Multithreaded Parallel Computing on Steele Cluster

Tzu-Cheng Chuang and Okan K. Ersoy
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907

Abstract

The Steele cluster was established at Purdue in May 2008. It is connected with Teragrid system. In our research, we focus on multi-threaded parallel computing on a single node. Several experiments have been tested on the Steele. Monte-Carlo method of computing π includes random number generator to uniformly pick one point from a unit square. The linking libraries, the compilers and the way of writing the code affect whether we can make the program run in parallel effectively. False sharing might make the program doesn't get the speedup. Concurrency affects whether we can get the correct answer. In the real-world application, libsvm is used for testing how the speedup it can reach with different number of threads and different compilers.

1 Overview of parallel computing

Parallel computing has been developed for a long time. In the past, we only can see parallel computing in the scientific world and business servers. Nowadays, the technique is widely used everywhere in the personal computer. From 1980 to 2004, the increasing frequency of the clock rate in the computer is the main method to increase the computation efficiency. When we pack more circuits into the small chip and increase the clock rate, the power consumption becomes a big issue. The physical characteristic in the IC, such as the capacitance of capacitors becomes more nonlinear. It's not easy to overcome the physical problems, so the industry changes the road to parallel computing.

In order to achieve the parallel computing [1], there are several parts need to be considered. From the top part to the lower part, applications, programming models, compilers, middleware to hardware. Some applications are easy to be parallelized, and some are not. For those parallelizable applications, which models should we use? Should we use shared memory or message passing among processes? How the divided jobs are distributed to different processors? There are so many questions to be asked.

1.1 Shared and Distributed memory architecture

There are two main differences in parallel computing. One is shared memory architecture, and the other one is distributed memory architecture. In the consumer markets, we see a lot about dual-core and quad-core processors. It even becomes hard to buy a new single-core machine in 2009. For a quad-core processor shown in Fig. 1, each core has its own L2 cache. This design can increase the cache usage within a certain scope. A lot of researchers believe that we will be able to pack more cores into one chip, so in the future we might be able to see 100 cores in a single processor. The best way to distribute smaller tasks to each core is using threads. In C++, there are no generic threads, so people use boost library or pthread. In java, the thread is built-in.

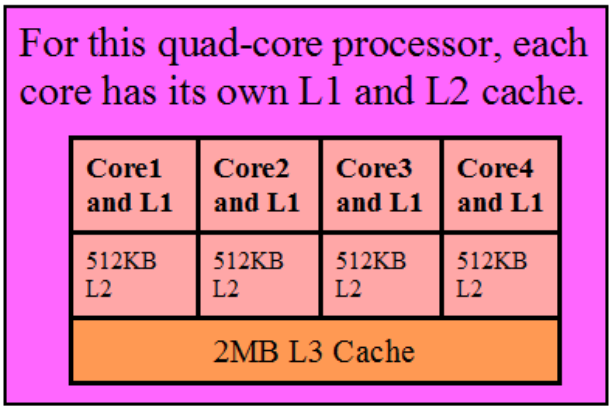

Figure 1 AMD Opteron processor.

For the distributed memory architecture, it is usually done with several computers connected with each other via high-speed Ethernet card. Each computer has its own memory. On top of the operating system in each computer, there is a middleware which can control how and where to distribute the data. People also call it message passing. The famous example is condor [2]. During the compilation, the compiler will arrange how many nodes to be used and where to send and receive the data and programs. Usually, there is also a scheduling mechanism involved there. Several groups of people can run a lot of different programs on this system. Sometimes people also call the distributed computing grid computing.

1.2 Some parallel paradigms

The simplest one is embarrassingly parallel which is shown in Fig. 2. There is no or less communication between threads. This problem can be easily run in parallel.

