Preliminary Contest Proposal

Ocean University of China

March 18, 2015

1 Background Description

1.1 History of Hardware Platform

Key Laboratory of Physical Oceanography MOE China High Performance Computing Platform facilities are mainly composed of two parts.

One part, purchased in 2006 at a cost of 10 million Yuan, is the High-End Computing Platforms SGI altix3700, of which the main technical indicators are: 224 Itanium2 processors, 224G memory and 5T high-speed hard drive providing 1.2 trillion calculations per second.

The second part is the High-Performance Computing Clusters. After two main stages of construction, the Computing Clusters has 3132 consolidated core CPUs in al total. Theoretically, the peak computing capacity is about 33.32 trillion times per second, and in the real measurement, It can provide 29 trillion calculations per second. The efficiency reached 89.8%.

In terms of storage, the Computing Clusters is equipped with 144T general storage, 300T bare capacity of near-line storage and 100T bare capacity of online storage and 1 capacity. Applying a high-speed parallel file storage system, the Computing Clusters is able to reach a reading and writing online storage of bandwidth 2.8Gb/s, near-line storage reach 2.1Gb/s.

The High-Performance Computing Clusters ranks 85 among the "TOP 100 High-Performance Computers of 2011 China List" at the China's High-Performance Computing Annual Meeting in October 2011.

1.2 Related Research

Last few years, we have been developing the numerical ocean models. Take GOCTM (Global Ocean Circulation and Tide Model) for example, comparing to other global models, GOCTM adopts Global FVM (Finite Volume Method) Variable Triangular Mesh, which could solve the problems of unknown physical process on the boundary and difficulties of polar forecast. This model takes Sun-Month-Earth gravitation into consideration, enabling itself to forecast the ocean of climatology investigation based on the gravitation. A new water-sediment joint density function ($\rho = \rho_w + (1 - \frac{\rho_w}{\rho_s}) * C$) has been built, according to this, the coupling problems of dynamic in the transport of water and sediment could get resolved.

1.3 Key achievements

Developed a high frequency fast real-time observation data assimilation system and constructed Bohai sea and Yellow Sea ocean circulation forecasting system. It contains the following main parts:

We invented the high frequency fast real-time observation data assimilation methods, and flow field prediction RMSE is reduced by about 40 percent, which has broken the bottleneck of high frequency observations of real-time business applications, and it authorized patents.

We developed the circular three-dimensional variation (Cycling-3dvar) assimilation techniques and model systems, which can make a 6-hour "hot start" prediction of the initial field while the meteorological forecast RMS error can get about 20% lower.

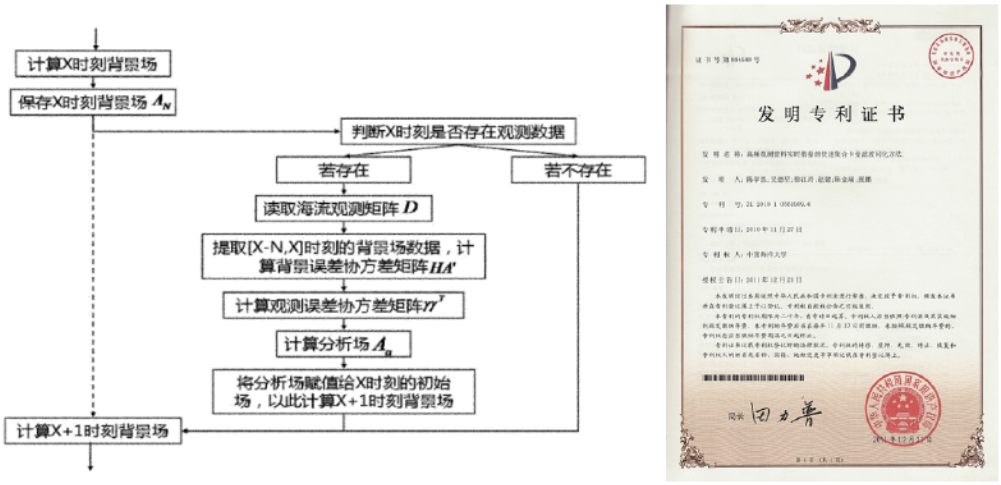

Figure 1: (a) The Ensemble Kalman Filter assimilation flow chart (b) Patent of high frequency fast real-time observation data assimilation methods

