Proposal for the Preliminary Contest of ASC16

Beihang University

Mar 2, 2016

Contents

I. Supercomputing Activities at Beihang University

1. Research Activities

High performance computing is one of major research directions of Beihang University. Currently there are a couple of research teams focus on HPC techniques and applications in the school of computer science & engineering and other schools such as school of aviation, school of materials, school of mathematics, etc.

Research team in school of computer, led by Prof. Depei Qian and Prof. Mingfa Zhu, mainly focuses on key techniques of HPC and applications including programming methods of multi-/many-core processors, performance tools, parallel file system, system power management, system simulation technology, etc. As a partner, we took part in several key projects of National Hi-tech R&D program (863 program) of China in HPC direction, e.g. national high performance computing and service environment(CNGrid), 100TF Lenovo DeepComp-7000 supercomputer (2008), 1PF Dawning Nebulae supercomputer (2010), Key technologies of Exa-scale supercomputer (2014) and a series of HPC applications such as drug discovery, climate change, aviation, etc.

Supercomputing-related award and some representative papers are listed as following:

[1] China National Grid, National science & technology progress award, second prize, 2007

[2] DeepComp-7000 Supercomputer, Beijing science & technology progress award, first prize, 2013.

[3] Huang Zhibin, Zhu Mingfa, Xiao Limin. LvtPPP: live-time protected pseudo-partitioning of multi-core shared caches, IEEE Transactions on Parallel and Distributed Systems, 2012:1-11.

[4] Ludan Zhang, Yi Liu, Rui Wang and Depei Qian, Lightweight dynamic partitioning for last-level cache of multicore processor on real system, The Journal of Supercomputing, 2014.

[5] Quan Chen, Hailong Yang, Lingjia Tang, Jason Mars, Baymax: QoS Awareness and Increased Utilization of Non-Preemptive Accelerators in Warehouse Scale Computers, Architectural Support for Programming Languages and Operating Systems(ASPLOS'16), 2016.

2. Supercomputing Facilities

Currently there are several HPC systems in Beihang university. The school of computer has different kinds of HPC platforms including servers, GPU, Intel MIC and FPGA accelerators, total aggregated performance is about 30TFlops. There is also a 64-node small cluster located in network center providing free computing services to the whole university.

Finally, a 200TFlops HPC system is under planning and will be built in the National Laboratory of Aviation Technology, located in Shahe campus of Beihang university.

3. Supercomputing-related Courses

Supercomputing-related courses are listed in Table 1.

Table 1 Supercomputing-related courses in Beihang university

	Course Name	Students	Teacher
1	High Performance Computer Architecture	Graduate students, school of computer	Prof. Mingfa Zhu
2	Parallel Programming	Undergraduate students, school of computer	Prof. Yunchu Li
3	Parallel Programming	Graduate students, school of computer	Prof. Yi Liu
4	Parallel Computing	Graduate students, university	Prof. Tiegang Liu
5	Parallel Computing	Ph.D students, school of advanced engineering	Prof. Teigang Liu and Yi Liu

II. Introduction of the Team

After receiving invitation of 2016 ASC student Supercomputer Challenge, we organize our team in following steps:

(1) Broadcast a call-for-participants' notification to all of grade-3 undergraduate students of school of computer (about 200 students);

(2) Two introduction meetings are held for all of the interested students to introduce supercomputing, the ASC16 contest and requirements for participants, on this basis, 11 students volunteer to standby as candidates;

(3) After exchanging opinions among professors in HPC direction and some discussions, the advisor and contact person of the team are determined. In addition, the advisor and two other teachers organize an advising group to give extra support such as working place, equipment and experimental environments;

(4) After the preliminary contest notification of ASC16 is released, a kickoff meeting is held for candidate students to introduce works that need to be done and time schedule. After a discussion, students are divided into four groups, corresponding to four part of works, i.e. system design, HPCG test, WASNUM_WAM test and DNN optimization;

(5) Since the beginning of winter vacation, 6 students chose to stay to do works, and after Chinese new year festival, some students return back to work before the ending of winter vacation, and final 5 team members are selected.

The team consists of 5 undergraduate students from School of computer science and engineering in Beihang university, as listed in Table 2.

Table 2 Team member list

Name	Gender	Grade / Role	Responsibility
Luming Wang	Male	Grade 3 / Team leader	System design & DNN
Changxi Liu	Male	Grade 3 / Team member	DNN optimization
Baozheng Liu	Male	Grade 3 / Team member	DNN optimization
Dongdong Yang	Male	Grade 3 / Team member	MASNUM_WAM test
Zhizhu Liu	Male	Grade 3 / Team member	HPCG test
Yi Liu	Male	Professor / Advisor	Advisor
Hailong Yang	Male	Ph.D / Advisor	Advisor
Yucong Zhou	Male	M.S. student / Advisor	Advisor

北京航空航天大学计算机学院
Beihang University
School of Computer Science & Engineering

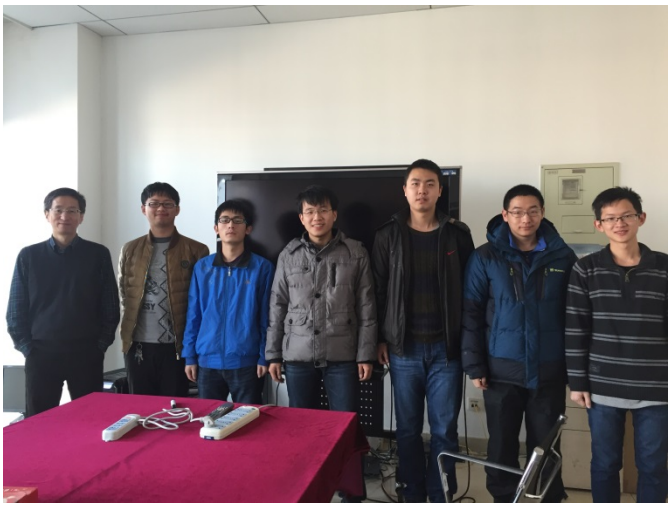


Figure 1 Logo and group photo of the team

The slogan of our team is: "***To do the most, to show the best !***"

III.Technical Proposal

1. System Architecture Design

1.1 System Architecture Design

Our high performance system is based on cluster architecture which is the most popular architecture in Top 500. The node of our cluster is Inspur NF5280M4 server (2 * Intel Xeon E5-2680v3 CPU and 128G Memory).

Power consumption of each component is listed in Table 3.

Table 3 Power consumption of each component

Item	Configuration	Power consumption estimation
CPU	Intel Xeon E5-2680v3, 2.5 GHz, 12 cores	120W
Memory	16G, DDR4, 2133MHz	7.5W
Hard Disk	1T SATA	10W
XEON PHI-31S1P	57 cores, 1.1GHz, 1003GFlops, 8GB GDDR5 Memory	270W
NVIDIA tesla	NVIDIA Tesla K40, 2699 CUDA cores, 0.745 GHz	235W
InfiniBand HCA card	Mellanox ConnectX-3, single port QSFP, FDR	9W
GbE switch	10/100/1000Mb/s,24 ports Ethernet switch	30W
FDR-IB switch	SwitchXTM FDR InfiniBand switch, 36 QSFP port	130W

To keep synchronous, parallel programs exchange messages between processes which may distribute on different machines. Therefore, communication latency and bandwidth between cluster nodes becomes an important factor. InfiniBand is the most commonly used interconnect in supercomputers as its high bandwidth (up to 25Gb/s) and low latency. So we choose InfiniBand for interconnect instead of Ethernet.

We considered and evaluated three kinds of system configurations: CPUs without accelerator, CPUs with Intel MIC and CPUs with GPU. For each kind of configuration, we calculated power consumption of single node, the result is listed in Table 4.

Table 4 Single node power consumption

Type	Configuration	Power consumption estimation
CPU	2*CPU+8*16G Memory+1* Hard Disk+1*HCA card	319W
CPU + MIC	2*CPU+8*16G Memory+1* Hard Disk+1*HCA card+1*MIC	589W
CPU + GPU	2*CPU+8*16G Memory+1* Hard Disk+1*HCA card+1*GPU	554W

Because of the limitation of power consumption (up to 3KW), the cluster can only contain 4 nodes with Intel MIC. That number will be 5 if we choose GPU as accelerator card instead of Intel MIC. However, the cluster can contain 8 nodes if we do not use any accelerator card. In addition, the total power consumption of 9 nodes is 3031W, which means that we can control the power consumption under 3KW by reducing the frequency of CPUs. The details are listed in Table 5.

Table 5 Total Power Consumption

	CPU	CPU+MIC	CPU+GPU
Description	No accelerator	1 MIC / node	1 GPU / node
Node power consumption	319W	589W	554W
Number of nodes (under 3kW)	8(or 9 with CPU frequency adjust)	4	5
Power consumption of interconnection network	160W	160W	160W
Total power consumption	2712W(+319W)	2516W	2930W
Peak performance(GFlops)	3532.8(+220.8)	5778.4	9359.0
Performance per watt(GFlops/Watt)	1.303(1.238)	2.297	3.194
Programmability	Easy	Medium	Hard
Porting + Optimization	Medium	Hard	Hard
Note: Peak performance of Intel Xeon E5-2670v3 = 220.6GFlops Peak performance of Intel Xeon PHI-31S1P=1003GFlops Peak performance of NVIDIA Tesla K40 = 1429GFlops			

As shown in Table 5, the performance per watt of CPU+GPU hybrid cluster is the best in the three candidate system configurations. However, the programing model of GPU is different from that of CPU. To be more specific, it uses CUDA, an application programming interface created by NVIDIA. To obtain performance optimized code, programmers need to take into account of various aspects of low-level hardware configurations, which may lead to difficulties on porting and optimizing programs. In addition, many applications do not support GPU (include MASNUM Wave). Porting applications to GPU will take up a long time. To sum up, although hybrid CPU+GPU platform has great performance per watt, it is not the best choice due to its programmability.

As for CPU+MIC hybrid platform, porting programs to it is also a challenge, although it supports x86-64 instruction set. MIC has 57 cores with 512-bit VPU (vector processing unit). However, each core in MIC is not as powerful as cores in Intel Xeon processor, and system bus bandwidth and latency may also limit the application performance. In addition, to use CPU resources simultaneously, programmers need to consider load balancing between CPU and MIC and the cost of data transfer.

Homogeneous CPU platform has relatively low peak performance, however, low peak

performance in theory does not mean poor performance in practical. Modern CPU is easy to programming, and suited for both computing-intensive and data-intensive applications. In addition, it has perfect compatibility and various software support. Programming and optimizing programs on CPU is relatively easy.