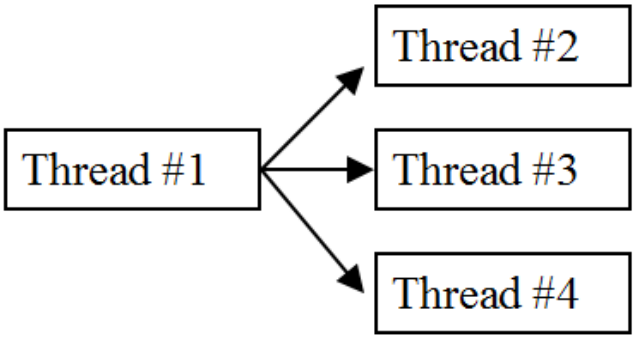

Figure 2 Embarrassingly parallel.

Message passing between threads can also be used. Some threads might need to wait the temporary results from other threads before it can start. It is also possible that two threads try to access or write the same data at the same time. This requires programmers spend more time on which part can be parallelized and which part cannot be parallelized. The execution sequence and concurrency also need to be considered.

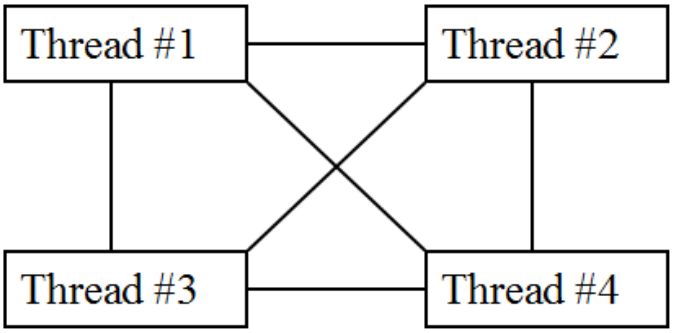

Figure 3. Message passing between threads.

No matter which paradigms we used, there is always fork and join overhead. The speed-up is not simply two times faster when we use two times more threads. Some parts can be parallelized easily like the embarrassingly parallel problem, while other parts cannot be parallelized. It's the programmer's responsibility to correctly transfer from the serial codes to parallel codes without errors. For more complicated or non-trivial problems, it requires programmers' efforts to do it right. So far, there is no such a powerful parallel compiler to parallelize codes automatically.

1.3 Software on the cluster

There are several tool-kits which can help us maintain and provision the status of the nodes on the cluster system. Some famous ones are XCAT, OSCAR, ROCKS and PERCEUS. Usually, this middleware help us configure the cluster correctly. For example, one node is restarted due to some reasons. After it is rebooted, the system should automatically add this node into the original system. When the node is offline, the system should not distribute some jobs to

the disconnected node. On a cluster system, when a program divides the data into several parts. Each smaller part is used to run on a single node. There might be copy process to move the data to the desired node. The network file system can help us do this process smoothly.

When we need to run a program on a number of nodes, usually we use message passing interface (MPI). There are two major implementations, OpenMPI and MPICH2. There are so many libraries and routings help the users run program with MPI. When we only need to run a program in parallel on a single node, we can use OpenMP or pthreads. Sometimes, we also call it multi-threaded programming. The performance really depends on the libraries and the compilers. Java also provided thread capability.

2 Overview of the Steele cluster

The Steele cluster was installed in May, 2008[3]. Steele consists of 893 8-core processor, which is dual 2.33 GHz Quad-core Intel E5410. Each node has 16 or 32 GB memory. The nodes are connected together via Gigabit Ethernet and Infiniband. Therefore, this system is a good experiment platform for both distributed and shared-memory parallel programs. The operating system on it is Red Hat Enterprise Linux x86_64 3.4.6-10.

The Steele (60 TFlops) is part of the TeraGrid system. Several universities connect their own clusters together to become a more powerful cluster. The user in one university is allowed to run the program on another university's cluster after the proper paper application form. The two biggest cluster systems are Kraken (608 TFlops) at the University of Tennessee and Ranger (579.4 TFlops) at the University of Texas at Austin.

The queues that we are allowed to run are tg_work, standby and standby-8. Since there are so many users running parallel programs on the Steele and the free resources are also limited, we focus our research on the multi-threaded parallel computing on a single node. Each node has 8-core processor and 16-32 GB memory and this is very suitable for our research.

