Parallel Benefit on Different Programming Paradigms

Chau-Yi Chou, Sheng-Hsiu Kuo, Chih-Wei Hsieh, Tsung-Che Tsai and Hsi-Ya Chang

National Center for High-Performance Computing, Taiwan

Abstract - *Multi-core platforms become ubiquitous nowadays. Even laptops contain multi-core processors now. There are multiple cores in a chip or socket or die. A computing node contains multiple chips. Multi-core platforms are rapidly increasing and the number of cores on these platforms is increasing rapidly too. How to enjoy the benefits of parallel computing on the multi-core platforms plays a key role in High Performance Computing. With the increasing complexity of modern multi-core processors, the problem of distributing a software application across different cores to maximize the utilization of the computing power becomes more and more difficult. Different programming patterns great influence the program performance. We implement different parallel programming paradigms on Himeno Benchmark via hybrid MPI/OpenMP in this paper. Moreover, we will evaluate the performance of those on NCHC GPU Cluster and NCHC ALPS. We establish a Roofline Model for NVIDIA GT200, too. We hope the results can give some useful information to the user of HPC.*

Keywords: Performance Evaluation, Parallel Programming, MPI, OpenMP

1 Introduction

Multi-core platforms become ubiquitous nowadays. Even laptops contain multi-core processors now. There are multiple cores in a chip or socket or die. A computing node contains multiple chips. For example, Intel X5472 consists of dual die quad-core CPUs manufactured on a 45 nm process. Multi-core platforms are rapidly increasing and the number of cores on these platforms. Moreover, many users of High Performance Computing, HPC, adopt the multi-core platforms. How to enjoy the benefits of parallel computing on the multi-core platforms plays a key role in HPC.

With the increasing complexity of modern multi-core processors, the problem of distributing a software application across different cores to maximize the utilization of the computing power becomes more and more difficult. Different programming patterns big influence the program performance. Two common programming patterns of parallel program are Message Passing Interface [1], MPI, and OpenMP [2]. MPI had widely used to parallel program on traditional parallel platforms and PC Cluster since 1994. Scientists have obtained a great help on MPI programming pattern in the last few decades. Even it enters multi-core platforms ear. Naturally it is in the wake of overheads, such as message passing inside a node and duplicate memory location.

OpenMP programming pattern is implemented for high efficient computing on Symmetrical multiprocessor system, especially on a multi-core platform. It adopts a fork-and-join execution model, that is, it is thread level parallelism. The behavior of compute is in system bus level and threads are able to share a memory space, but it is limited by scale. The version 2.5 of standard was released in 2005. Most of the compilers (Fortran or C or C++) support the functions of "directive", runtime libraries, and environment variables. Moreover, the version 3.0 of standard, which contains "task" parallelism implementations, was released in May 2008.

We intuitively think the hybrid MPI/OpenMP computing that MPI mainly handles inter message passing, while OpenMP focus on intra computing. But it inherits diverse hardware characteristics, such as the number of multi cores in a die, the number of die in a node, the bandwidth of memory and system bus, how many cores shared resources, and so on. In general, there are two hybrid programming paradigms on multi-core platforms. One is like this model, while the other model is that one adopts a "Parallel Region" directive of OpenMP and master thread to handle the message passing.

We adopt Himeno Benchmark [3], which performs computations of 19-points stencil, to evaluate the performance on different programming patterns of parallel program on NCHC (National Center for High-Performance Computing) platforms, "GPU Cluster", which contains 16 of Intel X5472, and NCHC ALPS hereafter. [4] depicts that the "Stencil" computation, such as Himeno Benchmark, is limited by the bandwidth of memory based on Roofline Model. It is conceivably improved the performance of parallel programs based on Thread Level Parallelization only.