Based on the above discussions, finally we choose homogeneous CPU as the system configuration of our cluster.

1.2 Local Testing Platform

In order to evaluate application performance of ASC16, we built a local testing platform in our lab. Due to limitation of equipment that are available, we only built a small cluster with 4 CPU nodes. We use this cluster to analyze and optimize HPCG and MASNUM Wave application. In addition, we setup another CPU+MIC node to use it as the testing and tuning platform for DNN.

The configuration of our local testing platform is shown in Table 6.

Table 6 Configuration of Local Testing Platform

Item	Name	Detail
CPU cluster (4 nodes)	CPU	2*Intel Xeon E5-2680v3(2.5 GHz, 12 cores)
	Memory	128GB DDR4
	Hard Disk	2*300GB SSD
	IB Switch	Mellanox InfiniBand switch, 8 QSFP port
	HCA Card	2*Mellanox ConnectX-4 HCA card, single port QSFP, FDR
	InfiniBand Cable	4*InfiniBand FDR optical fiber cable, QSFP port
CPU+MIC node (1 node)	Accelerator	1*Intel Xeon Phi coprocessor 31S1

As for software environment, we use CentOS 6.7 as the operating system due to its stability and reliability. To get better performance, we use the latest version of Intel C Compiler and Intel Fortran Compiler. In addition, all of the third party libraries are the latest version. The details are shown in Table 7.

Table 7 Software Environment

Type	Name	Version
Operating System	CentOS release 64-bit	6.7
Linux Kernel	Kernel	2.6.32-573.12.1.el6.x86_64
InfiniBand HCA Driver	MLNX_OFED_LINUX	3.2-1.0.1.1
Compiler	GCC	4.4.7
	ICC	16.0.1
	Ifort	16.0.1
Libraries	Intel MPL Library	5.1 update 1
	Intel MKL Library	11.3
	NetCDF	4.4.0(with --disable-netcdf-4 flag)
	NetCDF-Fortran	4.4.2

2. HPCG Test

2.1 Algorithm Analysis of HPCG

2.1.1 Mathematical Principle

(1) Poisson Equation

Poisson Equation, a kind of elliptic partial differential equation, could be used to solve the

heat diffusion problem, especially the problem in the active power thermal field. The Poisson Equation is:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}=f(x,y,z)$$

or abbreviated to:

$$\Delta u=f$$

Where Δ is Laplace operator, u and f are two ternary functions.

In addition, we need to consider the boundary conditions. We assume that it's Dirichlet condition, then the complete form of Poisson Equation is:

$$\begin{cases}\dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}+\dfrac{\partial^2 u}{\partial z^2}=f(x,y,z),(x,y,z)\in\Omega\\ u(x,y,z)=\varphi(x,y,z),(x,y,z)\in\Gamma\end{cases}$$

or abbreviated to:

$$\begin{cases}\Delta u=f\\ u|_{\Gamma}=\varphi\end{cases}$$

Where φ is a ternary function, and Γ is the boundary of space Ω. In HPCG, the Ω is a cube, and $\varphi=0$ when there is no adjacent cube. This problem could be converted to the problem of solving sparse linear equations by numerical method, such as finite-difference method, finite-element method and finite-volume method.

(2) Finite Difference Method

In Cartesian coordinate system, we establish three groups of plane families that they are orthogonal to each other. Besides, planes are parallel to each other in the family.

$$\begin{cases}x_i=im,i\in\mathrm{N}\\ y_j=jn,j\in\mathrm{N}\\ z_k=kl,k\in\mathrm{N}\end{cases}$$

Where m, n, l are distances between adjacent planes. There will be a series of intersection points. We assume that:

① $m=n=l=h$, h is small enough.

② $\varphi\equiv 0$.

③ Ω is a cube.

④ points are uniformly distributed in the edges, vertexes, faces and inner space of Ω.

Then there will be huge amounts of intersection points uniformly distributed in space Ω. By solving the problem on each node (x_i,y_j,z_k), we could obtain a numeric solution of Poisson Equation.

According to Taylor formula, we do a second-order difference and abbrev $u(x_i,y_j,z_k)$ to $u(i,j,k)$.

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_{(i,j,k)}=\frac{1}{h^2}[u(i+1,j,k)-2u(i,j,k)+u(i-1,j,k)]+\mathrm{O}(h^2)$$

$$\left(\frac{\partial^2 u}{\partial y^2}\right)_{(i,j,k)}=\frac{1}{h^2}[u(i,j+1,k)-2u(i,j,k)+u(i,j-1,k)]+\mathrm{O}(h^2)$$

$$\left(\frac{\partial^2 u}{\partial z^2}\right)_{(i,j,k)}=\frac{1}{h^2}[u(i,j,k+1)-2u(i,j,k)+u(i,j,k-1)]+\mathrm{O}(h^2)$$

Add the three above expressions, we could get:

$$\frac{1}{h^2}\left[\sum u_{adjacent}-6u(i,j,k)\right]=f(i,j,k)+\mathrm{O}(h^2)$$

Where $\mathrm{O}(h^2)$ is tiny. There will be n^3 nodes in cube Ω with n nodes on an edge. Totally,

we could get n^3 equations.

$$\sum u_{adjacent} - 6u(1,1,1) \approx h^2 f(1,1,1)$$

... ...

... ...

$$\sum u_{adjacent} - 6u(1,1,n) \approx h^2 f(1,1,n)$$

... ...

... ...

$$\sum u_{adjacent} - 6u(1,n,1) \approx h^2 f(1,n,1)$$

... ...

... ...

$$\sum u_{adjacent} - 6u(n,n,n) \approx h^2 f(n,n,n)$$

So it is formulated as a sparse matrix problem.

$$\mathrm{A}x = \mathrm{b}$$

Where the order of matrix A is n^3, and the element of main diagonal is constantly equal to -6, with 6 unit elements for every row. The structures of x and b are

$$x = (u(1,1,1), \dots, u(1,n,1), \dots, u(n,n,n))^T$$

$$\mathrm{b} = (h^2 f(1,1,1), \dots, h^2 f(1,n,n))^T$$

According to 0-Dirichlet condition, the value of u on boundary is equal to zero, so that some rows in matrix have less than 7 non-zero elements. Here we set a node associated with 6 adjacent nodes, while in HPCG, one node associated with 26 adjacent nodes and the diagonal element is equal to -26.

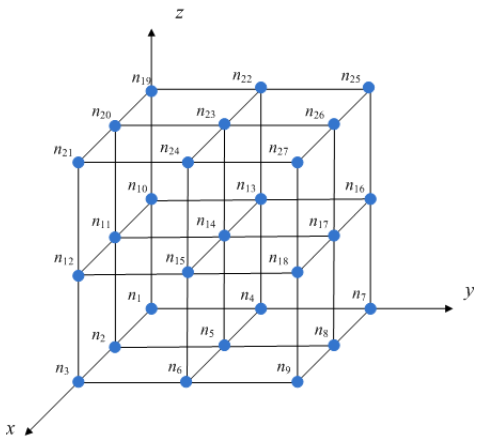


Figure 2 Distribution of nodes on cube Ω

(3) Preconditioned Conjugate Gradient Method

For linear equations $\mathrm{A}x = \mathrm{b}$, there is a quadratic function.

$$\varphi(x) = \frac{1}{2}[x, \mathrm{A}x] - [x, b]$$

Where $[\cdot,\cdot]$ means dot product. There is a theorem for the exact solution x^* of the linear equations.

$$\mathrm{A}x^* = \mathrm{b} \Leftrightarrow \varphi(x^*) = \min_{x \in R^n} \varphi(x)$$

Based on this theorem, the problem of solving sparse matrix is formulated as the problem of finding the minimum of $\varphi(x)$. For this problem, the method widely ever used is Gradient Descent, which initializes a vector $x^{(0)}$, then does a gradient descent in each iteration. The value of $x^{(k)}$

will approach to x^* after enough iterations. However, the descent speed will be slower after some iterations, which makes algorithm efficiency reduce. The Conjugate Gradient Method is an improvement of Gradient Descent. After the first iteration, it uses a series of conjugate gradients so that the convergence step-size will be n^3 theoretically. The formulas are:

$$\begin{cases} r^{(0)} = \mathrm{b} - \mathrm{A}x^{(0)}, z^{(0)} = r^{(0)} \\ \alpha_0 = \dfrac{[r^{(0)}, z^{(0)}]}{[\mathrm{A}z^{(0)}, z^{(0)}]} \\ x^{(1)} = x^{(0)} + \alpha_0 z^{(0)} \end{cases}$$

$$\begin{cases} r^{(k)} = \mathrm{b} - \mathrm{A}x^{(k)}, z^{(k)} = r^{(k)} + \beta_k z^{(k-1)} \\ \alpha_k = \dfrac{[r^{(k)}, z^{(k)}]}{[\mathrm{A}z^{(k)}, z^{(k)}]} \\ \beta_k = -\dfrac{[r^{(k)}, \mathrm{A}z^{(k-1)}]}{[z^{(k-1)}, Az^{(k-1)}]} \\ x^{(k+1)} = x^{(k)} + \alpha_k z^{(k)}, k > 0 \end{cases}$$

Where $r^{(k)}$ is the gradient direction, $z^{(k)}$ is the improving direction. Moreover, it could be improved by adding a preconditioner M^{-1} to make the condition number κ smaller and ensure fast convergence.

$$\mathrm{A}x - \mathrm{b} = 0 \Rightarrow M^{-1}(\mathrm{A}x - \mathrm{b}) = 0$$
$$\kappa(M^{-1}\mathrm{A}) < \kappa(\mathrm{A})$$

So that the preconditioned conjugate gradient method is

$$\begin{cases} r^{(0)} = \mathrm{b} - \mathrm{A}x^{(0)} \\ z^{(0)} = M^{-1}r^{(0)} \\ p^{(0)} = z^{(0)} \end{cases}$$

$$\begin{cases} r^{(k)} = r^{(k-1)} - \alpha_{k-1}\mathrm{A}p^{(k-1)}, z^{(k)} = M^{-1}r^{(k)}, p^{(k)} = z^{(k)} + \beta_{k-1}p^{(k-1)} \\ \alpha_{k-1} = \dfrac{[r^{(k-1)}, z^{(k-1)}]}{[\mathrm{A}p^{(k-1)}, p^{(k-1)}]} \\ \beta_{k-1} = -\dfrac{[r^{(k)}, z^{(k)}]}{[r^{(k-1)}, z^{(k-1)}]} \\ x^{(k)} = x^{(k-1)} + \alpha_{k-1}p^{(k-1)}, k > 0 \end{cases}$$

Where $p^{(k)}$ is the preconditioned improving direction.

