Parallel Two-dimensional Lattice Monte Carlo Simulation

Amina Mirsakiyeva
Fan Pan
Karim Elgammal

January 30, 2014

1 Introduction

Parallel programming is a useful and necessary tool in modern computing. A simultaneous use of several processors allows solving of a problem faster with access to a larger amount of memory which is not possible for serial computing. During our report we will look through two main ways of parallel programming: OpenMP (Open Multi-Processing) and MPI (Message Passing Interface).

OpenMP is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and FORTRAN, on most processor architectures and operating systems, including Solaris, AIX, HP-UX, GNU/Linux, Mac OS X, and Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.[1]

While **MPI** is a standardized and portable message-passing system defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in FORTRAN or the C programming language. In a distributed-memory architecture, each process doesn't share the same address space as the other process (which very possibly run on different machine). This means each process cannot "see" the other process variable(s). The process must "send a message" to the other process to change variable in the other process. Hence, the "Massage Passing Interface (MPI)". The MPI library such as OpenMPI basically is a sort of "middleware" to facilitate the massage passing between the processes, the process migration, initialization and tear-down.[2]

Meanwhile in a shared-memory architecture, there is usually one process which contains couple of threads which share the same memory address space, file handles and so on. Hence, the shared memory name in this architecture, each threads can modify a "precess" global data. Therefore, a semaphore mechanism must be in use. OpenMP simplify the programming for shared memory architecture by providing compiler "extensions" in the form of various standardized "pragma" s.

Consequently, it is obvious why OpenMP is more useful for "local execution within a machine" while MPI is more efficient for inter-machine process communication.

Although the efficiency of OpenMP compared to MPI varying from different cases, we found a particular example[3] of illustrating the efficiency of MPI, which is higher than that of OpenMP (script in appendix A) on calculating a simple integration. This example motivates us to focus on MPI in our project with a suitable problem. The success of MPI would give more freedom in programming and be able to use as many nodes as it possible.

2 Problem

Monte Carlo (MC) methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. Specifically on what our code does is to calculating the critical properties of two-dimensional Ising model, a model of describing the magnetic system in which spins constraining in a 2D plane lattice with freedom spin up and spin down. In such a model the ordering-disordering phase transition will be observed at critical temperature, and other interested quantities such as magnetization, susceptibility, coherence length will be obtained from the configuration of spin lattice.

From computing point of view, applying Monte Carlo method however in general would cost a lot of computational effort (memory and time). Take an example of 2D Ising model, the larger lattice domain and more iteration being calculated (meaning that more random sampling included) the better agreement to the analytic results can be found. Whereas in practice it is not possible to calculating infinite number of lattice, apart from that the periodic boundary condition will be needed and applied on the system, and this assumption will cause the deviation from its exact results.

As the high-performance computing (HPC) architectures applied in modern scientific researching, many software packages have already been self-updated in order to take the advantages of HPC. The MC method fortunately is one of those that not only possible but also suitable to be parallelizing. Concerning about the complexity of using MPI, we selected a rather simple problem that domain has certain symmetry to be divided evenly for all processors, with not much extra work load for the master processor; On top of that there is no particular time-depended procedure, therefore the ordering of passing massages between processors would be able to designing in the same way.

3 Implementation strategy

In any parallel computing, a crucial part is to decompose the computing load and later to be assigned to the working processors.[4] A simple scheme of spatial decomposition can be seen from Fig.1. In such a way the whole domain would be cut into row-strips like, each subsystem has two augmented rows which used to copying the spin states from the top and bottom rows of neighboring processors. (e.x. the bottom-rowspins in processor 2 interact with the top-row spins in processor 3.)

In order to passing the messages through the communication between processors, each processor carried an identity, and specifying in any sending / receiving address. The index of row and column number then changed to local index within one processor.

One risk which might take in such parallel MC computing is that the spins flipped between two processors are neighboring, even though it is only occurred at small chance, the consequence of the violation to the Metropolis algorithm will be observed in long term. By voiding the shortage of updating from the augmented domain one has to synchronize data rapidly, and this effect will lead to a low efficiency when large number of processors applied. The remedy for solving this problem is instead of randomly flipping one spin at any possible lattice one can make a order to flipping spins row by row. There are other methods more advanced, for example one can "frozen" the boundary of each processors and after a while "defrozen" that area and make a new boundary area alternatively. In our code we implemented the simple "sweeping in order" method.