Firstly, we evaluate the performance of Himeno Benchmark on 8 cores in a node on GPU Cluster on different compilers. We implement the hybrid MPI/OpenMP programming patterns on Himeno Benchmark and come out these results on hybrid mode and pure MPI mode on different mapping patterns of 8 nodes (64 Cores) of GPU Cluster. We will present the results of Himeno Benchmark and High Performance Linpack, HPL[5], for CPU binding on NCHC ALPS. We also illustrate the Roofline model on NVIDIA GT200. We hope the results can give some useful information to the user of HPC.

2 Test beds and software

We adopt two test beds, NCHC GPU cluster built in 2010 and upgraded in 2011 and NCHC ALPS built in 2011. We implement MPI/OpenMP hybrid for Himeno Benchmark and evaluate the performance on both platforms. HPL is employed for CPU binding on NCHC ALPS.

2.1 NCHC GPU Cluster

GPU Cluster contains 16 of Intel X5472, which consists of dual die quad-core CPUs manufactured on a 45 nm process. The motherboard adopts Tempest i5400XT (S5396) and Intel 5400 Chipset as shown in Figure 1. Figure 2 shows the logic picture, system bus and bandwidth of memory.

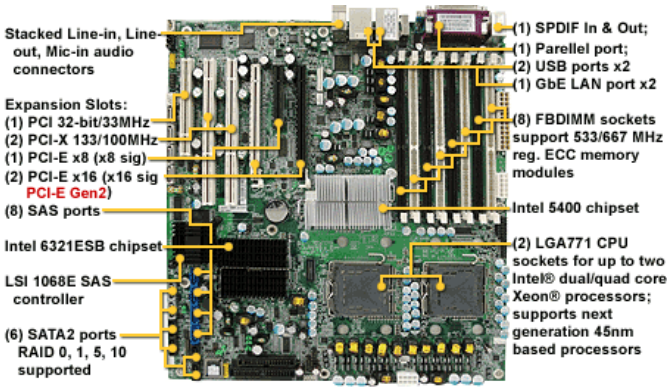

Figure 1. Motherboard of GPU Cluster

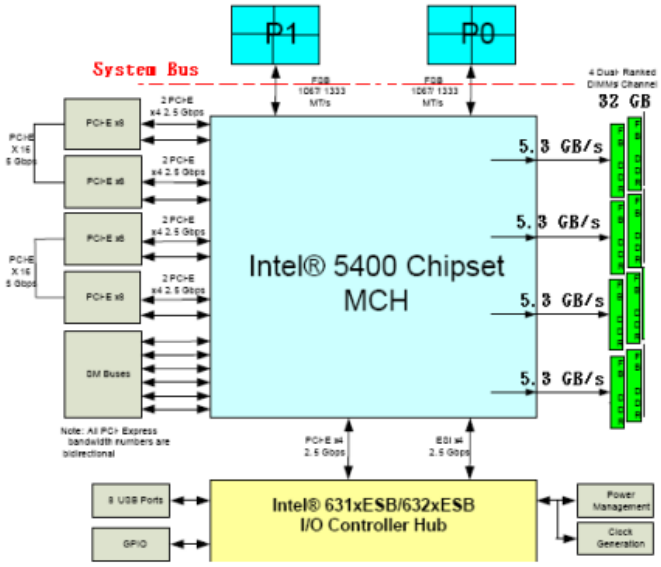

Figure 2. Intel 5400 Chipset

A computing node contains 32 GB RAM and a DDR Infiniband connected together. We adopt OpenSuSE 11.0 as Operation System, OS, different compilers, its version and its compiler option as Intel 11.0 (**`ifort -openmp –O3 -fast`**), PGI 9.0 (**`pgfortran –mp –O3 -fast`**), and GNU 4.3.2 (**`gfortran –fopenmp –O3`**), and MPI middleware as OpenMPI 1.2.8.