(4) Multi-grid Method

The main idea of multi-grid is to maintain the convergence speed by smoothing, restriction and prolongation. Using a hierarchy of discretization, it could eliminate all parts of frequency errors while general iteration methods are only relatively effective on specific frequency. In HPCG, a V-cycle multi-grid is used as a preconditioner with a symmetric Gauss-Seidel algorithm at each level.

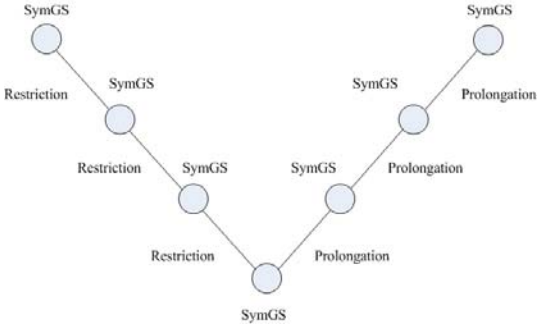


Figure 3 V-cycle multi-grid algorithm

In conclusion, HPCG 3.0 originated from the solving of a three-dimensional heat diffusion problem, where the Poisson equation is converted to sparse matrix by numerical method such as finite-difference method. It solves the sparse matrix by the Conjugate Gradient Method with the multi-grid method as a preconditioner. Based on this, it is implemented as a benchmark in HPCG so that HPCG has a closer connection with real applications than LINPACK.

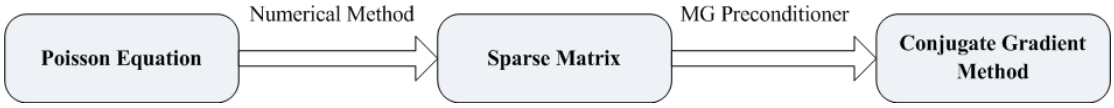


Figure 4 the mathematical principle of HPCG 3.0

2.1.2 Algorithm

There is the key algorithm of the Conjugate Gradient Method in HPCG.

$p_0 \leftarrow x_0,\ r_0 \leftarrow b - Ap_0$
for $i = 1,\ 2,$ **to** max_iterations **do**
  $z_i \leftarrow M^{-1} r_{i-1}$
  if $i = 1$ **then**
    $p_i \leftarrow z_i$
    $\alpha_i \leftarrow \text{dot\_prod}(r_{i-1}, z_i)$
  else
    $\alpha_i \leftarrow \text{dot\_prod}(r_{i-1}, z_i)$
    $\beta_i \leftarrow \alpha_i / \alpha_{i-1}$
    $p_i \leftarrow \beta_i p_{i-1} + z_i$
  end if
  $\alpha_i \leftarrow \text{dot\_prod}(r_{i-1}, z_i) / \text{dot\_prod}(p_i, Ap_i)$
  $x_{i+1} \leftarrow x_i + \alpha_i p_i$
  $r_i \leftarrow r_{i-1} - \alpha_i A p_i$
  if $\| r_i \|_2 <$ tolerance **then**
    STOP

It is a preconditioned conjugate gradient algorithm with the Multi-grid method as a preconditioner, where x stores the iterative solution and r stores the residual. $z_i \leftarrow M^{-1} r_{i-1}$ means do preconditioning to r_{i-1} and store result into z_i. $\text{dot\_prod}(r_{i-1}, z_i)$ is the dot product of r_{i-1} and z_i. $\| r_i \|_2$ is the 2-norm which is equal to $\sqrt{\text{dot\_prod}(r_i, r_i)}$. Ap_i is the product of matrix A and vector p_i.

The algorithm will end after max_iterations times of iterations or the 2-norm of residual is less than tolerance. In program the tolerance is equal to zero so that the algorithm must run max_iterations times which is influenced by the parameter rt.

The time complexity of dot_prod is $O(\text{orderG})$, where orderG is the order of global matrix which is equal to $np \cdot nx \cdot ny \cdot nz$. The time complexity of Ap_i is $O(nonzeros \cdot \text{orderL})$, where $nonzeros$ is the average number of nonzero value per row and orderL is the order of local matrix which is equal to $nx \cdot ny \cdot nz$. The time complexity of $z_i \leftarrow M^{-1} r_{i-1}$ is equal to the time complexity of the symmetric Gauss-Seidel method which is $O(nonzeros \cdot \text{orderL})$. In summary, the total time complexity is:

$$O(nonzeros \cdot \text{orderL}) + \text{max\_iterations} \cdot (O(nonzeros \cdot \text{orderL}) + O(\text{orderG}))$$

After simplification, the result is:

$$O(\text{max\_iterations} \cdot (nonzeros + np) \cdot nx \cdot ny \cdot nz)$$

2.2 Code Analysis of HPCG

2.2.1 Program Structure

The main code of HPCG could be divided into four parts. Firstly, it initializes HPCG problem and MPI, then does a reference timing and optimizes problem according to the user's setting. Finally, it runs the benchmark and reports result. In addition, it could run under the *DEBUG* mode or the *DETAILED_DEBUG* mode by setting compiling options where the customer could get more detailed information from *.txt* file. Single process under the *DETAILED_DEBUG* mode will generate *.dat* files about information of the local sparse matrix and the vectors.

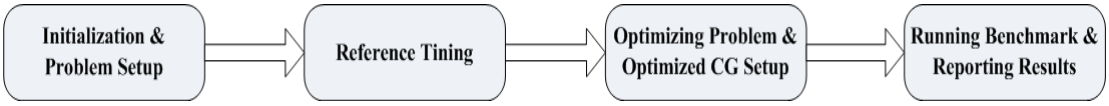


Figure 5 Program structure of HPCG

(1) Initialization & Problem Setup

At the first step, the program initializes the MPI, gets the number of processes `size` and own process number `rank`. Then it extracts parameters from command line or from the file *hpcg.dat*, and setups the problem. Parameters include the edge lengths of the cube nx, ny, nz and the running time rt. When $rt = 0$ the program will run benchmark only once under *QUICK* mode. If $\min\{nx, ny, nz\} / \max\{nx, ny, nz\} < 0.125$, It will report the structure error and exit because the shape of global geometry does not look like a cube.

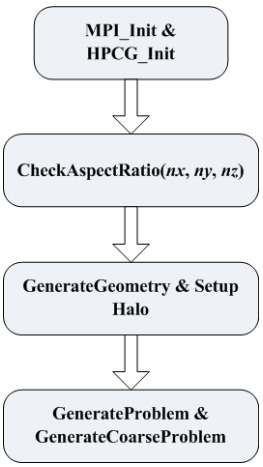


Figure 6 Initialization procedure of HPCG

(2) Reference Timing

After the problem setup, the program runs a reference sparse matrix-vector multiplication timing, a reference multi-grid timing and a reference conjugate gradient timing. It will judge the optimization effect by these reference time costs.

(3) Optimizing Problem & Optimized CG Setup

HPCG provides an interface *Optimize Problem* for users to optimize the data structures and improve performance, then tests the validation. After optimization, the program runs the optimized conjugate gradient timing and calculate the worst time cost *opt_worst_time*.

(4) Running Benchmark & Reporting Results

Finally, it runs the benchmark. The program first calculates:

$$\text{numberOfCgSets} = \frac{rt}{\text{opt\_worst\_time}}$$

Then it runs the optimized conjugate gradient *numberOfCgSets* times, statistics the performance and generates a yaml format file a report. Official resulting execution time should be at least 1800 seconds.

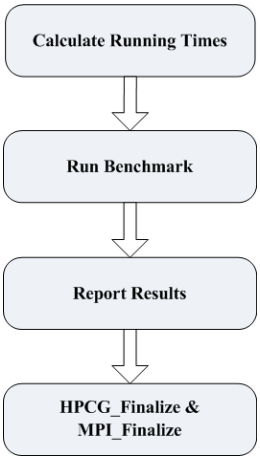


Figure 7 Ending procedure of HPCG

2.2.2 Data Structure

The core data structures in HPCG application include *Vector*, *SparseMatrix* and *Geometry*.

```
struct Geometry {
    int size;                 // number of MPI process
    int rank;                 // this process's rank ([0, size-1])
    int numThreads            // this process's number of threads
    int nx, ny, nz;           // number of i-direction grid points for each local subdomain
    int npx, npy, npz;        // number of processes in i-direction
    int ipx, ipy, ipz;        // this process's location in global domain
};

struct Vector {
    local_int_t localLength;      // length of local portion of the vector
    double *values;           // array of values
};

struct SparseMatrix {
    Geometry *geom            // geometry associated with this matrix
    char *nonzerosInRow  // the number of nonzero in a row (<=27)
    ……
    global_int_t **mtxIndG              // global matrix indices
    local_int_t **mtxIndL         // local matrix indices
    double **matrixValues         // values of matrix entries
    double **matrixDiagonal     // values of matrix diagonal entries
    std::map<global_int_t, local_int_t> globalToLocalMap
    std::vector<global_int_t> localToGlobalMap
    mutable struct SparseMatrix_STRUCT *Ac     // coarse grid matrix
#ifndef HPCG_NO_MPI
    local_int_t numberOfExternalValues
    int *neighbors
    local_int_t *receiveLength    // lengths of messages received
    local_int_t *sendLength             // lengths of messages sent
```

```
        double *sendBuffer
        ……
#endif
};
```

Where Geometry describes the geometry information for each process. For example, there are three processes as below

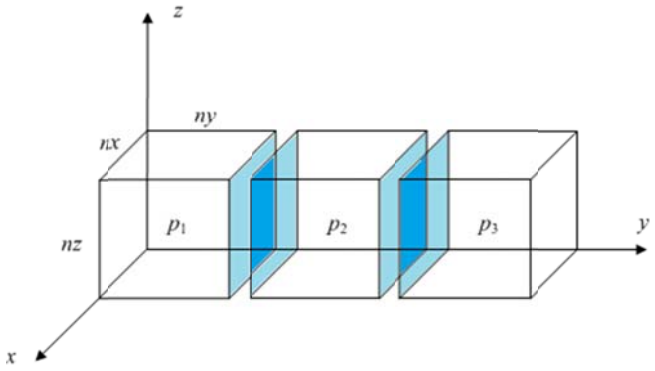


Figure 8 Geometry distribution of processes

Where nx, ny, nz are the parameters extracted from command line or hpcg.dat, and $npy = 3$, $npx = npz = 1$. For process p_2, $ipx = ipz = 0, ipy = 1$. In addition, there will be MPI communications between cubes on adjacent faces which are blue in the above figure.