Another problem will be faced is the so called "dead-lock" avoidance. As a circular message has been passed, it could cause a deadlock. Instead of a circular send-receive relation, we separated into two steps: first round is sending messages from even processors to its neighboring (odd of course) processors, then the second round is passing messages from odd to even processors Fig.2

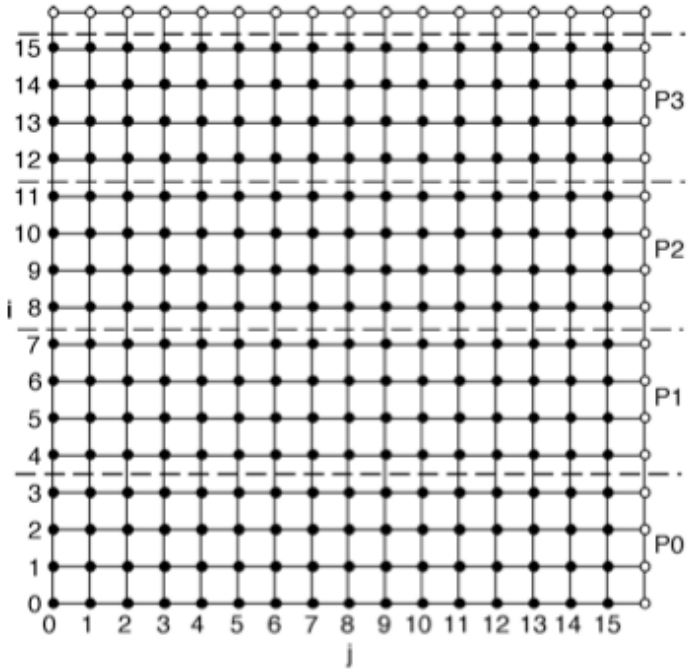


Figure 1: 16*16 grid points are divided into 4 subsystems

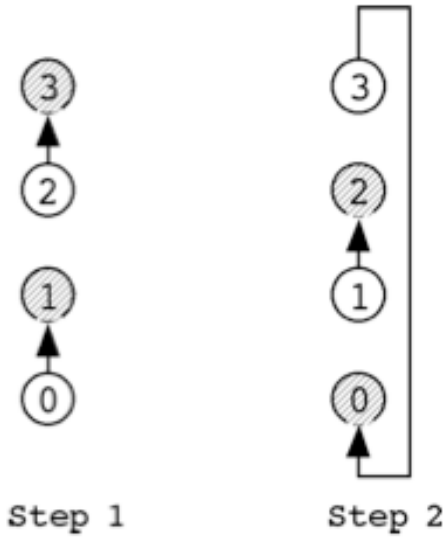


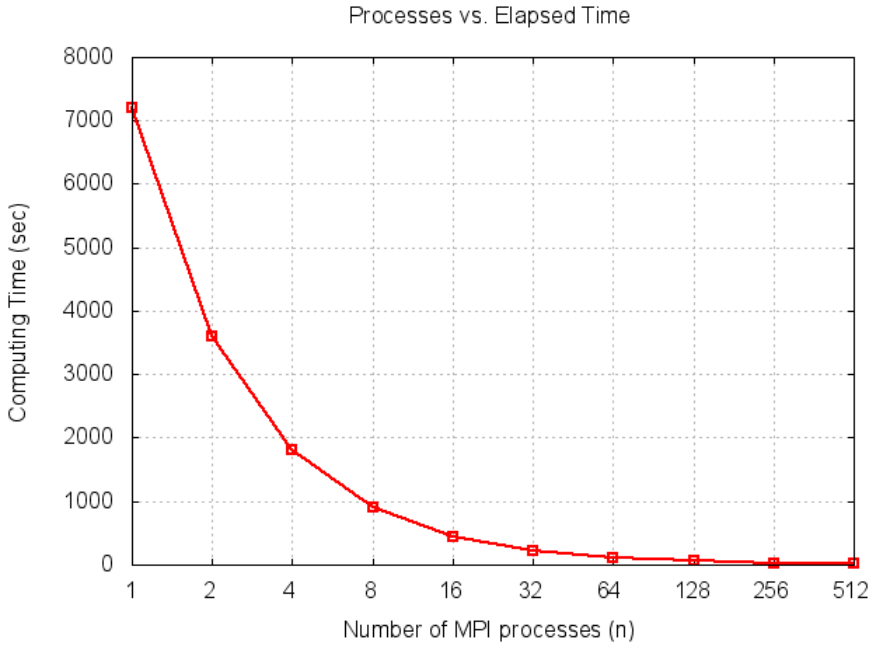
Figure 2: Avoiding circular message passing

