller than its own label, then it replaces its label with that of its neighbor. This procedure is repeated until there is no change to the labels, at which time each cluster will be labelled by the minimum initial label of all the sites in that cluster.

When the clusters are of the size of the system, this algorithm is not very efficient, since it take of order L iterations for the labels to converge. It is possible to improve the algorithm so that each labeling takes time of order $(\ln L)^2$. In the improved method, one looks nearest neighbors, then neighbors of distance 4 apart, distance 8 apart, etc. This speeds up the labelling.

3–3. MIMD parallel Swendsen-Wang algorithm for Ising model

As in the Metropolis algorithm, we divide the system into strips. In the Swendsen-Wang algorithm, the crucial part is to identify the clusters. For each strip, we use the usual methods of cluster labelling algorithms, assuming all the bonds between strips are breaking. When this is done, we must consider the connections between strips. Because of the global feature of the clusters, this part cannot be done efficiently in parallel. One method is to build a global list pointers to the clusters by the master processor while the slave processors are waiting. The result of the list pointers is broadcast back to the slave processors. They use the information to update the spins reside in their processors.

Let's give a rough analysis of the computational complexity. On a single processor, we can perform one Monte Carlo step in a time proportional to the number of spins L^2. On a parallel computer with p processors, each processor works on a strip, this will take time proportional to L^2/p for local cluster labelling and updating of the spins. However, the final global cluster labelling is essentially done in serial. The amount of operations is proportional to the number of clusters which are at the boundaries of the strips. Information of the these clusters (which cluster is connected to which) at the strip boundaries have to be sent to the master processor, and the labeling takes a time proportional to the number of clusters involved. Clearly the number of clusters at the boundaries depend on the temperature of the system. It can vary between p (when there is a single cluster when finally labelled) and pL (at the infinite temperature). In any case, it is proportional to p. Thus we have the total parallel computer time $T_p = \tau L^2/p + \alpha p$. The speedup is then

$$speedup = \frac{p}{1 + \alpha p^2/(\tau L^2)}. \tag{3}$$

We get good speedup for large system size L with small number of processors. As we add more processors, the speedup reaches a maximum and drops to smaller value. It may happen that for some large value p, it takes more computer time on a parallel p processor computer than on a serial single processor. This is due to the global nature of the cluster algorithms. It is still a challenge to find good parallel algorithms for cluster labelling.

4. Data Parallel Algorithms

Load balance is important for parallel computation. It is most efficient if each processor has roughly the same workload. For Ising model, a geometric decomposition of the problem is most appropriate. We can also use a geometrically parallel algorithm for molecular dynamics. In such a situation, it is possible that some of the processors need to handle a lot of particles while other processors have just a few particles. Since the time t has to be synchronized, the speed of the simulation is controlled by the slowest processor. Geometric parallel algorithms is suitable for system with short-range interactions. The Lennard-Jones potential is short-ranged, while the electrostatic or gravitational interaction ($F \propto 1/r^2$) is long-ranged. For long-range interactions where each particle is interacting with other particles, data parallelism is more appropriate. On a single processor, the time complexity for such N-body problem is N^2, where N is the number of particles. On a parallel computer with p processors, we can reduce it to N^2/p with data parallelism.

Data parallel algorithms divide data into equal parts for each processor. This achieves good load balancing in a straightforward way. Taking the molecular dynamics algorithm as example, we assign $n = N/p$ particles in each processor at random. The data presenting the particles will stay in the same processor during the whole simulation. We'll call these n particles in a given processor local particles. The spatial separation of the particles associated with any particular processor is unimportant. The processors are connected in a ring topology. The computationally intensive calculation of the force proceeds in the following way. Each processor sends the mass, position, and momentum of a particle to the next processor in forward direction around the ring and at the same time receives a set of variables of a particle from the neighbor processor in other direction. The force between this travelling particle and all the local particles are then computed. The travelling particle is then send to the next processor in the ring in a cyclical fashion and the whole procedure is repeated until the complete ring of processors has been visited. This entire procedure must be carried out for every one of the n local particles in each processor and the forces acting on each particle due to all other particles are accumulated in order to update the particles one time step in the usual fashion.