In the HPCG 3.0 reference implementation, an improved *ELLPACK* storage format is used for the sparse matrix. The new added double-pointer *matrixDiagonal* is used to store the pointer of diagonal element in *matrixValues*. For example, we assume that the maximum of number of non-zero value per row is 2, and the structure is:

$$matrix = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 6 & 0 \\ 4 & 0 & 0 & 7 \end{bmatrix}, mtxInd = \begin{bmatrix} 0 & * \\ 1 & 3 \\ 2 & * \\ 0 & 3 \end{bmatrix}, matrixValues = \begin{bmatrix} 2 & * \\ 1 & 5 \\ 6 & * \\ 4 & 7 \end{bmatrix}$$

$$matrixDiagonal = [2 \quad 1 \quad 6 \quad 7], nonzerosInRow = [1 \quad 2 \quad 1 \quad 2]$$

The difference between *mtxIndG* and *mtxIndL* is that *mtxIndG* describes the indices in global matrix. In Figure 9, the node n_{25} of p_3 in fact is the node n_{79} of global matrix. There are globalToLocalMap and localToGlobalMap in SparseMatrix for mapping. For example,

$$\begin{cases} globalToLocalMap(n_{79}) = n_{25} \\ localToGlobalMap(n_{25}) = n_{79} \end{cases}$$

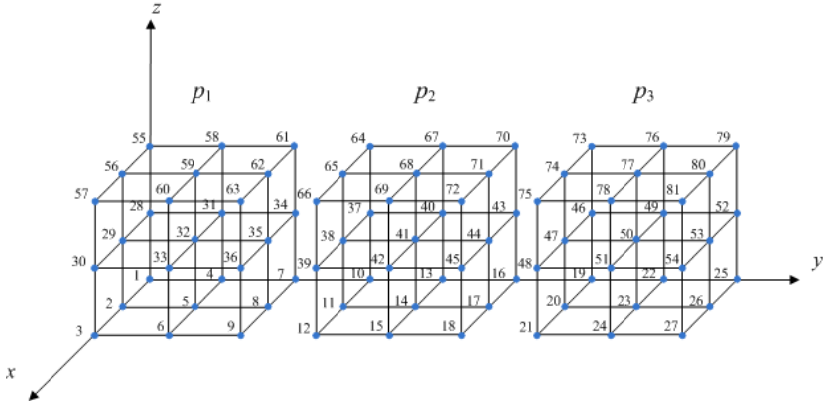


Figure 9 Global nodes distribution between processes

In HPCG, five vectors *x*, *b*, *xexact*, *x_overlap*, *b_computed* and one sparse matrix *A* are used,

so that the space complexity is approximately:

$$5 \times O(nx \cdot ny \cdot nz) + O(nonzeros \cdot nx \cdot ny \cdot nz) = O((nonzeros + 5) \cdot nx \cdot ny \cdot nz)$$

Where $nonzeros$ is the maximum of number of non-zero values per row.

2.3 Optimization

2.3.1 Optimization of Configuration Parameters

In comparison to LINPACK, the number of parameters in HPCG is relatively fewer. It only includes parameters managing the number of processes, the size of the cube and the number of threads for each process. However, it is necessary to analyze these parameters for best performance on the local testing platform. The optimization configuration parameters are listed in Table 8.

Table 8 Optimization of configuration parameters

Configuration parameters	Analysis for optimization
$nx,\ ny,\ nz$	ni is the side length of the cube at axis i and the size of sparse matrix is approximately $27 \cdot nx \cdot ny \cdot nz$. By tuning these parameters, we can change the workload for every process and the surface area of cube for MPI communication. At normal state, the fewer these parameters are, the higher the performance will be.
rt	rt tunes the running time of the benchmark. Long running time could make the performance result reliable. Official results execution time must be at least 1800 seconds.
OMP_NUM_THREADS	It's the number of threads in each process for OpenMP. When the order of matrix is few, thread method is not necessary.
np	np is the number of processes. By computing the optimal shape from $nx,\ ny,\ nz$ and np, it makes the global geometry as close as possible similar to a cube rather than a plate. More processes could increase the workload of MPI communication due to increment of adjacent area.

2.3.2 Optimization of Compiling Parameters

Compiling options and parameters play a role in HPCG test, so we modified some of them for optimization. The compiling options and parameters we used:

HPCG_OPTS = -DHPCG_NO_OPENMP

CXXFLAGS = $(HPCG_DEFS) –std=c++11 ***–O3 -xhost -no-prec-div -no-prec-sqrt –ipo –fomit-frame-pointer -parallel***

Parameters contribute to a better performance in HPCG test are listed as follows:

–O3: optimize for maximum speed and enable more aggressive optimizations.

-xhost: generates instruction sets up to the highest that is supported by the compilation host.

-no-prec-div -no-prec-sqrt: reduces precision of floating-point divides and square root computations.

–ipo: permits inlining and other interprocedural optimizations among multiple source files.

–fomit-frame-pointer: uses EBP as a general-purpose register.

–parallel: tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.

2.3.3 Data Unification

There are three date types for integers in HPCG, *global_int_t* is used to define the indices of nodes in global sparse matrix, *local_int_t* is used to define the indices of nodes in local sparse matrix and *char* is used to define the number of non-zero values per row. By default, *global_int_t* is defined as *long long int* and *local_int_t* is default as *int*. This is because *int* is not enough to store the order of global matrix when np is huge in some supercomputers.

Considering that the processing speed of *int* is faster and the number of processor cores in our system is small. It's completely enough to store the order of global sparse matrix by *int*. So we change the integer types unified to *int* and set the preconditioning flag #define HPCG_NO_LONG_LONG.

2.4 Evaluation on Local Testing Platform

We evaluate HPCG 3.0 on our local testing platform with total 4 nodes (96 cores). Firstly, we analysis the basic information of program, then test performance of OpenMP and MPI respectively for the best configuration parameters. Especially we evaluate communication overhead of different cores. Finally, we test the effect of compiling parameters and data unification.

2.4.1 Basic Test

There are three parameters nx, ny and nz limited to be same, so that we use nx to represent the edge length of cube for each process. By change the nx in different number of processes, we get the performance information of HPCG as follows.

As Figure 10 shows, our local testing platform obtains highest performance, when $nx = 16$, approximately double than the others, and the performance keeps stable when $nx > 16$.

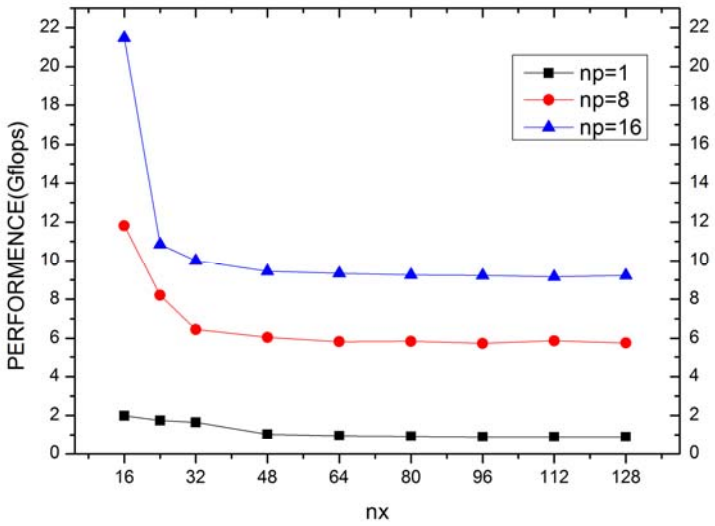


Figure 10 Performance of HPCG under different *nx*

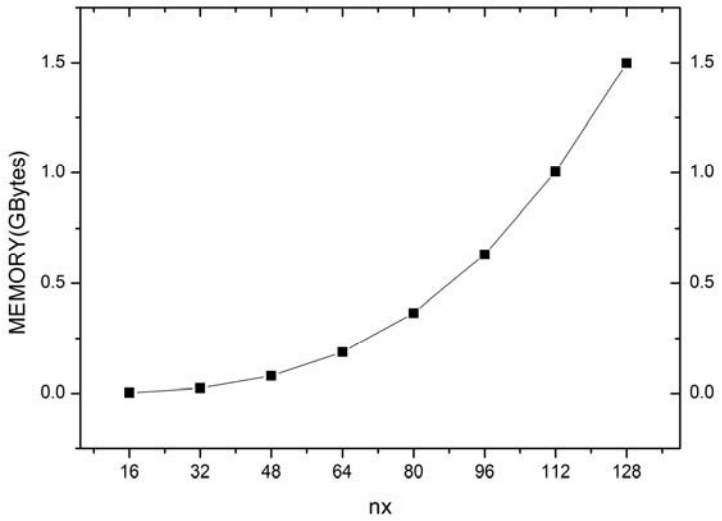


Figure 11 Memory of HPCG under different *nx*

Then we analysis memory information for each process.

According to the space complexity $O((nonzeros+5)\cdot nx\cdot ny\cdot nz)=O(nonzeros\cdot nx^3)$, the memory used will be cubic increasing with the nx. When $nx=16$, the memory usage is so small that it is possible to store all of the data in cache. There are the numerical statistical results as follows.

Table 9 Memory of HPCG under different *nx*

nx	16	24	32	48	64
memory(Gbytes)	0.002952	0.0098887	0.02342	0.07904	0.1874

The memory used of $nx=16$ is about 0.002952 GBytes, while the cache size of the processor is about 32768 Kbytes (equal to 0.03125 Gbytes). So the program data under $nx=16$ could be stored in cache that it makes the performance higher. Although, the size of problem could not be so small in practice.

2.4.2 OpenMP Test

We then test the effect of OpenMP in reference version under different nx with one process having 8 threads and the other has only one thread.