2.2 NCHC ALPS

The hardware of computing nodes on NCHC ALPS consists of 600 of Acer AR585 as shown in Figure 3. They are connected together with Qlogic InfiniBand in 4x QDR (40Gb) and the bandwidth throughput of this system achieves 51.8 Tbps. In logic point of view, the system comprises 8 computing clusters, which consists of 4 of AMD Opteron 6174 inside 12 cores running at 2.2 GHz, that is, 48 cores a node sharing 128 GB RAM in 4-memory-controller non-uniform memory access architecture, and 1 large memory cluster that includes 4 of AMD 6136, which comprises 8 core running at 2.2 GHz, that is, 32 cores a node sharing 256GB RAM. There are 25,600 computing cores of AMD Opteron 6100 in this system and the maximal Linpack achieves at 177 TFlops and at 42th place in Top500 list in June 2011 [6].

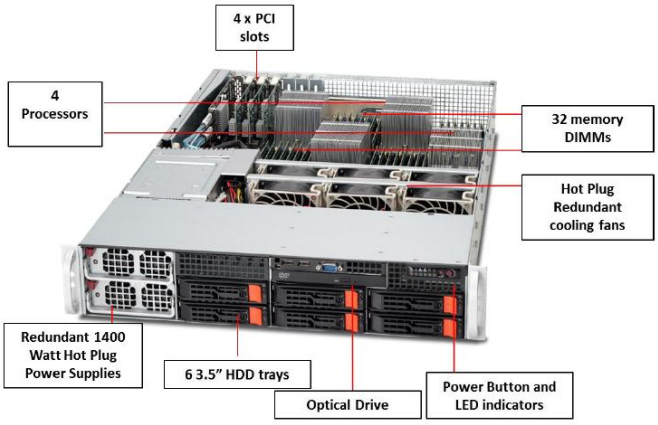

Figure 3. The architecture of Acer AR585

This system adopts the Novell SuSe Linux Enterprise 11 SP 1 for its operation system. The parallel file system is Lustre. Platform LSF is for job scheduler and queuing system. Message passing interface libraries are installed such as Platform MPI, OpenMPI, mvapich, and so on. The debug tool is Allinea DDT, Distribut Debugging Tool. There are four compilers in this system: Intel, PGI, Open64, and GNU. Intel MKL and AMD ACML, AMD Core Math Library, inherit this system for math libraries.

2.3 Himeno Benchmark

To solve the pressure commonly adopts Poisson equation solver in computational fluid dynamic, such as incompressible Navier-Stokes equations solver. The Poisson equation is shown as Figure 4 and the kernel computing of Himeno Benchmark. Use central finite difference and Jacobi iteration. To calculate the value of a point requires reading 18 points of neighbors, named it as "19-Points Stencil". It performs 34 of floating-point operations and uses 14 of three-dimensional matrices per iteration. The computational intensity achieves 0.6 *Flop/Byte* on a problem size of 1024×512×512 in single precision compute (required around 14*GB*) sequentially.

Poisson Equation: $\Delta p = \rho$

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2} + \alpha \frac{\partial^2 p}{\partial x \partial y} + \beta \frac{\partial^2 p}{\partial x \partial z} + \gamma \frac{\partial^2 p}{\partial y \partial z} = \rho$$

$$\frac{p_{i+1,j,k} - 2p_{i,j,k} + p_{i-1,j,k}}{\Delta x^2} + \frac{p_{i,j+1,k} - 2p_{i,j,k} + p_{i,j-1,k}}{\Delta y^2}$$

$$+ \frac{p_{i,j,k+1} - 2p_{i,j,k} + p_{i,j,k-1}}{\Delta z^2}$$

$$+ \alpha \frac{p_{i+1,j+1,k} - p_{i-1,j+1,k} - p_{i+1,j-1,k} + p_{i-1,j-1,k}}{4\Delta x \Delta y}$$

$$+ \beta \frac{p_{i+1,j,k+1} - p_{i-1,j,k+1} - p_{i+1,j-1,k} + p_{i-1,j,k-1}}{4\Delta x \Delta z}$$

$$+ \gamma \frac{p_{i,j+1,k+1} - p_{i,j+1,k-1} - p_{i,j-1,k-1} + p_{i,j-1,k-1}}{4\Delta y \Delta z} = \rho_{i,j,k}$$

Figure 4. Himeno Benchmark

2.4 The High Performance Linpack, HPL

The High Performance Linpack[5], HPL, employs the LU decomposition to solve a dense $N \times N$ system of linear equation in a floating point workload of 2/3 N^3+2N^2. HPL utilizes LU factorization with row partial pivoting to solve a dense linear system while using a two-dimensional block-cyclic data distribution for load balance and scalability.