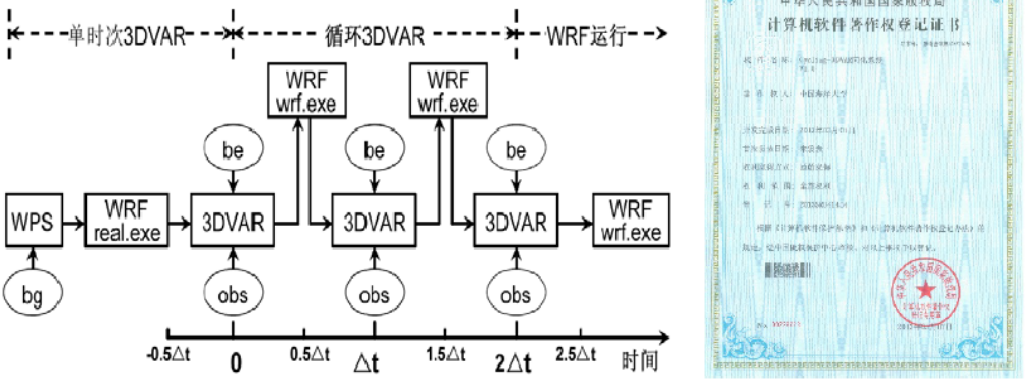

Figure 2: (a) WRF Cycling 3DVAR assimilation flow chart (b) Patent of Cycling-3DVAR assimilation system

2 Team Description

On one afternoon a year ago, when Professor Shen Biao from College of Physical and Environmental and professor Zhang Linjie from school of mathematical science talking about the developing of large-scale scientific computing and science, they both hold the opinions that the combination of these two subjects is of great importance for the advance of technology and future. Besides, for big scientific calculations, reducing the time from months to days or days to hours becomes quite worthwhile. However only few people in our university realize the significance of it, they proposed to the university to build the interest group focused on developing students who have great interest on computer and sciences, and named OUCSC Club.

As soon as OUCSC Club was build up, it immediately attracting students from college of physical and environmental, school of mathematical science, college of information science and engineering and so on. They have great enthusiasm for what they have participated. Meanwhile, the OUCSC Club carried on serious learning activities such as learning using FORTRAN and C++ program language for more practical use, the introduction of Computer composition. Besides, the teacher showed us how the cluster works.

Club members were working together, struggling together, learning from each other. When we heard the news that ASC was about to come, everyone wants to have a try. After a series of assessment and screening, we finally established our team — Season: The son of sea.

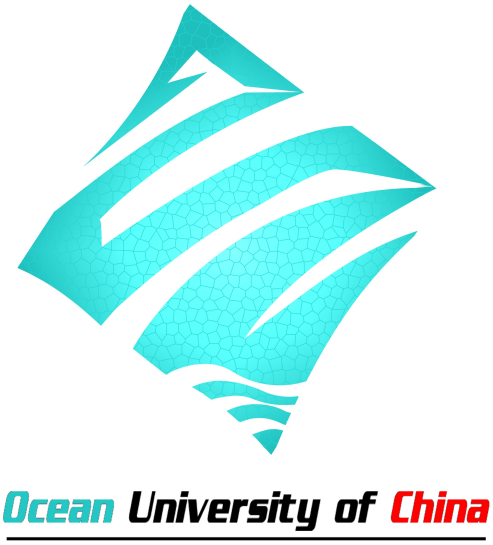

Figure 3: Team Logo

3 System Configuration

3.1 Hardware Configuration

As we do not have the specific hardware described in *Preliminary Contest Notifications*, we decide to use our own platform instead. The hardware configuration of each node is shown in Table 1.

TDP, namely thermal design power, is power consumption of full loaded hardware. Con-

Item Name	Configuration	TDP
CPU	Dual-Socket Six-Core Intel Xeon X5650 @2.66Ghz CPUs	190W
Memory	4GB x 6, DDR3 1333Mhz Memory	60W
Hard Drive	160GB 7.2 RPM SATA 2.5'	5W
MotherBoard	SuperMicro X8DTT-IBQF	10W
Interconnection	40Gb/s QDR InfiniBand 10Gb/s Ethernet	30W
Operating System	Red Hat Enterprise Linux Server release 5.4	None

Table 1: Hardware Configuration

sidering the cluster almost runs at full load when doing computation, therefore the use of TDP as a standard to design a cluster is reasonable. Specific thermal design power of cluster hardware are listed in Table 1, and according to the estimation, the overall power consumption of our cluster is 2980W. By measure of an external power sensor, a single node power consumption under full workload is 280W, so the power consumption of a 10-node cluster is obviously less than 3000W, completely meet with the requirements of the competition.