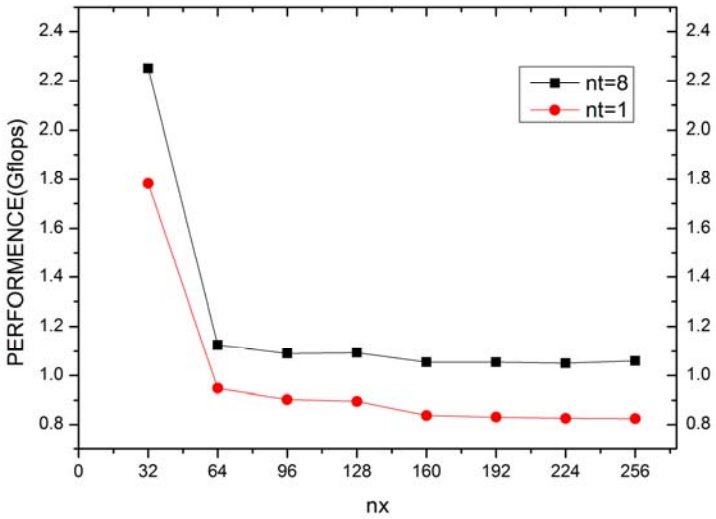


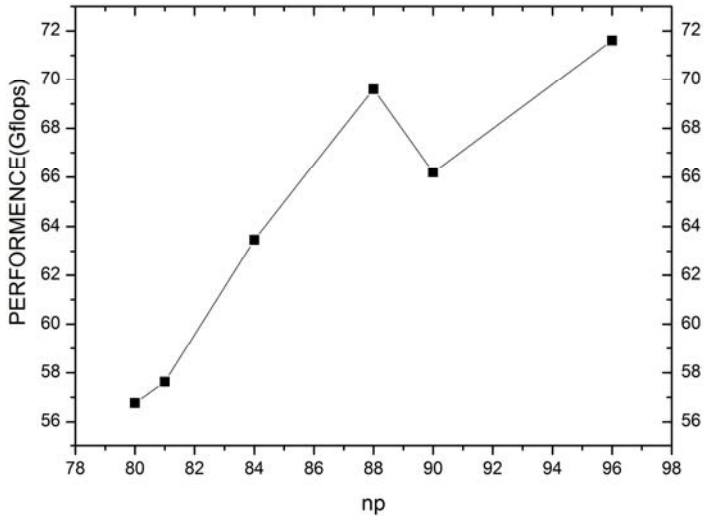
Figure 12 Performance of HPCG under different thread number

Where we find that more threads could maintain a stable higher performance and the speedup

ratio will be higher when used more memory.

2.4.3 MPI Test

We limit the $nx = 16$, then evaluate performance under different number of MPI processes. Considering that the communication overhead will increase along with the increasing of process number, and the limited number processors in our local testing platform. Here we test the performance with *np* from 80 to 96.



Notes: many parameter settings may lead runtime error such as $np = 87$

Figure 13 Performance of HPCG under different numbers of processes

The top 3 high performance are obtained under $np = 88$, $np = 90$ and $np = 96$. Then we analysis the communication overhead of these three settings. Considering the geometry distribution of processes in program (ipx, ipy, ipz).

$$\gamma_{88} = (2,4,11)$$
$$\gamma_{90} = (3,6,5)$$
$$\gamma_{96} = (4,4,6)$$

In Figure 8, the total area of adjacent faces is 2×16^2. In the same way, we could calculate that:

$$\delta_{88} = [(2-1) \times 4 \times 11 + 2 \times (4-1) \times 11 + 2 \times 4 \times (11-1)] \times 16^2$$
$$\delta_{90} = [(3-1) \times 6 \times 5 + 3 \times (6-1) \times 5 + 3 \times 6 \times (5-1)] \times 16^2$$
$$\delta_{96} = [(4-1) \times 4 \times 6 + 4 \times (4-1) \times 6 + 4 \times 4 \times (6-1)] \times 16^2$$

The total area of adjacent faces determines the communication overhead, and the ratio between $np = 88$, $np = 90$ and $np = 96$ is:

$$\delta_{88} : \delta_{90} : \delta_{96} = 2.15 : 2.3 : 2.\dot{3}$$

So the communication overhead of $np = 88$ is lower than $np = 90$ and $np = 96$ which means $np = 88$ also has a high performance with lower energy consumption in comparison to $np = 96$.

2.4.4 Optimization Test

Finally, we do the optimization test in different levels. At level 1, we use the compiling flags ***–O3*** and ***–xhost***; At level 2, add the compiling flags ***-no-prec-div*** and ***-no-prec-sqrt***; At level 3, add the compiling flags ***–ipo***, ***–fomit-frame-pointer*** and ***–parallel***. We test $np = 88$, $np = 96$ and find that the average performance of $np = 88$ is stable and even higher than $np = 96$.

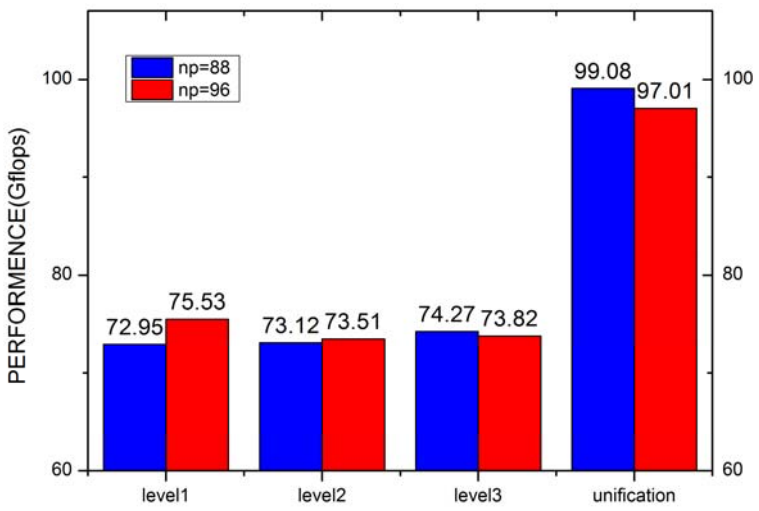


Figure 14 Optimized performance of HPCG

The final optimized performance obtained on our 4-nodes local testing platform is **101GFlops**, approximately **40%** performance improvement than the original version of HPCG application.

The HPCG results submission form is as follows:

HPCG	
HPCG version	3.0
System Spec	CPU: 2*Intel Xeon E5-2680v3(2.5 GHz, 12 cores) Memory: 128GB DDR4 Hard Disk: 2*300GB SSD IB Switch: Mellanox InfiniBand switch, 8 QSFP port HCA Card: 2*Mellanox ConnectX-4 HCA card, single port QSFP, FDR InfiniBand Cable: 4*InfiniBand FDR optical fiber cable, QSFP port
Compute Nodes	4*NF5280M4 cluster with Infiniband
OS	CentOS 6.7 release 64-bit
MPI	Intel MPI Library 5.1 Update 1
Compiler	ICC 16.0.1
Compiler Flags	–std=c++11 –O3 -xhost -no-prec-div -no-prec-sqrt –ipo – fomit-frame-pointer -parallel
Environment Variables	---
Turbo (ON/OFF)	OFF
	Results
Number of nodes	4
Number of cores	88
HPCG (base or optimized)	Optimized
HPCG result	101.456GFlops

3. MASNUM_WAM Test

MASNUM (Marine Science and Numerical modeling) is designed by LAG(Laboratory of Geophysical Fluid Dynamics) and FIO (First Institute of Oceanography) of SOA (State Oceanic Administration) of China. The model has been compared with WAM model in typical wind fields, such as the same grid wind, GFS wind, QuikSCAT BLN wind, NCEP re-anal wind, which brings about concerted results in general sea state. In practice, the model has been used in China seas for forecast and hindcast practices and calculate results consistent with filed measured data.

3.1 Basic Theory of MASNUM

In this part, we will show the basic mathematical equation in the MASNUM which leads us to better understand the program.

3.1.1 Wave Energy Spectrum Balance Equation

The Numerical Model of MASNUM WAVE, its wave energy spectrum balance equation in spherical coordinates is as follows.

$$\frac{\partial E(\vec{\mathrm{k}})}{\partial t}+\left(\frac{C_{g\lambda}+U_{\lambda}}{Rcos\varphi}\right)\frac{\partial E(\vec{\mathrm{k}})}{\partial \lambda}+\left(\frac{C_{g\varphi}+U_{\varphi}}{R}\right)\frac{\partial E(k\vec{\mathrm{k}})}{\partial \varphi}-\frac{\left(C_{g\varphi}+U_{\varphi}\right)tan\varphi E(k\vec{\mathrm{k}})}{R}$$
$$=S_{in}+S_{ds}+S_{bo}+S_{nl}+S_{cu}$$

Where **the left term**: $E=E(\vec{\mathrm{k}})$ is the wave-number spectrum, latitude φ and longitude λ; $U=\left(U_{\lambda},U_{\varphi}\right)$ is background current velocity; $C_g=\left(C_{g\lambda},C_{g\varphi}\right)$ represents the group velocity; $C_{g\lambda}$ represents the group velocity in the direction of latitude φ, $C_{g\varphi}$ represents the group velocity in the direction of longitude λ; U_{λ} represents the velocity in the direction of latitude φ; U_{φ} represents the velocity in the direction of longitude λ.

the right term: S_{in} represents the wind input source function, S_{ds} represents the wave breaking dissipation function, S_{bo} represents the bottom friction dissipation function, S_{nl} represents the wave-wave weak nonlinear interaction function and S_{cu} represents the wave-current interaction.

3.1.1.1 Wind Input Source Function

The wind input into the wave energy source function could be description as follows:

$$S_{in}(\mathrm{E})=\alpha+\beta\mathrm{k}(\vec{\mathrm{K}})$$

Where, $\alpha(\mathrm{K})=80\left(\frac{\rho_0}{\rho_w}\right)^2\frac{{U_*}^4\sigma}{g^2K^2}cos^4(\theta_1-\theta)\cdot H[\cos(\theta_1-\theta)].\frac{\rho_0}{\rho_w}=1.25*10^{-3}$; U_* is the friction velocity; θ is the direction of wind (measured anticlockwise relative to true east). H is the Heaviside function; towed coefficient C_d is be written as $C_d=(\frac{U_*}{W})^2=(0.8+0.065\mathrm{W})*10^{-3}$, where W is the velocity at 10 meters above the sea level;

Coefficient $\beta=0.25\frac{\rho_0}{\rho_w}\sigma[28\frac{U_*}{C}\cos(\theta_1-\theta)-1]\cdot H\left[28\frac{U_*}{C}\cos(\theta_1-\theta)-1\right]$, where C is wave number; $\mathrm{C}=\frac{\sigma}{K}$.