3 Methodology

We implement two hybrid models, which both models will be described in detail in next Section, for Himeno benchmark in Fortran 90 and evaluate the performance on NCHC GPU Cluster first. The results are performed for GPU version on NCHC GPU Cluster, too. Moreover, we establish a Roofline model for GT200.

Since NCHC ALPS inherits the non-uniform memory access, NUMA, and consists of 48 cores and 8 of NUMA servers as shown in Figure 5, we expect the limit of system bus and memory bandwidth. In order to get the best performance, we adopt the CPU binding for memory use efficiently. First we find the rank pattern via HPL and evaluate the performance. Next, we compare the performance of CPU binding with those for other models via Himeno benchmark. The performances of Himeno benchmark via different compilers are shown, too.

	0	1	2	3	4	5	6	7
0	10	16	16	22	16	22	16	22
1	16	10	22	16	22	16	22	16
2	16	22	10	16	16	22	16	22
3	22	16	16	10	22	16	22	16
4	16	22	16	22	10	16	16	22
5	22	16	22	16	16	10	22	16
6	16	22	16	22	16	22	10	16
7	22	16	22	16	22	16	16	10

Figure 5. Distance of NUMA server

4 Results

We follow 95% confidence to evaluate the results and employ the ganglia and "top" command to monitor the status of the system for dedicated usage.

4.1 NCHC GPU Cluster

We show that the performance of this program at 3.35 *GFlops*, 2.85 *GFlops*, and 2.57 *GFlops* on 8 cores a node via Intel, PGI, and GNU, respectively. It is expectable. Intel compiler gets more benefit of intrinsic computation involving Streaming SIMD Extensions than the others. The performance score of Intel compiler slightly surmounts the peak performance of 3.20 *GFlops*. Commercial PGI compiler shows the performance based on standard computational pattern. The score is lower than the peak performance. The public GNU compiler obtains the worst performance in three compilers.

We implement two hybrid MPI/OpenMP parallel programming patterns: First Model (Hybrid Model 1) is that MPI handles inter message passing between nodes, while OpenMP handles intra computation between 8 cores a node. Its advantage is that the data and program flow are clear. That is, the program is able to be divided into some sub-program blocks, which many sub-program blocks perform heavy operation, while few sub-program blocks perform message passing between nodes. The downside is that all threads are idle on message passing. Amdahl's law points that it big decreases the parallel performance!

The other model (Hybrid Model 2) is that one use a parallel region involves all operations and message passing, which master thread performs message passing between nodes. The race conditions become more and more. The program structure and data flow becomes complicated, too. That is, a programmer requires more human time to better performance. It is opportunity to get parallel efficiency in hidden overhead of message passing carefully. The first model shows 15.0 *GFlops* on 8 nodes with Intel compiler and OpenMPI 1.2.8, while the second model shows 22.50 *GFlops*. It is expectable that the last model presents better performance than those on the first model. We very surprise that different programming patterns can improve 7.5 *Gflops*!

Though MPI 2 has new features of communicator management and OpenMP 3.0 increases the task parallelism, we don't unfortunately perform our overlap version between computation and massage passing (**MPI_THREAD_MULTIPLE**) on our test beds. We adopt MPI 1.2 standard and OpenMP 2.5 to implement our hybrid MPI/OpenMP parallel programming pattern as end-users. The second model of hybrid MPI/OpenMP parallel programming patterns with Intel compiler and OpenMPI 1.2.8 are evaluated on GPU Cluster. Therefore, MPI can perform diverse MPI task mapping, for example a node has 1×1×8, 1×8×1, 8×1×1, 1×2×2, and so on. That is, the data decomposition is 1-dimension in z- or y- or x-direction or 2-dimension in y- and z-direction.