According to the above power estimation and actual measurement using power measurement outlet, we decide to use 10 nodes to set up our cluster. Here is a schematic diagram in Figure 4.

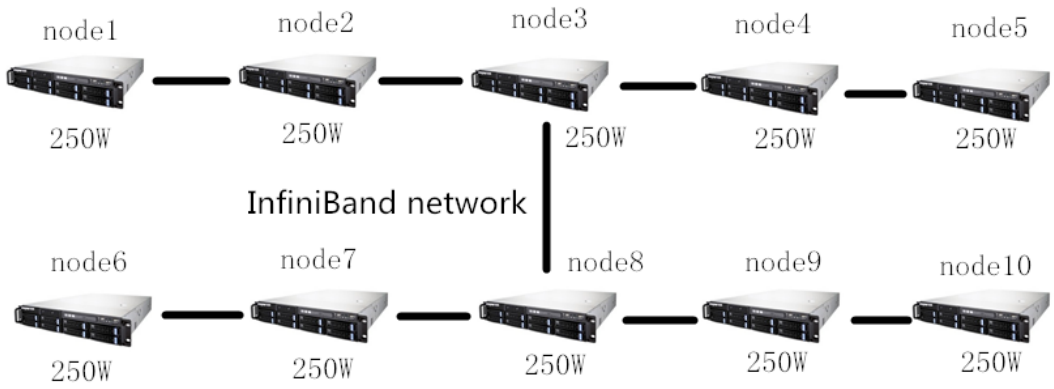

Figure 4: Cluster Configuration

3.2 Software Configuration

It is generally believed that in most cases, running applications compiled by Intel compilers on Intel CPUs could achieve best performance. So we will mainly use Intel compilers, MPI implementation and performance analyse tools to do the following benchmarking and optimizing. In addition, we will use another two different MPI implementations: MPICH and OpenMPI to compare the runtime performance of NAMD application.

Item Name	Version
Compilers	GCC version Red Hat 4.1.2-46 Intel Composer XE 2015 v15.0.0
MPI Library	Intel MPI 5.0.1.035 MPICH version 3.1.4 OpenMPI version 1.8.4
Performance Analyzer	Intel VTune Amplifier 2015.1.0.367959 Intel Trace Analyzer and Collector 9.0.1.033

Table 2: Software Configuration

4 HPCC Test

5 Running, Profiling and Optimization of NAMD

5.1 Overview

NAMD is a parallel molecular dynamics program for multi-platform designed for high-performance simulations of large biomolecular systems in structural biology. It is developed by the joint collaboration of the Theoretical and Computational Biophysics Group (TCB) and the Parallel Programming Laboratory (PPL) at the University of Illinois at Urbana-Champaign.

This section of our proposal was done to provide:

- Guides of running and compiling of NAMD
- NAMD performance benchmarking and profiling
- Performance comparison with different MPI implementations
- Understanding NAMD communication pattern
- Productivity optimization

5.2 Performance Measurement

As described in the document, the easiest and most accurate way to do this is to look at the "Benchmark time:" lines that are printed after 20 and 25 cycles (usually less than 500 steps). NAMD uses a value called "ns/day" to measure the performance. From the literal meaning, the higher this value, the better performance.

5.3 Running pre-compiled version

NAMD runs on a variety of serial and parallel platforms. While it is trivial to launch a serial program, a parallel program depends on a platform-specific library such as MPI to launch copies of itself on other nodes and to provide access to a high performance network such as Myrinet or InfiniBand if one is available.

As official pre-built binaries are provided, we could first use one node as a single workstation, and run these pre-built version to analyze the runtime characteristics of this application.

5.3.1 Serial version

We use this command to lauch serial application:

namd2 <configfile>

As for the apoa1 workload, it takes about 11 minutes (646.89s) to finish this job on a single node described in 1, with a very low benchmark score: 0.067 ns/day.