On the Steele, there are three C++ compilers that we can use, Intel compiler (icc verion 10.1), GNU compiler (g++ version 4.3.0) and PGI compiler (pgCC version 7.1-6). Boost C++ library installed on the Steele is version 1.36. The java compiler is version 1.6.0_05, but the runtime java is version 1.4.2. When we ran the java program, we need to soft add the runtime library.

3 Connection to the Steele

The easiest way to connect to the Steele is go to the TeraGrid (http://www.teragrid.org/). On the top menu, click "My TeraGrid" and then SSH terminal. This terminal is embedded in the browser and font is slightly smaller. The other way to launch the terminal is by downloading the program from GSI-SSHTerm for NCSA (http://grid.ncsa.uiuc.edu/gsi-sshterm/). Besides these

two ways of connecting to the Steele, there are other ways which are shown on the TeraGrid website. The host name of the Steele is:

tg-login.purdue.teragrid.org

After we log in the system, we can require a node with 8-core processor. By doing so can make sure nobody else uses this node.

qsub -q standby -I -l select=1:ncpus=8 -A TG-ASC090049

4 Time measurement

There are several ways to measure the execution time. We can use the command "time" in Linux by typing in "time program". However, this shows the user time, system time, total time and CPU utilization percentage. When CPU utilization is more than 100%, it means more than one CPU is running and it also means there is parallel computing. This measurement couldn't truly reflect the details of time spent in a subroutine in the program.

We can use "ctime" to measure the time spent in a subroutine.

```
#include <ctime>
clock_t time_begin=clock();
//……..
//Computations to be measured
//……
clock_t time_end=clock();
double diffticks=time_end-time_begin;
double elapsed_time=(diffticks)/(CLOCKS_PER_SEC);
```

Figure 4. Time measurement through ctime library

The above method can be run in both Windows and Linux, but the precision is only millisecond. On the Steele machine, we can use native Linux time measurement.

```
#include <sys/param.h>
#include <sys/times.h>
#include <sys/types.h>
 struct tms tt,uu;
 long time_begin,time_end;
 time_begin = times(&tt);
//……..
//Computations to be measured
//……
time_end = times(&uu);
float elapsed_time= (float)(time_end-time_begin)/HZ );
```

Figure 5. Native Linux time measurement

The above method only works for program running in Linux. The precision is only millisecond. OpenMP program can use the built-in get wall time as long as the complier supports OpenMP. OpenMP time measurement also gives us higher precision.

```
#include <omp.h>
double time_start, time_end; /* for timing */
time_start = omp_get_wtime( );
//……..
//Computations to be measured
//……
time_end = omp_get_wtime( );
double elapsed_time=  time_end-time_start;
```

Figure 6. OpenMP time measurement

5 Monte-Carlo method to compute π

The traditional way to compute π is done by uniformly randomly choosing a point within a square region (Figure 7). Suppose the length of the square is 1, the area of the quarter circle is $\pi*1^2/4 = \pi/4$. The probability of the point within the square region is equal to

$$\frac{\pi/4}{1\times 1} = \frac{\pi}{4} \tag{1}$$

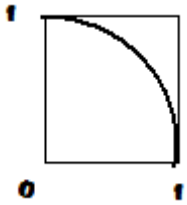

Figure 7. The quarter circle is in the square region.

Suppose the maximum number of points we want to generate is *N*. The number of data points that fall in the quarter circle is C where $x^2 + y^2 \le 1$. Then $\frac{C}{N} = \frac{\pi}{4}$. The estimated π is $4\times\frac{C}{N}$.

Since the number of generated random points can be chosen in parallel, this problem can be solved with parallel computing. However, there are more details and more issues to implement this idea.

```
#include <omp.h>
#include <cstdlib>
#include <stdio.h>
long sum=0;
#pragma omp parallel for
//#pragma omp parallel for reduction (+:sum) //This is the correct one.
for(int i=0;i<N;i++){
        double x=(double)rand()/RAND_MAX;
        double y=(double)rand()/RAND_MAX;
```

```
        if((x*x+y*y)<1){
                sum+=1;
        }
}
double comp_pi=4.0*sum/N;
```

Figure 8. The wrong version of computing π.