Like MPI mapping, hybrid MPI/OpenMP parallel programming pattern contains large amount of combination of MPI and thread. We evaluate all mapping on combinations of MPI Processes and threads to get interest results. First of all, we define "1×2×4×8" as x-, y-, and z-directions of MPI Processes and the number of threads a MPI Process, that is, x-, y-, and z-directions use 1, 2, 4 MPI Process, respectively, and a MPI Process uses 8 threads. The special case "4×4×4×1" means pure MPI programming paradigm, because of a thread used a MPI process. Table 1 shows the partial (better) results in *GFlops* on 8 nodes (64 cores in total) on different programming paradigms. To our surprise, the pure MPI programming paradigm (4×4×4×1) outperforms!

This is because that the Intel compiler abundantly enjoys the benefit of the hardware, such as SSE intrinsic. Hybrid programming paradigm doesn't have enough room for thread level parallel. Our experience via PGI compiler, pure MPI obtains 19.18 *GFlops* vs. Hybrid model achieves 20.99 *GFlops*. Our opinion is confirmed. The other reason is hardware limitation, such as, obstruction of memory bandwidth. We adopt the Phillips's results presented in Cluster 2009 conference to establish a Roofline Model for GT200 as shown in Fig. 5. The performance of CUDA version of Himeno Benchmark achieves at 767 *GFlops!*

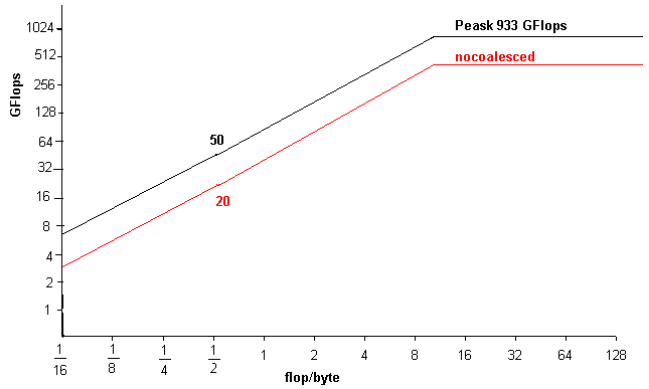

Figure 5. Roofline model on GT200

4.2 NCHC ALPS

For our clear description, we define the notations, **Intel**, **Op64**, **GNU**, and **PGI**, in Table 1. The compiler option is "**Ofast**" for Op64 and GNU, while it is "**fast**" for Intel and PGI.

Table 1. Notations

Notation	Compiler	MPI
Intel	Intel 12.0	openMPI 1.4.3
Op64	Open64 4.2.5	openMPI 1.4.4
GNU	gcc 4.6.2	openMPI 1.4.4
PGI	PGI 11.10	openMPI 1.4.4

We adopt N=1000 for HPL using two cores to obtain the best CPU bind mapping 0-3. We adopt the mapping for **CPU bind** model hereafter and use GNU compiler with AMD acml5.1.0 single thread. Table 2 shows the maximal Linpack in *GFLOPS* for different N=80000 vs. N=100000 with/out CPU bind on 48 core a node. CPU bind model outperforms.

Table 2. Maximal Linpack in *GFLOPS* for different N with/out CPU bind

N	CPU bind	Not CPU bind
80000	286.7	278.0
100000	287.9	274.7

Table 3 depicts the performance of Himeno benchmark in *GFLOPS* for different compilers for $1024 \times 512 \times 512$ problem size in $3 \times 4 \times 4$ partition pattern with/out CPU bind on 48 cores a node. As we expected, CPU bind model outperforms. It is very interesting for CPU bind that PGI outperforms on NCHC ALPS, instead of Intel! It is different from those performed on NCHC GPU Cluster. When we do not use bind processor, Intel outperforms like those on NCHC GPU Cluster. It is because that PGI shows slightly better than Intel for memory affinity on NCHC ALPS.