First we take a look at the CPU usage when running apoa1 workload on a single core, shown in Figure 5.

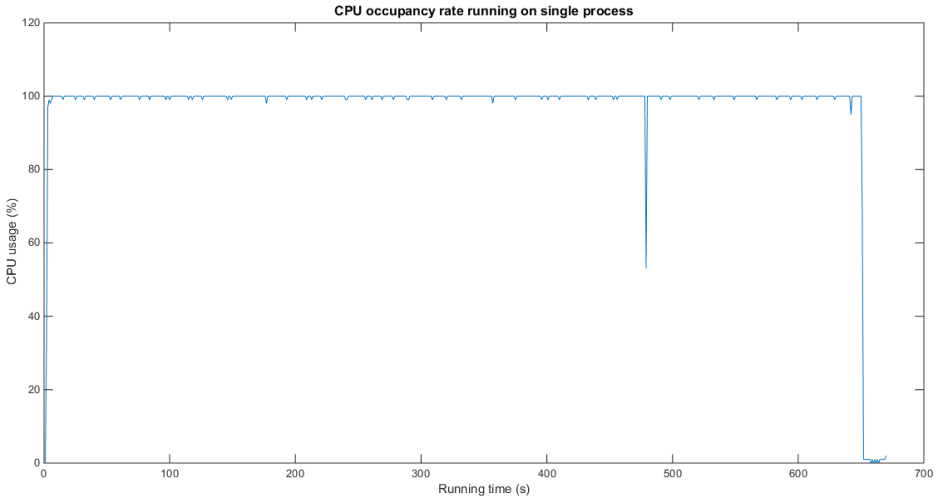

Figure 5: CPU occupancy rate running on single process

The figure above shows that NAMD is a compute-intensive application.

Then we collect read and write information of hard drive as Figure 6.

We can infer from this graph that, although there are many disk I/O operations in the entire calculation process, each time the amount of data is small, with an average of about 56 KB/s. Actually, the final size of output data depends on num of atoms, simulation step, enr, xvg, tpr and so on, and for our apoa1 workload, the output file is also small. All in all, for this NAMD application, there isn't much pressure on disk IO.

5.3.2 Multiprocessor version

We use this command to run NAMD on a single node in parallel:

namd2 +p<procs> <configfile>

Or lauch with the help of *charmrun*:

charmrun namd2 ++local +p<procs> <configfile>

On our single node, it takes about ??? s to finish the apoa1 workload, and ??? s to finish the f1atpase workload.

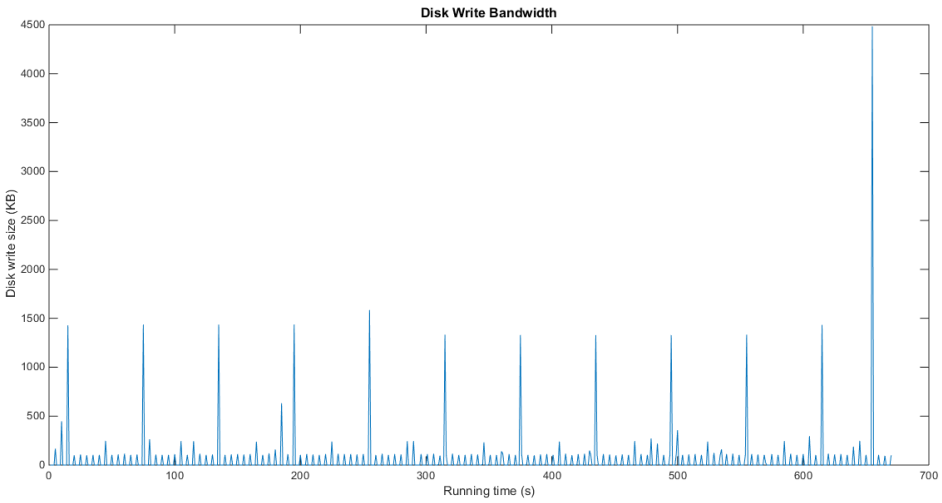

Figure 6: Disk Write Bandwidth

5.3.3 Linux clusters with InfiniBand

It is a little more complex to run NAMD on multiple nodes:

charmrun namd2 +p<procs> ++nodelist <hostfile> <configfile>

We should specify a "nodelist" here, which includes a list of hostnames of each node. As NAMD application is developed base on "Charm++" framework, the *charmrun* command could be regarded as a Multi-Process Scheduler. So of course we could build our own MPI version and use **mpiexec.hydra** command to manage processes instead, this brought us better flexibility to tune some runtime parameters to achieve better performance.