4 Result and analysis

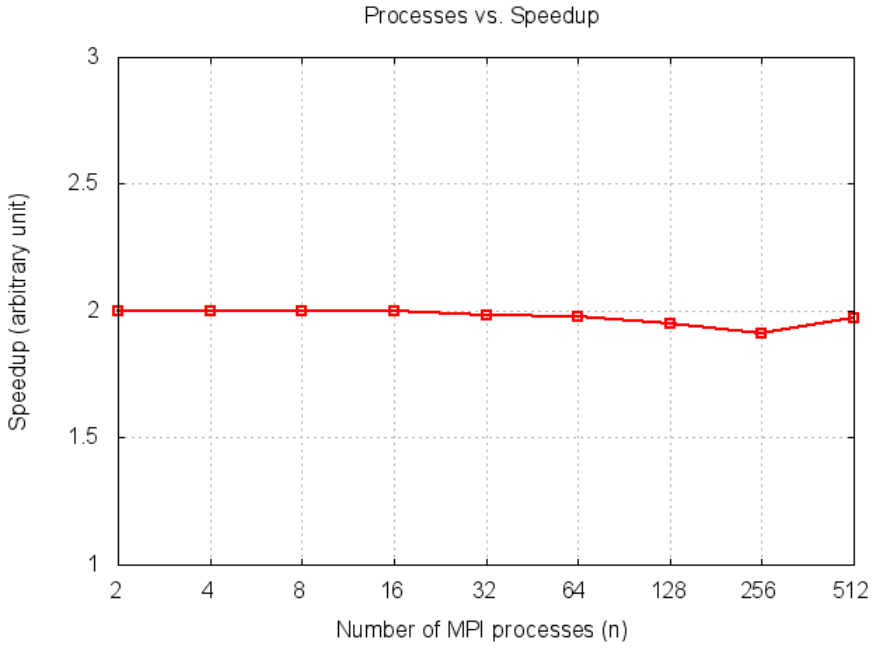
Here, we operate the code with different number of processes acting on a specified cell size of (1024x1024) tested against specific number of processes as shown in table (1). The computing wall times are shown in fig (3a) as a function of the number of processes while, the corresponding speedups are available in figure (3b). So, Figure (3) show a positive view of paralleling performance, particularly as one can seen on the speedups keeping roughly as a factor of 2 upon doubling the number of processes. However, it would seem to be necessary also testing of even larger domain size. This disability is possibly because of the extremely long computing time which can not be fitted in the job requiring wall time of less than 24 hours.

Processes	Computing Time (sec)	Speed-up
1	7206.818996906281	base
2	3603.401446104050	2.000004469304467
4	1802.526098966599	1.999084200872268
8	901.9490389823914	1.998478873041728
16	450.9236099720001	2.000225801080581
32	227.3321640491486	1.983545143548239
64	114.9379720687866	1.977868235861145
128	58.86776494979858	1.952477254178135
256	30.80744099617004	1.910829431016908
512	15.61976718902588	1.972336759142908

Table 1: Compputing wall time for specific number of processes acting on cell size of (1024×1024).