Table 3. Performance of Himeno benchmark in *GFLOPS* for different compilers for 1024×512×512 problem size in 3×4×4 partition pattern with/out CPU bind

Compiler	CPU bind	Not CPU bind
PGI	36.69	17.02
Intel	35.93	20.12
Op64	31.77	16.22
GNU	20.44	12.12

Table 4 depicts the performance of Himeno benchmark in *GFLOPS* for different partition patterns for 1024×512×512 problem size with/out CPU bind on 48 cores a node. As we expected, CPU bind model outperforms, again. It is very interesting for CPU bind that the three-dimensional partition pattern, $4 \times 4 \times 3$, cannot enjoy the benefits of parallel computing, while the two-dimensional partition pattern, 8×6×1, outperforms. It is different from those on NCHC GPU cluster again.

Table 4. Performance of Himeno benchmark in *GFLOPS* for different partition patterns for 1024×512×512 problem size in with/out CPU bind

partition patterns	CPU bind	Not CPU bind
48×1×1	26.08	9.25
1×48×1	29.25	14.20
1×1×48	27.82	11.53
8×6×1	40.85	26.12
6×8×1	40.24	26.15
6×1×8	36.67	19.54
3×4×4	36.69	17.02
4×3×4	34.79	27.57
4×4×3	40.09	22.30

Table 5 shows the performance of Himeno benchmark in *GFLOPS* for different processor binds for 1024×512×512 problem size in 8×6×1 partition pattern on 48 cores a node.

Rank model, which we define the processor mapping by myself, outperforms. The model that binds each MPI process to a core show comparable results to those on Rank model. The other model, which it binds each MPI process to a processor socket, performs worse results. Every core performs around 60% of workload based on "**top**".

Table 5. Performance of Himeno benchmark in *GFLOPS* for different processor binds for 1024×512×512 problem size in 8 ×6×1 partition pattern on 48 core a node

partition patterns	GFLOPS
Rank model	40.85
Bind each MPI process to a core	40.07
Bind each MPI process to a processor socket	14.75
Not bind processes	26.12

5 Conclusion

We implement two hybrid MPI/OpenMP models for Himeno Benchmark and evaluate the performance on NCHC GPU Cluster and NCHC ALPS. Intel compiler outperforms on NCHC GPU Cluster, while PGI compiler outperforms on NCHC ALPS for CPU bind! It is because that Intel compiler gets more benefit of intrinsic computation involving Streaming SIMD Extensions on Intel platform than PGI and GNU. Moreover, it does not have enough room for thread level parallel via hybrid MPI/OpenMP. Consequently, The pure MPI parallel programming paradigm on 4×4×4 partition pattern outperforms on NCHC GPU Cluster!

Himeno Benchmark and High performance linpack can enjoy the benefits of parallel computing for CPU binding on NCHC ALPS from our various evaluations. The two-dimensional partition pattern, 8×6×1, with PGI compiler outperforms for Himeno Benchmark on NCHC ALPS. Our defined rank model outperforms for Himeno Benchmark and High performance linpack on NCHC ALPS. We establish a Roofline Model for GT200, too.

Acknowledgment

We would like to thank NVIDIA Co. for supporting CUDA version of Himon Benchmark. We are grateful to the National Center for High-Performance Computing for computer time and facilities.

References

[1] Snir, M., Otto, S., Huss-Lederman, S., Walker, D. and Dongarra, J. *MPI: The Complete Reference*, MIT Press, Cambridge, MA, 1996

[2] Ayguade, Eduard, Copty, Nawal, Duran, Alejandro, Hoeflinger, Jay, Lin, Yuan, et al. The Design Of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 404-418, 2009

[3] Himeno Benchmark **http://accc.riken.jp/HPC_e/HimenoBMT_e.html**

[4] Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM* , volume (52), 65-76, 2009

[5] HPL Web site, **http://www.netlib.org/benchmark/hpl/**.

[6] Top 500 Web site, **http://top500.org**