3.1.1.2 Wave Breaking Dissipation Function

The Wave breaking dissipation function could be description as follows:

$$S_{ds} = -d_1\varpi\left(\frac{\sigma}{\varpi}\right)^2\left(\frac{\vartheta}{\vartheta_{PM}}\right)^{\frac{1}{2}}\cdot\exp\left\{-d_2\left(1-\varepsilon^2\frac{\vartheta_{PM}}{\vartheta}\right)\right\}\mathrm{E}(\vec{\mathrm{K}})$$

Where:

$$\varpi = (\iint E(\vec{\mathrm{K}})\sigma^{-1}d\vec{\mathrm{K}}\,/\,\vec{\mathrm{E}})^{-1},\varpi = g\vec{\mathrm{k}},$$

$$\vec{\mathrm{E}} = \iint E(\vec{\mathrm{K}})\mathrm{d}\vec{\mathrm{k}},\vartheta = \vec{\mathrm{E}}\,\varpi^4 g^{-2},$$

$d_1 = 1.32*10^{-4}$, $\mathrm{d}_2 = 2.61$, $\vartheta_{PM} = 3.20*10^{-3}$, ε is the spectral width.

3.1.1.3 Bottom Friction Dissipation Function

The Bottom friction dissipation function could be description as follows:

$$S_{bo}(E) = -C_b\frac{8K}{\sinh 2Kd}\varpi\,\vec{\mathrm{E}}^{1/2}E(\vec{\mathrm{K}})$$

where fitting coefficient $C_b = 2.5*10^{-3}$

3.1.1.4 Wave-wave Weak Nonlinear Interaction Function

The wave-wave weak nonlinear interaction function could be description as follows:

$$S_{nl}(E) = \mathrm{R}(\mathrm{Kd})\sigma\iint A(\vec{\mathrm{K}1},\vec{\mathrm{K}2},\vec{\mathrm{K}3},\vec{\mathrm{K}})\cdot[\mathrm{N}_1N_2(N_3+N)-N_3N(N_1+N_2)]\cdot\delta(\vec{\mathrm{K}1}+\vec{\mathrm{K}2}$$

$$-\vec{\mathrm{K}3}-\vec{\mathrm{K}})\delta(\sigma_1+\sigma_2-\sigma_3-\sigma)d\vec{\mathrm{K}1}\,\mathrm{d}\vec{\mathrm{K}2}\,\mathrm{d}\vec{\mathrm{K}3}$$

where $\mathrm{A}(\vec{\mathrm{K}1},\vec{\mathrm{K}2},\vec{\mathrm{K}3},\vec{\mathrm{K}})$ is the wave-wave interaction function.

3.1.1.5 Wave-current Interaction Function

The wave-current interaction function could be description as follows:

$$S_{cu}(E) = -\left\{\left[\frac{C_g}{C}(1+cos^2\theta_1)-\frac{1}{2}\right]\frac{\partial U_x}{\partial x}+\frac{C_g}{C}sin\theta_1 cos\theta_1\left(\frac{\partial U_x}{\partial y}+\frac{\partial U_y}{\partial x}\right)\right.$$

$$\left.+\left[\frac{C_g}{C}(1+sin^2\theta_1)-\frac{1}{2}\right]\cdot\frac{\partial U_y}{\partial y}\right\}E(\vec{\mathrm{K}})$$

3.1.2 Complicated Characteristic Equations

The complicated characteristic equations in spherical coordinates may be written,

$$\frac{d\lambda}{dt} = \frac{C_{g\lambda}+U_\lambda}{Rcos\phi}$$

$$\frac{d\phi}{dt} = \frac{C_{g\emptyset}+U_\phi}{R}$$

When the wave energy spreads as above functions, the variation law of the mode and angle of the wave-number is as follows.

$$\frac{\partial K}{\partial t}+\frac{C_{g\lambda}+U_\lambda}{Rcos\phi}\frac{\partial K}{\partial t}+\frac{C_{g\emptyset}+U_\phi}{R}\frac{\partial K}{\partial\emptyset}+\left(U_\lambda sin\theta_1-U_\phi cos\theta_1\right)tan\emptyset R^{-1}Kcos\theta_1$$

$$= -\frac{cos\theta_1}{Rcos\emptyset}\left(\frac{\partial\sigma}{\partial D}\frac{\partial D}{\partial\lambda}+Kcos\theta_1\frac{\partial U_\lambda}{\partial\lambda}+Ksin\theta_1\frac{\partial U_\phi}{\partial\lambda}\right)-\frac{sin\theta_1}{R}\left(\frac{\partial\sigma}{\partial D}\frac{\partial D}{\partial\phi}\right.$$

$$\left.+Kcos\theta_1\frac{\partial U_\lambda}{\partial\phi}+Ksin\theta_1\frac{\partial U_\phi}{\partial\emptyset}\right.$$

$$\frac{\partial\theta_1}{\partial t}+\frac{C_{g\lambda}+U_\lambda}{Rcos\phi}\frac{\partial\theta_1}{\partial\lambda}+\frac{C_{g\emptyset}+U_\phi}{R}\frac{\partial\theta_1}{\partial\emptyset}+\left(U_\lambda cos\theta_1+U_\phi sin\theta_1\right)tan\emptyset R^{-1}cos\theta_1$$
$$=-\frac{sin\theta_1}{Rcos\emptyset}\left(\frac{1}{K}\frac{\partial\sigma}{\partial D}\frac{\partial D}{\partial\lambda}+cos\theta_1\frac{\partial U_\lambda}{\partial\lambda}+sin\theta_1\frac{\partial U_\phi}{\partial\lambda}\right)-\frac{cos\theta_1}{R}\left(\frac{1}{K}\frac{\partial\sigma}{\partial D}\frac{\partial D}{\partial\phi}\right.$$
$$\left.+cos\theta_1\frac{\partial U_\lambda}{\partial\phi}+sin\theta_1\frac{\partial U_\phi}{\partial\emptyset}\right)$$

where θ_1 is wave direction (measured anticlockwise relative to true east). Above two equations, the modulation of background current to wave evolution and the refraction of waves propagating along great circles are included; $(k\cos\theta, k\sin\theta)$ and $(\cos\theta, \sin\theta)$ are the wave vector and its unit vector; $\omega(K, D)$ is dispersion relation.

3.2 Program Analysis of MASNUM

In this part, we will analyze the program based on the original code which leads us to understand how the program runs and makes it easy to know where and how to optimize.

3.2.1 Module of MASNUM Wave Model Based on MPI

After being parallelized, the whole program has changed a lot. The relationship of modules could be seen from the Figure 15, which shows that the module wammp_mod is the core module because it controls the flow of the program.

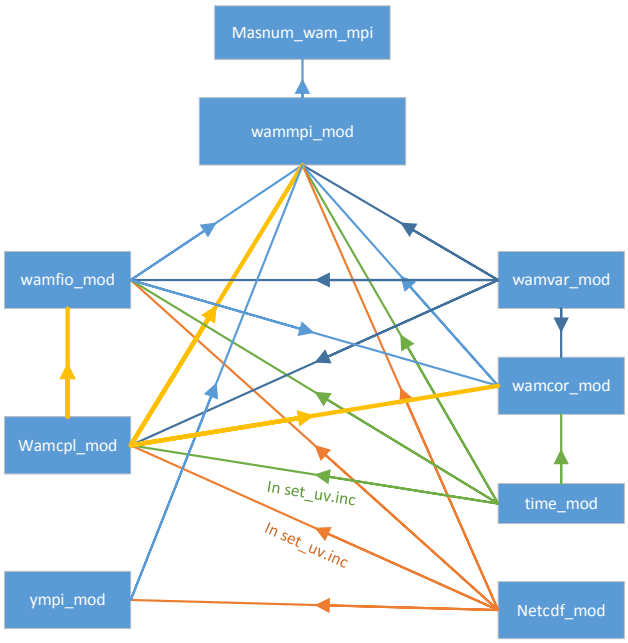


Figure 15 The relational graph for each module

In order to introduce the modules in Figure 15, we use different color line to make the relationship of the module more clearly (shown in Table 10). The whole program is arranged by 6 remaining-modules (including 4 main modules and 2 tool modules), 2 mpi-modules and 1 top-module(masnum_wam_mpi), which is quite different from the previous version as shown in the given doc. The main program is launched in the masnum_wam_mpi module.

Table 10 shows the description of every core module. We divide it into two parts according to whether it has a direct relationship with MPI. And the upper part is the original code and the below is added into the program for parallelization.

Table 10 The description of modules

Module	Desciptions
time_mod	Used to deal with the time
netcdf_mod	Used to input/output data through NetCDF format
wamvar_mod	Include all the global variables used in this model
wamfio_mod	Subroutines for I/O data or model results
wamcpl_mod	Subroutines for coupling w/current model
wamcor_mod	The core subroutines of this model
MASNUM_WAM_MPI	Launch the whole program
WAMMPI_MOD	Call different subroutines and loop
YMPI_MOD	Split the task and control inter process communication

3.2.2 The Flow of MASNUM Wave Model Based on MPI

Before we get the flowchart of the main procedures, some illustrations of the program structure are needed. In the numerical scheme of the model, it could totally be divided into two parts. One part is initialization and the other part is calculation.

3.2.2.1 Initialization

Module manum_wam_mpi uses the public function wammpi_init in wammpi_mod to initialize the main parts (including mpi initialization and mypebox part), update matrix ee (Wave spectrum), call the wamfio_mod_init to initialize the fundamental data and finally set the time-related variables, rather than module wamcor_mod uses the public function precom to call function wamfio_mod_init in module wamfio_mod.

(1) Parameter Intialization

First part, the program read the control parameters. The control parameters are as follows.

Table 11 The control parameters

Symbols	**Meaning**
DATA_PATH	The path for file of topography
WIND_PATH	The path for wind data
TITLE	Symbol for model output
CISTIME	Start time for integration
CIETIME	End time for integration
COOLS_DAYS	The time(days) for cool start
DELTTM	Length of integral time step, in minutes.
WNDFREQ	The frequency of wind data (hours).
WNDTYPE	The wind type:0 for wind in the same grid with model.1 for GFS wind.2 for QuikSCAT BLN wind.3 for NCEP re-anal wind.
OUTFLAG	The method of output: 0 One file for each run;1 One file each year;2 One file every month;3 One file for every day.
WIOFREQ	The output frequency for wave results (hour).
CIOFREQ	The output frequency for current coef.s (hour).
RSTFREQ	The output frequency for model restart (hour).

Second part, global variables initialization.

In wamvar_mod_init, the program initializes global variables and allocates the memory to global arrays such as e, ee, ea, wx, wy. In settopog, the program reads the topography file wamyyz.nc. In nlweight, the program calculates the wave-wave weak nonlinear interaction function. In set_uv0, the program sets the velocity of wind to ZERO. And the related settings

including longitude, latitude, grid spacing and etc. are also contained in this part.