(a) Computing time for processes applied on cell size of 1024×1024)



(b) Speedups corresponding to processes on cell size 1024×1024)

Figure 3: Computing time and speed-ups

4.1 Performance Analysis

We have analyzed our code using CrayPat performance analysis tool,[5] and we obtained both graphs sorted according with cells as well as time, shown in Fig 4. The balance of MPI_send and MPI_recv might not be looked surprising, it is due to the one-to-one massage sending and receiving conditions between the boundary of each processor's domain cell to the neighboring cell. The drawbacks of only using "blocking sending" instead of "nonblocking" sending" can be clearly seen from sort by time in which up to 50% time is occupied by synchronization.

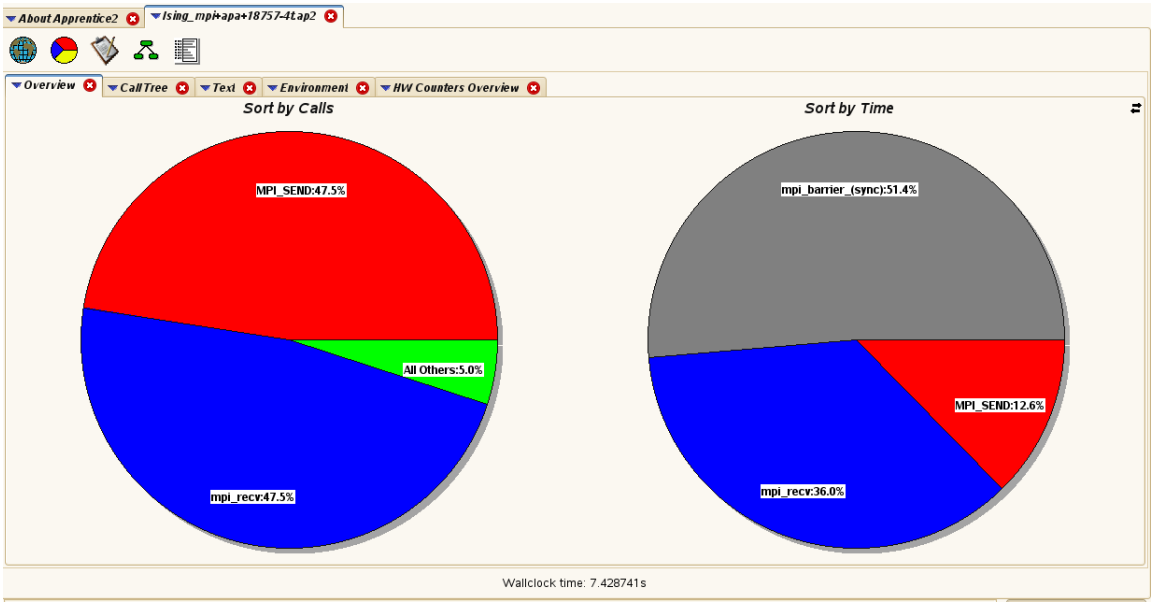


Figure 4: CrayPat performance analysis

5 Conclusion and outlook

To conclude, we have applied the basic knowledge and skills learned from PDC summer school on paralleling a serial Monte Carlo code of two dimensional Ising model using MPI. From learning point of view, in the first part we achieved to make paralleled code working on a rather large domain size (1024*1024) without sacrificing its accuracy; in the second part the paralleled code was tested under the PDC-summer_school_2013 lindgren supercomputer, in region of using 1 to 512 processes the required wall time is continuing decreasing, and the speedup scales linearly. Applying CrayPat tool on the code might be a trivial case, but it helped to showing the actual weakness in the blocking way of message communication.

Although the paralleled code performed with some sense of successful, the simplicity of no-time-depended and indistinguishable between master / working processes are the main reasons contributed to such a success. Further optimization could be done by using non_blocking sending, separating code into several subroutines with clearer structure and making full use of CrayPat tool, learning advanced way of decomposing domain to accessing large number of processes.

Finally, we would thank for all the organizers and invited speakers of PDC summer school. Without their effort it wouldn't give chance for students acquainted with a comprehensive and solid basis of high performance computing. It was both a nice experience and challenge for us, as we realized the more we learned the more we need to learn.