In Figure 8, this code will give us a wrong answer. After reading the variable, if the random generated point is within the quarter circle, the variable is increased by 1. However, if two threads try to increment the variable, each thread might read and increment the variable *sum* at the same time. The concurrency issue can be eliminated by including "reduction (+:sum)" clause.

After making sure we get the right estimated π, let's see whether the number of threads we use can reduce the computation time. The result is shown in Table 1.

Table 1 The results of OpenMP and rand() function using cstdlib library which are compiled by Intel compiler.

#threads	1	2	4	8
Time(sec.)	1.08	8.02	15.3	5.58
π	3.14113	3.141614	3.141554	3.141551
Error	-0.000463	0.000021	-0.000038	-0.000042

Three different compilers were tested, but none of them give us good indicators that using more threads should reduce the computation time. This problem can be solved by using our own pseudo random number generator.

5.1 Pseudo Random Number Generator

Pseudo Random Number Generator (PRNG) can be computed by using the following equation:

$$x_{i+1} = (x_i \times a + b) \bmod (Max) \tag{2}$$

where x_i is the i^{th} generated random number, a and b are arbitrary numbers, *Max* is the maximum integer of these sequences of random numbers. In my design, I set *a*=137, *b*=24681 and *Max*= 1478471. The results are shown in Table 2. In later research, we found that parameters *a* and *b* affect the error precision. We should use larger *a* and *b* to reduce the error.

Table 2 The results of OpenMP and rand() function using cstdlib library which are compiled by Intel compiler.

#threads	1	2	4	8
Time(sec.)	0.29	0.15	0.08	0.04
Pi	3.145312	3.145374	3.145293	3.14528
Error	0.003719	0.003782	0.0037	0.003687

```
#include <omp.h>
double x,y;
#pragma omp parallel for private(x,y) reduction (+:sum)
for(int i=0;i<N;i++){
        x=(double)prng_rand()/PRNG_RAND_MAX;
        y=(double)prng_rand()/PRNG_RAND_MAX;
        if((x*x+y*y)<1){
                sum+=1;
        }
}
double comp_pi=4.0*sum/N;
```

Figure 9. The computation of π by using pseudo random number generator.

5.2 Partition a large task with multiple threads

In OpenMP, we don't need to consider how the parallel for loop is done. Actually, there are three ways we can specify how the for-loop is scheduled, which are static, dynamic and guided. In multithreaded programming, the threads are executed at the same time. Suppose there are two threads and each thread is assigned with a similar workload. Thread 1 is executed before thread 2, but it doesn't mean thread 1 finished earlier than thread 2. Even if in the previous run, thread 1 is executed earlier than thread 2, it doesn't guarantee thread 1 will be executed earlier than thread 2 in the following run. We also cannot guarantee thread 1 is always executed on the core 1. It really depends on how the Operating System schedules the threads and processes.

In pthread or boost::thread, we need to specify how the for-loop is executed in each thread. Suppose there are 4 threads and the for-loop executes from i=1 to i=1000, we can equally distribute the 1000 sub-tasks into 4 tasks. Each thread runs 250 sub-tasks. It's not a good idea to let each thread run only one sub-task, because it creates too much overhead. We should eliminate the fork-and-join overhead as much as possible.

After we specify 4 sub-tasks, each thread might try to update a global variable. If each thread updates the global variable frequently, we should generate a local variable and only update this local variable. After this sub-task is done, we update the global variable once. This can avoid the issue of lock-unlock the global variable too often.

6 Issues of parallelization

In previous section, we have discussed several issues when we implement parallel computing. Some libraries or functions cannot be easily parallelized. Those problems can be overcome by writing our own functions. In this section, we'd like to discuss more issues in parallel programming.