5.3.4 SMP builds with InfiniBand

According to the document description, SMP builds generally do not scale as well across nodes as single-threaded non-SMP builds because the communication thread is both a bottleneck and occupies a core that could otherwise be used for computation. So these builds would not be considered.

5.4 Compiling NAMD

5.5 MPI Optimization

6 Optimization of the Gridding Program

6.1 Application Analyse

6.1.1 Introduction

Function of the provided program is making use of convolution to compute new **4096*4096** data points with 3,200,000 discrete samples located in u, v space.

The related files introduction is given in Table 3, the UML figure is given in Figure 7, and the flow chart is given in Figure 8. Note that more than 99 percent computation time is spent on the "Gridding", so it is most important to speed up this part. We will have in-depth analysis to this in the next section.

File name	Instruction
input.dat	Input file, including some important parameters for this program
input.dat	Input file, including some important parameters for this program
randnum.dat	Input file, including 3,200,000 discrete samples located in u ,v space
grid.dat	Output file, including the computed results
log.dat	Output file, including the Gridding time and the Gridding rate
Benchmark.h Benchmark.cc	Kernel of Gridding, includes initialize, Gridding and print function
Stopwatch.h Stopwatch.cc	A simple class for timing
tConvolveMPI.h tConvolveMPI.cc	Entrance of this program

Table 3: Introduction of related files

As the official provided test platform is not convenient to use (takes a long queue), the following tests and analyse are carried out on our local machine, and its configuration is shown in Table 5. **But the final test and result would be based on the official platform.**

Item Name	**Configuration**
CPU	Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz x 2, 8 cores
Memory	8GB x 12, DDR3 1333Mhz Memory
Hard Drive	1TB 7.2 RPM SATA 2.5
Accelerator card	Intel Xeon Phi Coprocessor 3120P (6GB, 1.1 GHz, 57 cores) x 2
Operating System	Red Hat Enterprise Linux Server release 6.4

Table 4: Configuration of local test machine

6.1.2 Hotspot Analyse

6.2 Optimization on CPU platform

6.2.1 MPI Parallelization

We consider the optimization on CPU platform with MPI and OpenMP technique. First, we use MPI technique only, then we make use of MPI and OpenMP together.

The MPI parallelization of this program is based on the classical Map-Reduce model. All the samples in the *samples* vector are regarded as "tasks", and when the MPI processes are lauched by mpirun, we "map" these tasks equally to different processes, let each process calculate its