A An example of comparing OpenMP and MPI

To compare OpenMP and MPI let us look on simple example:

$$\int_0^{\pi} (x^2 \sin(x) \cos(x) + 2) dx$$

The result for OpenMP is:

```
Number of quantization steps 500M
Start working thread # 0 of 2
Start working thread # 1 of 2
Result with OpenMP: 3.815784

real 17.042s
user 33.700s
sys 0.020s
```

and for MPI:

```
Start working
Number of quantization steps 500M
Start working
Job is done!  [250000000; 500000000] Result = 0.307341
Job is done!  [0; 250000000] Result = 3.508443
Result with MPI: 3.815784

real 17.342s
user 0.090s
sys 0.030s
```

```
============================================================
#include <omp.h>
#include <stdio.h>
#include <cmath>
#include <string>

// Integral limits [a;b]
const double a = 0;
const double b = Pi;

// Number of quantization steps in [a,b]
const int N = (500*1000*1000);
```

```
// Quantization step const double step = (b - a) / N;

// Integral evaluation function
static inline double f(double x)
{
   return cos(x)*sin(x)*x*x + 2;
}

double evalIntegral()
{
   double result = 0;
# pragma omp parallel for reduction(+:result) schedule(dynamic,(N/1000))
   for(int i = 0; i < N; i++)
   {
      double x = a + i * step;
      result += f(x) * step;
   }
   return result;
}

int main(int argc, char **argv)
{
   printf("=> OpenMP\n");
   printf("Number of quantization steps %iM\n", N / 1000000);
   fflush(stdout);

# pragma omp parallel
   {
# pragma omp critical
      {
         printf("Start working thread #%i of %i\n",
         omp_get_thread_num(), omp_get_num_threads());
      }
   }

   double result = evalIntegral();
   printf("Result with OpenMP: %lf\n", result);
   return 0;
}
```

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <string.h>

// Integral limits [a;b]
const double a = 0;
const double b = pi;

// Number of quantization steps in [a,b]
const int N = (500*1000*1000);

// Quantization step
const double step = (b - a) / N;

// Integral evaluation function
static inline double f(double x)
{
   return cos(x)*sin(x)*x*x + 2;
}

double evalIntegral(int start, int end)
{
   double result = 0;
   for (int i = start; i < end; i++)
   {
      double x = a + i * step;
      result += f(x) * step;
   }
   return result;
}

int main (int argc, char* argv[])
{
   MPI_Init(&argc, &argv);
   int rank;
   int sizeOfWorld;
   MPI_Status mpiStatus;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &sizeOfWorld);

   static const int MasterRank = 0;

   if (rank == MasterRank)
```

```
    {
      // Master
      printf("=> MPI\n");
      fflush(stdout);
    }

    printf("Start working\n");
    fflush(stdout);

    int numOfWorkers = sizeOfWorld;
    int start = N / numOfWorkers * rank;
    int end = start + N / numOfWorkers;

    double result = evalIntegral(start, end);
    printf("Job is done!  [%i; %i] Result = %lf\n", start, end, result);
    fflush(stdout);

    if (rank != MasterRank)

    {
      // Workers send results to master
      MPI_Send(&result, 1, MPI_DOUBLE, MasterRank, 0, MPI_COMM_WORLD);
    }

    // Master collect results
    if (rank == MasterRank)
    {
      double resultFromWorker = 0;
      for (int id = 1; id < sizeOfWorld; id++)
      {
        MPI_Recv(&resultFromWorker, 1, MPI_DOUBLE, id,
          MPI_ANY_TAG, MPI_COMM_WORLD, &mpiStatus);
        result += resultFromWorker;
      }
      printf("Result with MPI: %lf\n", result);
    }
    MPI_Finalize();
    return 0;
}
```

References

[1] "Openmp." http://openmp.org/wp/about-openmp/.

[2] "Mpi." http://http://www.open-mpi.org/.

[3] "Openmp vs mpi." http://brunql.blogspot.se/2011/06/openmp-vs-mpi-vs.html.

[4] "Parallel monte carlo simulation." http://cacs.usc.edu/education/cs653/07-1ParIsing.pdf/.

[5] "craypat document." http://docs.cray.com/cgi-bin/craydoc.cgi?mode=Show;q=;f=man/xt3_patm/41/cat1/craypat.1.html.