6.1 Linking libraries

From Table 1, we know that some libraries or functions cannot generically be run in parallel. There are other programming ways to run threads, such as pthread and boost::thread. These two ways require the programmer explicitly assign the sub-tasks to each thread. Pthread is especially very low level.

Table 3 The results of pthread with prng_rand() function which are compiled by Intel compiler.

#threads	1	2	4	8
Time(sec.)	0.23	0.12	0.06	0.04
Pi	3.145312	3.145374	3.145293	3.141528
Error	0.003719	0.003782	0.0037	0.003687

From Table 3, we can see that the execution time is slightly better than OpenMP which is shown in Table 2. However, it's more tedious to program with pthread. We need to create mutex to make sure the concurrency. The way to pass the starting position and ending position of the for-loop is done by constructing a struct. Each thread has its own memory of parameters so that it guarantees each thread reads in the information correctly. Every time the thread tries to update the global variable; it needs to check the mutex_variable is unlocked.

It's slightly easy to program with boost::thread, we can pass the parameters (the starting position and ending position of the for-loop) to each thread directly. Because each thread tries to update the global variable, we still need to use boost::mutex. The computation time (Table 4) is much longer than OpenMP and pthread. We'd like to also point out that we need to make a makefile to compile boost::thread programs otherwise it cannot run.

Table 4 The results of boost::thread and prng_rand() function which are compiled by Intel compiler.

#threads	1	2	4	8
Time(sec.)	0.46	0.23	0.12	0.06
Pi	3.139531	3.142128	3.138194	3.138164
Error	-0.002061	0.000536	-0.003399	-0.003428

Java is a more modern programming language. Thread is already built-in. We can create an object which extends from thread. Each thread has its own starting position and ending position of the for-loop. Each thread also records its local variable. After each thread has finished its task, the global variable is updated by reading each local variable within the thread. Here, the java.util.Rand function is used to generate the random number. We don't need to write our own pseudo random number generator. The results of computation time are shown in Table 5.

Table 5 The results of Java thread.

#threads	1	2	4	8
Time(sec.)	1.331	0.693	0.347	0.231
Pi	3.1414072	3.1421248	3.14181	3.1425064
Error	-0.000185	0.000532	0.000217	0.000914

Apparently, java is the slowest program. On the Steele machine, we are not sure why it needs extra effort to "soft add +java-1.4.2", otherwise the program cannot be compiled smoothly.

6.2 False sharing

The computer usually reads in a block of memory into the cache which is later used by a core. This can improve the data-retrieving speed. This design is due to the reference of locality. From Figure 1, we can see that each core has its own L1 and L2 cache. If we don't do the coding

wisely, we might have false sharing problem. For example, we create a consecutive array. Thread 1 tries to update the first element of the array while thread 2 tries to update the second element of the array. From the hardware point of view, it finds that the block of memory is duplicated into two caches. Cache 1 updates one part of the block while cache 2 updates the other part of the block. In order to make the block of memory be consistent between 2 caches, the hardware needs to take extra effort to copy the updated value to the other cache. The code is shown in Figure 10 and the computation time is shown in Table 6.

```
double x,y;
#pragma omp parallel
{
int id=omp_get_thread_num();
sum_part[id]=0;
        #pragma omp for private(x,y)
        for(int i=0;i<N;i++){
            x=(double)prng_rand()/PRNG_RAND_MAX;
            y=(double)prng_rand()/PRNG_RAND_MAX;
            if((x*x+y*y)<1){
                sum_part[id]+=1; //The consecutive memory is updated by each thread.
            }
        }
}
for (int i=0;i<num_threads_to_use;i++){
        sum+=sum_part[i];
}
double comp_pi=4.0*sum/N;
```

Figure 10. The code of generating false sharing problem.

Table 6 The computation time when false sharing occurs.