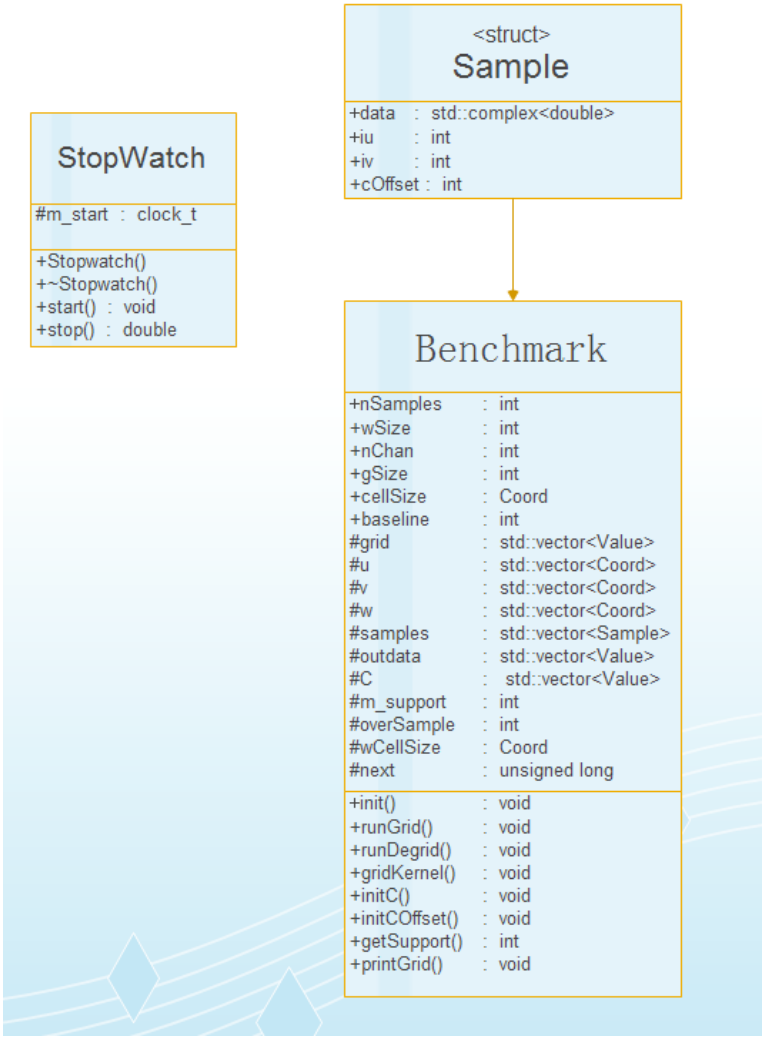

Figure 7: UML of provided program

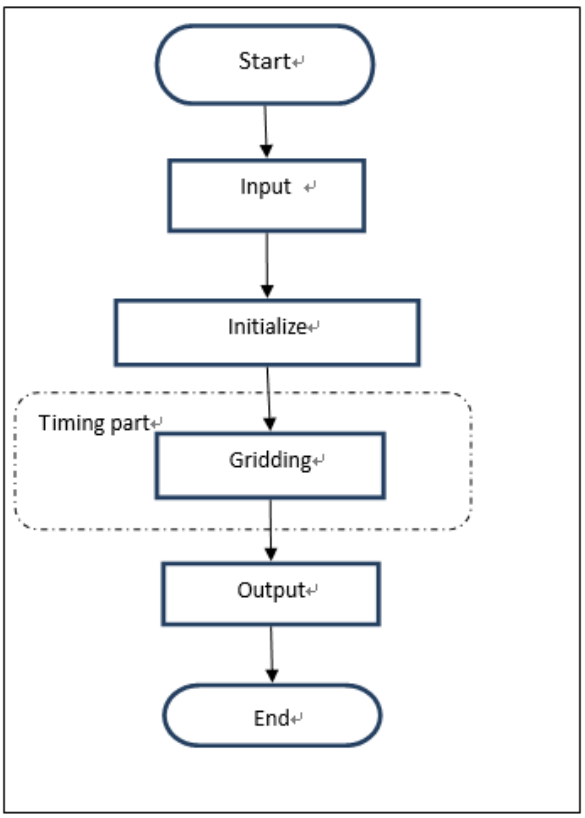

Figure 8: Flow chart of provided program

workload individually. And after all the calculation, the main (rank 0) process reduce all the result together, and write them into output file.

For example, if we have 28 values in the *samples* vector, then the "map" procedure is shown in Figure 9, and the "reduce" procedure is shown in Figure 10.

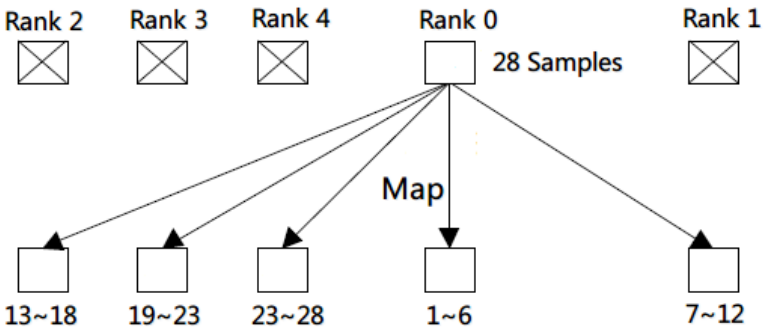

Figure 9: Map tasks to processes

Now we start to analyse the structure of the `gridKernel` function, which is the hotspot of this application.

```
void Benchmark::gridKernel(const int support,
                           const std::vector<Value>& C,
                           std::vector<Value>& grid, const int gSize)
```

The code section shown above is the kernel of the gridding program, and the main part of