This algorithm will provide an efficient means of parallelizing the problem as long as the time spent in communication of the particle masses, positions, and momenta is small in comparison with the calculation of the forces. This can be arranged simply by ensuring that the granularity is not too fine, i.e., that there are a sufficient number n of local particles on each processor. In such a case this algorithm proves to be extremely effective.

Let us make a more quantitative analysis of the performance of the algorithm. On a single processor, the most time-consuming part is the calculation of the interacting forces between every pair of particles. Since there are $N(N-1)/2$ pairs, we need to compute the distances this many times. The updating of the particle state takes time of order N. Thus the computational complicity for long-range molecular dynamics is $O(N^2)$. On a distributed memory parallel computer with ring topology, each processor have N/p particles, we must calculate the force between the particles in the processor and that of a travelling particle. Since there are N travelling particles (each particle traverses the ring exactly once), the computation for the forces takes time proportional to N^2/p. In addition, there is communication overhead, which is proportional to N, since each processor received N particles and sent to its neighbor N particles. Thus the total time is $T_p = \tau N^2/p + bN$. The speedup is similar to that of Metropolis algorithm for Ising model.

5. Parallel Algorithms with Relicated Workers

The geometrically parallel algorithm and data parallel algorithm all have a static distribution of the data. The amount of work for each processor is decided in the beginning and will not change later on at the runtime. Large class of problems in physics fits into this category. However, there are certain problems in which the amount of work is not known beforehand, it is generated during runtime dynamically. Many combinatorial problems and algorithms on graph (short path on a graph, travelling salesman problem) belong to this catagory. Even for geometrically decomposable problem, we might want to allocate the mount of work for each processor dynamically. For example, on our workstation cluster, the actual running speed is not the same even though the hardware is identical. This is because someone may use the workstation at the moment. A better method is to give a lighter work on the heavily loaded processor, and give more work on free processors.

The idea of replicated workers is to construct a data structure called work pool and n identical worker processes. Each worker gets a task from the work pool and does its computation based on the data taking from the work pool. The worker may also generate new tasks, which is sent back to the pool. As long as there are enough tasks in the work pool, all the workers (processors) are busy doing computation in parallel. The computation terminates when the work pool is empty and all the workers are idle. As you might have thought, the actual implementation of the idea is somewhat complicated.

References

1. B. P. Lester, *The Art of Parallel Programming*, (Prentice Hall, 1993).
2. D. W. Heermann and A. N. Burkitt, *Parallel Algorithms in Computational Science*, (Springer-Verlag, 1991).
3. J. Apostolakis, P. Coddington, and E. Marinari, International. J. Mod. Phys. C, **4**, 749 (1993).

LABORATORY EXERCISE 4.

1. Implement a parallel Monte Carlo simulation program with PVM for the two-dimensional Ising model, using the single-spin-flip algorithm and strip geometric decomposition. Each of the processor handles a strip of $L \times (L/p)$, L is a multiple of p. The processors are assumed to have a ring connection topology. Periodic boundary condition should be used. To test your program, calculate the average energy per spin. The exact value of energy per site at the critical temperature ($k_B T_c/J = 2/\ln(\sqrt{2}+1) \approx 2.2691853$) is $-1.453\,J$ on a 16×16 system. You may also compare your parallel program data with earlier data of single processor Ising model simulation program.

TUTORIAL PROBLEM 4.

Please spend a little time to work on the problems before attending the tutorial classes.

1. Propose parallel algorithms for the one-dimensional Ising model with periodic boundary condition. The single-spin-flip Metropolis algorithm for Monte Carlo simulation should be used. What you might do in each case if the parallel computer is shared memory type, or with lot of processors like CM-2, with slow network connection like workstation cluster?