#threads	1	2	4	8
Time(sec.)	0.29	0.23	0.22	0.43
Pi	3.145312	3.145374	3.145293	3.14528
Error	0.003719	0.003782	0.0037	0.003686

6.3 Concurrency and mutex

When two threads try to update the same variable, we need to make sure the second thread can only read the variable before the first thread has finished its reading and updating. The following example is an example telling us why this is an important issue.

Suppose the original balance in your banking account is $1,000. Now you try to deposit two $200(total $400) into your banking account. Thread 1 read the balance $1,000 and at the same time thread 2 tries to read and update the balance before thread 1 finishes its job. Thread 2 successfully adds $200 in the balance, but thread 1 doesn't know about it. Thread 1 still thinks the beginning balance is $1,000. After thread 1 finishes its job, it writes $1,200 back to the balance. Thread 2 also writes $1,200 back to the balance. There is $200 missing in this process. Only the last finishing thread updates the balance.

Thread 1	Thread 2	Balance
Read balance: $1,000		$1,000
	Read balance: $1,000	$1,000
	Deposit $200	$1,000
Deposit $200		$1,000
Update balance $1,000+$200		$1,200
	Update balance $1,000+$200	$1,200

Figure 11. The concurrency problem when two threads try to update the same variable.

In order to overcome this issue, we should create mutual exclusion (mutex). When thread 1 accesses the balance variable, it locks a mutex. When thread 2 tries to read the balance, it finds the mutex is locked. Thread 2 needs to wait until thread 1 finishes its job and releases the mutex. What if there are two threads that try to grab each other's mutex? Then the dead-lock occurs. The programmer should consider thoroughly preventing creating concurrency and dead-lock issues.

6.4 Compilers issues

On the Steele machine, there are three compilers we can use, Intel, GCC and PGI. After we have finished programming the right code, the compiler still can play an important role to affect the execution time. Even worse, the compiler might not make the program run smoothly in parallel. In Table 7, GNU and PGI compiler don't get the benefits when more threads are utilized. The way of constructing PRNG is described in Figure 12.

Table 7 The computation time among different compilers.

#threads	1	2	4	8
Intel Compiler(icc)	0.29	0.15	0.08	0.05
GNU Compiler(g++)	0.8	0.4	0.2	0.16
PGI Compiler(pgCC)	**1.03**	**3.17**	**8.91**	**15.38**

```
long prng_current=12345;
#pragma omp threadprivate(prng_current)
void prng_seed(long num){
        prng_current=num;
}
long prng_rand(){
        long cal_num= (prng_current*137+24681)%PRNG_RAND_MAX;
        prng_current=cal_num;
        return cal_num;
}
```

Figure 12. The proper way to construct PRNG. When we make the comment on threadprivate (prng_current), the computation time of the code compiled by PGI is much less. However, the execution time is still not reduced consistently when more threads are used.

Different ways of writing codes also give us different execution time among different compilers. A struct is constructed to wrap the PRNG function, and we call it method 2. Each thread has its

own PRNG. From Table 8, the execution time among different number of threads and different compilers is stranger. This method 2 of building PRNG is shown in Figure 13.

Table 8 The computation time among different compilers with method 2 of constructing PRNG.

#threads	1	2	4	8
Intel Compiler(icc)	**0.33**	**0.38**	**0.23**	**0.61**
GNU Compiler(g++)	**0.84**	**1.32**	**1.45**	**1.95**
PGI Compiler(pgCC)	**0.32**	**0.37**	**0.25**	**0.39**

```
struct PRNG{
        long prng_current;
        PRNG(){
                prng_current=12345;
        }
        PRNG(long input_num){
                prng_current=input_num;
        }
        long prng_rand(){
                prng_current= (prng_current*137+24681)%PRNG_RAND_MAX;
                return prng_current;
        }
};
```

Figure 13. Constructing PRNG with method 2.

7 Support Vector Machine using OpenMP

Support Vector Machine (SVM) was first developed by Vapnik [4]. It is a powerful tool in machine learning. There are several groups implementing his idea, such as svmlight and Libsvm. From their websites, we can find that libsvm[5] is still very active and updated several times a year. They also provide source codes written in C++, Java and Python. We downloaded the source code of C++ version from the website [6]. We refer to the author's suggestion making two for-loop run in parallel. The code that we modified in svm.cpp is shown in Figure 14.