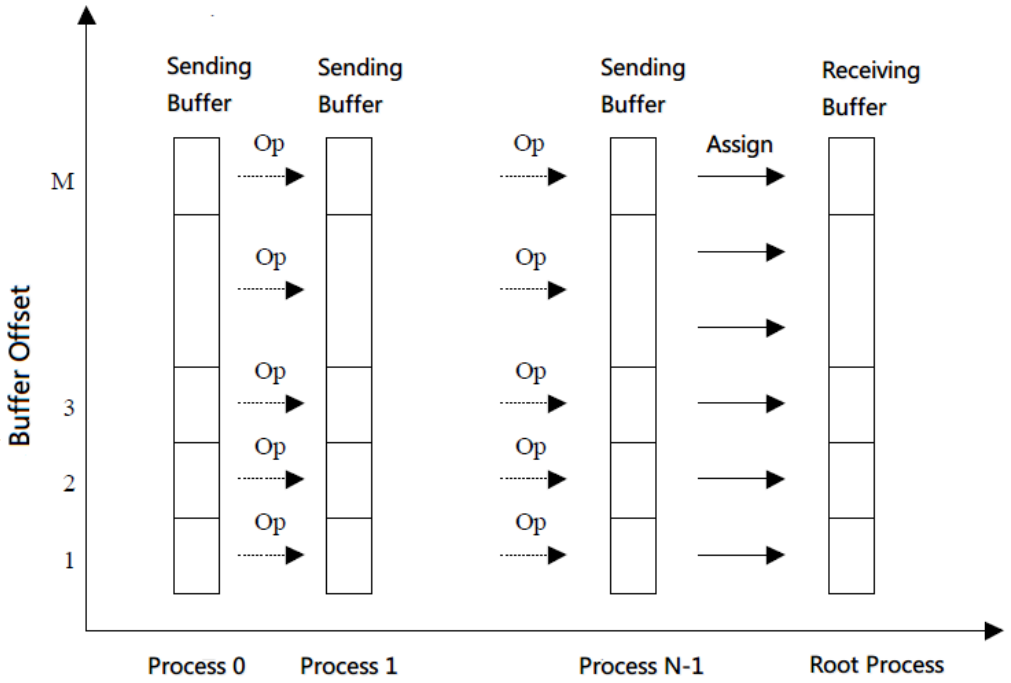

Figure 10: Reduce results of each process

this function is like this:

```
for (int dind = 0; dind < int(samples.size()); ++dind) {
    ......
}
```

For the given workload, there are 3,200,000 samples saved in *samples* vector, so this cycle will be performed for 3,200,000 times. We consider dividing these cycles into n processes in average, each process only deals with a part of all samples. Detail for each process in case of 1 node, 12 processes is given as bellow.

	Process 0	Process 1	.....	Process 11
Step 1	input	input	.....	input
Step 2	initialize	initialize	.....	initialize
Step 3	Gridding (only deal with 1/n samples)	Gridding	.....	Gridding
Step 4	Reduce from the other processes		.....	
Step 5	Output		.....	

Table 5: Configuration of local test machine

6.2.2 OpenMP Parallelization

First we should rewrite the *gridKernel* function. Consider the main part of function gridKernel again:

```
for (int dind = 0; dind < int(samples.size()); ++dind) {                    1
    // The actual grid point from which we offset                           2
    int gind = samples[dind].iu + gSize * samples[dind].iv - support;       3
    // The Convoluton function point from which we offset                   4
    int cind = samples[dind].cOffset;                                       5
    for (int suppv = 0; suppv < sSize; suppv++) {                           6
        Value* gptr = &grid[gind];                                          7
        const Value* cptr = &C[cind];                                       8
        const Value d = samples[dind].data;                                 9
        for (int suppu = 0; suppu < sSize; suppu++)                         10
            *(gptr++) += d * (*(cptr++));                                   11
        gind += gSize;                                                      12
        cind += sSize;                                                      13
    }                                                                       14
}                                                                           15
```

There are 3 *for* cycles in line 1, 6, 10 as the code shown above. If we want to parallel the first *for* cycle with OpenMP, we would have to face the "writing conflict" problem of "**gptr** pointer. Actually, the *gind* value of each thread is calculated with the use of *samples[dind].iu*, *samples[dind].iv* and *support* in line 3, thus each thread would write a range of [gind, gind + gSize*gSize) in *grid* vector. But the ranges of each thread may overlap, so this leads to a conflict. Since the cost for adding "write lock" or "atomic operation" is not cheap, this way is not feasible. We have to focus on the second *for* cycle instead.