(2) Pebox Initialization

Pebox Initialization is the part split the matrix waiting for the program to calculate and output. Table 12 shows the 17 fundamental parameters of pebox. Each process has a unique pebox to record the information of the distributed area. And only when the distribution strategy is reasonable, it is good for load balance. Load balance is one of the restricted factors in MPI. And in our test, we find that this part could be optimized and the program performs better. Some parameters (such as i1,i2,j1,j2) in pebox are essential for IPC(inter process communication). That is the reason why we give the parameters in Table 17 because it records the number of adjacent distributed matrix.

Table 12 Data structure of pebox and their meanings

Name	Description	Name	Description
myid	index of the current process	np	the sum of mask in the area from i1 to i2 and from j1 to j2
i1	begin position of the line	ei1	the begin position of the matrix minus halosize for the inter process communication
i2	end position of the line	ei2	the end position of the matrix plus halosize for the inter process communication
j1	begin position of the column	ej1	the begin position of the matrix minus halosize for the inter process communication
j2	end position of the column	ej2	the end position of the matrix plus halosize for the inter process communication
dn	id of up	halo	the size in spatial partition.
up	id of down	isize	the size of e in the direction of i
nr	the number of the right pebox	jsize	the size of e in the direction of j
nl	the number of the left pebox		

Table 13 Adjacent matrix close to the mypebox

Name	Description	Name	Description
mypebox%left(i,1)	the group of the index of the left process number and it has the number of nl	mypebox%right(i,1)	the group of the index of the right process number and it has the number of nr
mypebox%left(i,2)	the boundary index of the left close mypebox matrix	mypebox%right(i,2)	the boundary of the right close mypebox matrix
mypebox%left(i,3)		mypebox%right(i,3)	
mypebox%left(i,4)	the length size and the width size of the transfer matrix	mypebox%right(i,4)	the length size and the width size of the transfer matrix
mypebox%left(i,5)		mypebox%right(i,5)	

With the data structure of mypebox, the matrix is divided to distribute the task of calculation and output. The exhibition of load balance could be seen as the Figure 16. And the arrow shows the direction of data transfer. The number of grids in the whole matrix is equal to the number of processors. The model shows that the original load balance strategy (under 24 processes) and the outcomes of task distribution are stored in pebox.

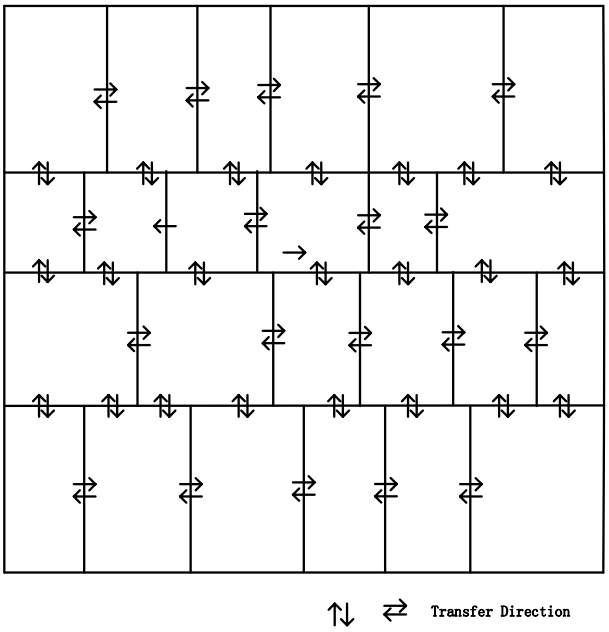


Figure 16 Data transfer among adjacent matrix in 4*6 MATRIX

3.2.2.2 Calculation

Module manum_wam_mpi uses the public function readwi_mpi in wammpi_mod to calculate. In addition, this part also reads the wind data. The core function is given as follows.

Function	Discription
get_wind	According to the control parameters wndtype, read the corresponding wind data and call bilinear_4points to realize bilinear interpolation for the wind data.
propagat	The function is the most significant function in the whole program because the running time of it is longest. The function is to solve the spread of wave equation.
implsch	The function is the second significant function in the whole program because the running time of it is the second longest. The function is to solve the local changes caused by the source functions.
mean1	To solve the characteristic of the wave equations.
output	Export the netcdf file according to the outflag. The output files include the statistic fields of ocean surface wave and the wave-current mixing coefficients.

3.2.2.3 The flowchart of the main program

With the above explanation, we could understand how the program run in some way. And Figure 17 shows that the core functions and the flow of the whole program.

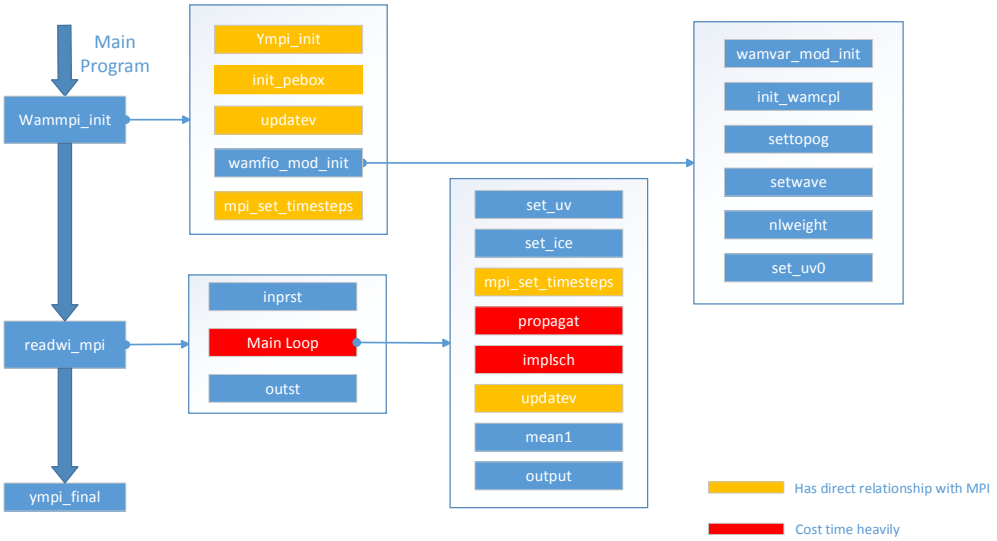


Figure 17 The flowchart of the main program

3.2.3 Performance Overview

According to *Performance Characterization and Efficient Parallelization of MASNUM Wave Model*[1], it is the long latency of floating-point instructions on multiplication and division, the performance of disk accessing and SIMD(single instruction multiply data) that results in the low effective of floating-point performance. IPC (inter process communication) performance and the bandwidth of I/O affect the performance. In addition, load balance is restriction factor.

3.3 Optimization and Evaluation on Local Testing Platform

Settings of arguments in MASNUM's makefile are shown in Table 14. We use NetCDF-4.4 with NetCDF-Fortran-4.2 for NetCDF file support. As for compiler, we select *ifort*. In our MASNUM's test, we do most of our exams on Exp1.

And more importantly, for better explaining the key problem, we decide to experiment in one node (24 CPU cores) in the part of 3.3.1~3.3.6. And duration of calculation of the program is from 2009:01:01 to 2009:01:02.

Table 14 MASNUM Configure

F77	mpiifort
LIBRARY	-L$(NETCDF_PATH)/lib -lnetcdf -lnetcdff -L/usr/local/lib

3.3.1 Initial Run

We have tested the speed of the program using different number of processes. The results are shown in Figure 18, which indicates that the larger the number of processes is, the faster the program runs. It is reported that programs, especially high performance programs, run slowly when hyper-thread technology is enabled. However, our tests on MASNUM show that hyper-thread technology leads to higher performance.

[1] Zhang ZhiyuanYufeng, Liu Li, and Yang GuangwenZhou. (2015). Performance Characterization and Efficient Parallelization of MASNUM Wave Model. Journal of Computer Research and Development, 851-860.

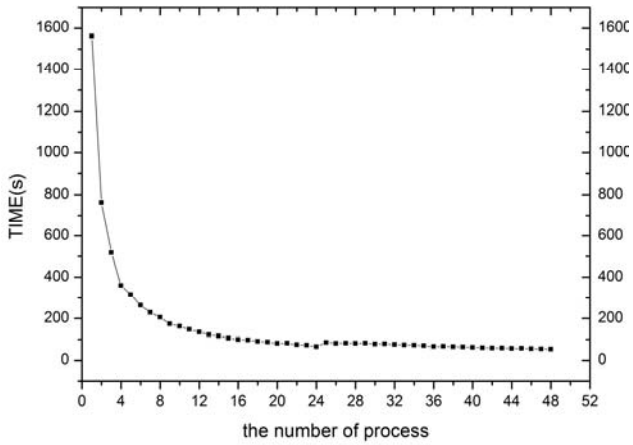


Figure 18 Relationship between execution time and process number

In order to improve throughput, modern CPUs implements concurrency of floating-point instructions by using pipeline architecture. One floating-point instruction can be finished in 1 cycle. However, the cost of floating-point division is 10-100 cycles. In addition, modern CPUs cannot hide the latency of floating-point division by applying pipeline architecture. Thus, it is floating-point division that affects the performance of the program. Enabling hyper-thread reduces the average cost of floating-point division as programs can execute multiple computing operations in parallel. Thus, maximize the number of worker threads to achieve higher performance is a good choice.

3.3.2 Optimization Options

Table 15 Compiler Options

Options	Explanation
-xHost	Optimize and tune for the compiling CPU
-O3	Enable more aggressive loop transformations. Note that the O3 optimizations may not cause higher performance.

-xHost option enables all instructions sets supported by the compiling CPU. To be more specific, *-xHost* option causes compiler applying auto-vectorization to the program. It leads to performance improvement, as can be seen in Figure 19.

–O3 option is suitable for the program that has heavily loops. Applying *–O3* to the program reduces time elapsed (from 1:09.19 to 1:05.76). Also, we tested the combination of *–O3* and *–xHost*. Interestingly, the time consumption of it is same as the program that only applying *–O3* option.