In Qfloat *get_Q(int i, int len) const, we change it to

```
        #pragma omp parallel for private(j)
        for(j=start;j<len;j++)
                data[j] = (Qfloat)(y[i]*y[j]*(this->*kernel_function)(i,j));
```

In void svm_predict_values(const svm_model *model, const svm_node *x, double* dec_values),

```
        #pragma omp parallel for private(i)
        for(i=0;i<l;i++)
                kvalue[i] = Kernel::k_function(x,model->SV[i],model->param);
```

Figure 14. Two for-loop are changed to run in parallel.

After modifying the code to make it run in parallel, we used real-world dataset to test if there is speedup when program is run in parallel. Ringnorm is binary class problem and it can be downloaded from Delve's website (http://www.cs.toronto.edu/~delve/data/). We used this dataset because it takes much longer time to do training and testing for classification. Three different compilers are run and the results are shown in Table 9, 10 and 11. When we ran libsvm-predict which is compiled by Intel compiler, we encountered a problem. It shows this, *** glibc detected *** corrupted double-linked list: 0x0000000000531400 *** Abort.

Table 9 Rinnorm dataset is used. The code is compiled by Intel compiler (icc 10.1). The computation time is measured with seconds.

#threads	1	2	4	8
Libsvm-train	0.77	0.53	0.41	0.37
Libsvm-predict	0.59	0.39	0.28	0.25

Table 10 Rinnorm dataset is used. The code is compiled by GNU compiler (g++ 4.3.0). The computation time is measured with seconds.

#threads	1	2	4	8
Libsvm-train	1.97	1.31	**0.97**	**0.96**
Libsvm-predict	1.66	1.29	1.04	0.97

Table 11 Rinnorm dataset is used. The code is compiled by PGI compiler (pgCC 7.1-6). The computation time is measured with seconds.

#threads	1	2	4	8
Libsvm-train	**1.04**	**1.06**	**1.2**	**1.47**
Libsvm-predict	0.66	0.45	0.31	0.25

8 Summary

In our research, we found that it's easier to write the program with OpenMP. We simply put "#pragma" in front of the for-loop, and then just let the compilers do the rest of things. However, different compilers treat the code slightly differently. Among Intel, GNU and PGI compilers, PGI is slightly worse than the other two. The linking libraries also affect whether the program can be run in parallel. In our research, we found that the random number generator which uses "cstdlib" library couldn't be parallelized.

There are also other ways of writing multithreaded programming, such as pthread, boost::thread or Java thread. There is more tedious work to consider how the thread is initialized and how the work is distributed to each thread. The good thing is that all the three compilers can get the speedup when more threads are used.

There should be more work on the compiler company to standardize which syntax represents which hardware utilization model. Sometimes, the fork-join overhead is larger than the benefit and it results in using more threads leads to more computation time.

From the programmer's view, we should consider some problems occurring only in parallel computing, such as false sharing and concurrency. It's slightly difficult to debug in the parallel code. When the program becomes more complicated and more realistic, we need to make sure the serial part and parallel part work perfectly together. The speedup is not simply doubled when two times threads are used.

LIST OF REFERENCES

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick, "The landscape of parallel computing research: A view from Berkeley, " Technical Report No. UCB/EECS-2006-183, December 18, 2006.

[2] http://www.cs.wisc.edu/condor/

[3] http://www.rcac.purdue.edu/userinfo/resources/steele/userguide.cfm

[4] Vapnik, V. (1995) The Nature of Statistical Learning Theory. New York, NY: Springer-Verlag.

[5] R.-E. Fan, P.-H. Chen, and C.-J. Lin. Working set selection using the second order information for training SVM. Journal of Machine Learning Research 6, 1889-1918, 2005.

[6] http://www.csie.ntu.edu.tw/~cjlin/libsvm/