After some thought, we finally decide to unroll the inner loop, and add *#pragma* directive on line 4 to parallelize this loop, just as the code shown below. In this way, this hotspot code section could be successfully parallelized.

```
for ( int dind = start; dind < end; ++dind ) {                              1
    int gind_base = samples[dind].iu + gSize * samples[dind].iv - support;  2
    int cind_base = samples[dind].cOffset;                                  3
#pragma omp parallel for schedule(static) num_threads(thread_num)           4
    for ( int idx = 0; idx < sSize * sSize; idx++ ) {                       5
        int gind = gind_base + gSize * (idx / sSize) + idx % sSize;         6
        grid[gind] += samples[dind].data * C[cind_base + idx];              7
    }                                                                       8
}                                                                           9
```

6.2.3 Optimization Results

We use **one node** to test our optimized program on the remote CPU+MIC hybrid cluster, but we only make use of the CPU to do computation here. Table 6 and Table 7 describes the optimization result of MPI version and MPI+OpenMP version. Note that we repeat each job for 3 times, and take the best one.

Num of Processes	**Time (s)**	**Gridding rate** (million grid points per second)

Table 6: Results of MPI version

6.3 Optimization on CPU+MIC platform

It is easy to add simple *#pragma* directive to offload all the computation tasks to MIC acceleration card, just transfer all the needed data — the *samples, grid, C* vectors to the card would

Num of Processes	Num of Threads	Time (s)	Gridding rate (million grid points per second)

Table 7: Results of OpenMP version

be OK. But the actual optimization result is very poor. This is because the actual amount of calculation for the inner loop, which starts from line 5, is very small, just about 10^4 plural plus and multiply operations. Thus, not all the threads are involved in the calculation, and time was spent more on thread creation and destruction. In most cases, there are only half of maximum available threads started when one cycle is finished, just as shown in Figure 11; and after this point, all the threads are destroyed to get to next loop.

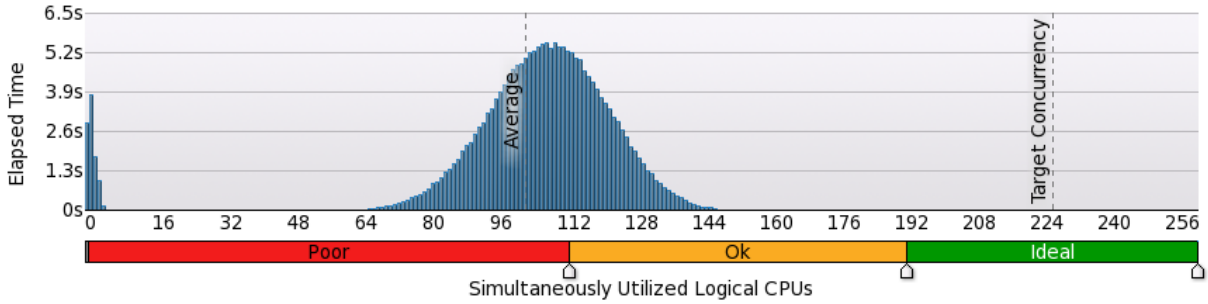

Figure 11: Parallelization result when parallelizing inner loop

This means we should again turn our eyes to the outer loop. Since we can have up to 200+ threads running on MIC card concurrently, and extra cost would be too expensive if we try to parallelize the inner loop, so we can allocate a private memory space for each thread to store the calculated *grid* vector, and reduce them all together to get a correct result when calculation is finished. This solution could be understood as a thread level Map-Reduce algorithm, but its problem could be obvious:

6.3.1 Thread Level Map-Reduce

6.3.2 Offload on MIC

6.4 Extra Optimization

6.5 Summary

References

[1] https://software.intel.com/zh-cn/intel-software-technical-documentation

[2] MIC [M]. , 2012.

Appendices

Appendix A Problems and solutions

A.1 ssh_exchange_identification: Connection closed by remote host

There is a limit to the number of ssh connections coming from the same machine that the sshd daemon allows. The default number is 10. Therefore, if one runs namd and asks for more than 10 threads for each node, the ssh connection will be refused, by default.

This option is set in the **/etc/ssh/sshd_config** file. One needs to change **MaxStartups 10** to something greater.