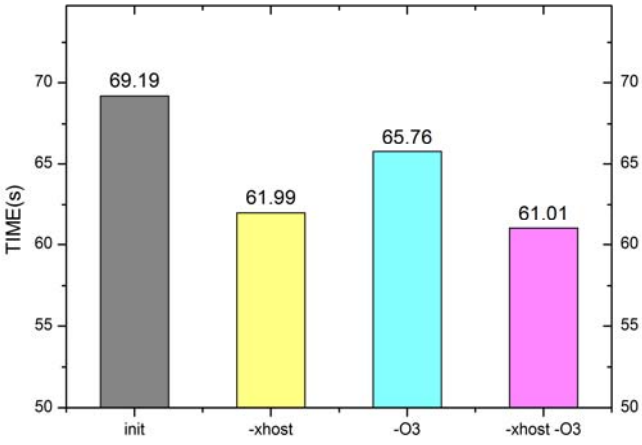


Figure 19 The runtime of different compiler parameter

3.3.3 Core Function of MASNUM

Top Hotspots

This section lists the most active functions in your application. overall application performance.

Function	Module	CPU Time
wamcor_mod_mp_propagat_	masnum.wam.mpi	676.202s
wamcor_mod_mp_implsch_	masnum.wam.mpi	322.656s
pmpi_waitall_	libmpifort.so.12	190.162s
pmpi_bcast_	libmpifort.so.12	122.802s
pmpi_sendrecv_	libmpifort.so.12	60.485s
[Others]	N/A*	213.220s

Figure 20 The screen snapshot from the original program

From the Figure 20, we could find that the function propagate is the hottest function and implsch is the second. And there are two solutions to improve the performance.

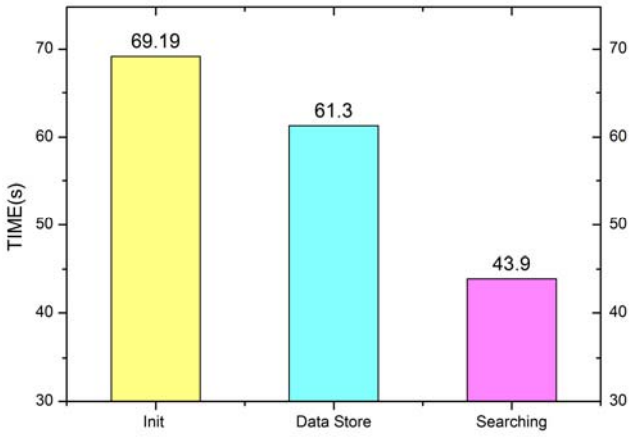


Figure 21 The two core algorithms optimization

(1) Reduce the Amount of Calculation Optimization

The pseudocode is shown in Algorithm 1. Because the datum in PART A in loop i1 and i2, and PART B in loop i1,i2 and j are the same. We store the data waiting for computing and visiting into a matrix. In this way, the amount of calculation in 4 loops have a significant compact on the performance of the program.

Algorithm 1 Reduce the amount of calculation

```
Do Loop i1
Do Loop 12
  ...
  Do Loop j
  If (Jflag==0)
     PART A:
     Data has no relationship with Loop i1, Loop i2 &&
     Store those Data into Matrix
  Else
     ...
  Do loop k
     If (Kflag==0)
          PART B:
          Data has no relationship with Loop i1, Loop i2, Loop j &&
          Store those Data into Matrix
       Else
```

```
            ...
    End Loop k
    End Loop j
End Loop i2
End Loop i1
```

(2) Searching algorithm Optimization

The source code of the searching algorithm in the inner loop is as follows:

```
1  do iii = ixs, ixl-1
2      if (xx >= x(iii) .and. xx <= x(iii+1))then
3          ixx = iii; exit
4      endif
5  enddo
```

In the program, x is the array of longitude (rang from 0~360) in ascending order. And due to the integral formula, xx increases up slowly. After analysis the law of the array of x and xx, we could modify the algorithm as the following:

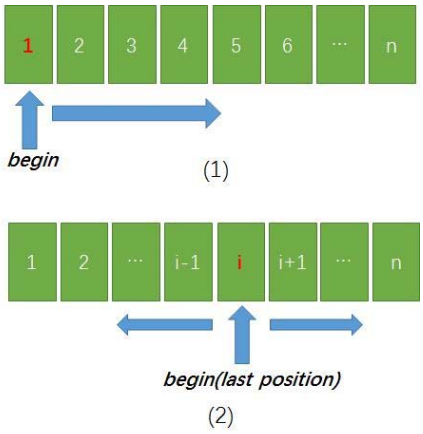


Figure 22 The model of init and optimized searching algorithm

Figure 22 shows that how could we find the right position in use of the result of last loop. Figure 22-(1) shows that the begin position is the initial position. Thus, the previous algorithm complexity is O(N). Figure 22-(2) shows that the begin position is the last position, and due to variation of x and xx is slow, the algorithm complexity tends to O(1). Although the algorithm complexity of the binary search is O(logN), the law of xx and x makes my algorithm works very well, even better than the binary search.

(3) Hash Function on Implsch and Propagat

After analyzing the core function, we find that there exits a large amount of trigonometric functions. Considering the computation cost of them, it is wiser to build a hash table to store the computed trigonometric functions. Thus, macro definition hash table and common division are needed to implement this optimization.

3.3.4 Improve the Data Locality

The array in Fortran is organized in column major order. It would be faster by column first and then by line, if visiting 2-D array. So, it is better to make most of the array visiting to visit the lower dimension first. So we change loops in the core function such as the file propagat, impluxsh, mean1, readwi_mpi. The data stored in a cache might be the result of an earlier computation.

In this way, the cache hit ratio will increase and less cache misses occur, which reduces the time.

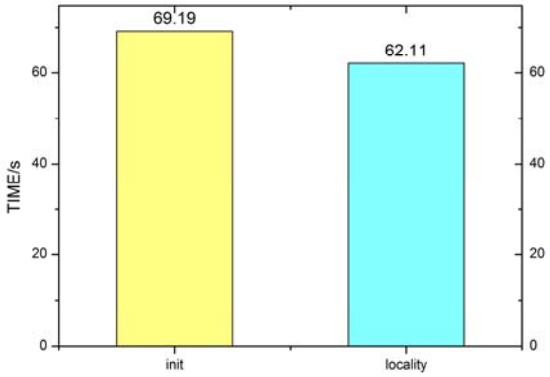


Figure 23 Init is visited by line and locality is visited by column

3.3.5 Load Balance Optimization

Taking land-sea distribution into consideration, initial program takes such kind of strategy to divide the matrix. First, divide it into several lines and then into several columns. x_proc is the processor number of the longitude and y_proc is in the latitude. The algorithm first distributes all of the grid into x_proc lines, and the sum of grid number is not less than the average load balance; then the algorithm distributes all of the grid in each line into y_proc columns. The distribution algorithm makes the load of each processors by line and by column imbalanced and decrease. The load of last column/line is far less than the average load. The distribution result is similar to Figure 16.

Algorithm 2 Original algorithm

1) Suppose N_{total} is the total grid number of the matrix. x_proc is the processor number of the longitude and y_proc is in the latitude.
2) From the serial processor number i=0 to x_proc-2. $N_{avg} = N_{total}/(x_{proc} - i)$
3) Distributes N_i to processor i, N_i is not less than N_{avg}
4) Distributes the remaining to the number of x_proc-1.
5) For each columns(x_proc-1 columns), from the serial processor number j=0 to j_proc-2. $N_{avg} = N_{total}/(y\_proc - i)$
6) Distributes N_j to processor j, N_j is not less than N_{avg}
7) Distributes the remaining to the number of x_proc-1.

There are two ways to improve:

(1) x_proc must be not less than y_pro

After analyze how the mpi communication, we make a conclusion that the communication in latitude would be more easily and cost less time for the following reasons. From the model of Figure 16, it is easier for the grid to communicate with grids close to it in the direction of longitude than the direction of latitude because the data transferring is no-blocking in the direction of longitude and the data transferring is blocking in the direction of latitude. Thus it is wiser to distribute more in the direction of latitude.

(2) change the N_{avg} when if-condition

Thus, we modify the distribution strategy in step 2,3) and 5,6). It is wise to make every N_i is close to N_{avg}. We change the condition of if. If N_i is less than N_{avg}, N_{avg} is $N_{already-i}/(x_{already-i} - 1)$, $N_{already-i}$ represents total number of distributed grid so far. $x_{already-i}$ is the index of matrix waiting for distributing. The distribution strategy of latitude is the same as that of longitude.

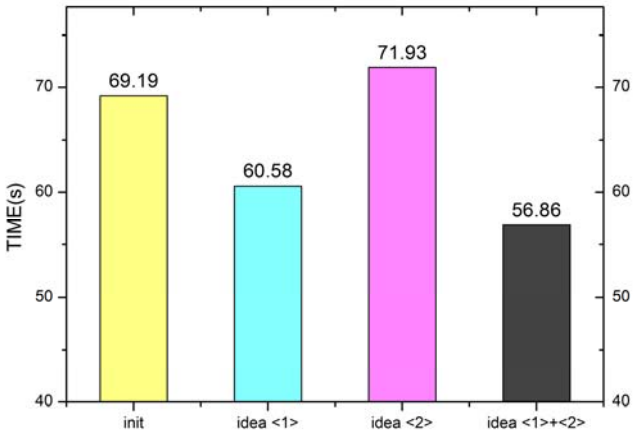


Figure 24 The two ways to optimization

We could find there is a abnormal data from Figure 24. Idea (2) ought to cost less time but in fact it costs more here. But in the combination of idea (1) and idea (2), idea (2) works not bad. The possible reason here is that in the condition of given line size and column size (it has no relationship with their size relationship), it results in more computing cost or more processor communication cost. In the following extreme condition, it will occur. The 0~i, although the initial strategy make the load of grid 0~i bigger than total average, but they are almost the same exception the last one. But my load distribution strategy make the load of different grids fluctuate which will result in more imbalanced load distribution.

3.3.6 Multi-nodes on MASNUM

We could get a conclusion that with the initial I/O codes, the best choice of nodes number is about 3. And our optimization works well.

However, with the number of nodes increase, the I/O cost time has greatly increased. We have test the communication bandwidth of MPI, the result shows that the peak speed is about 30Gbps/s. But the peak speed of shared file system is about 300Mbps/s. And I/O is confirmed to be the most essential restriction when we want to run our program in multi-nodes.

The reasons why we don not use hyper-threading technology in multi-nodes is for the following reason. The reason is that when we hyper-threading does not mean there are real two cores in one CPU core. The main function of hyper-threading is to increase the number of independent instructions in the pipeline; it takes advantage of superscalar architecture, in which multiple instructions operate on separate data in parallel. But the more processors we use, the slower I/O we would get. Since there are enough CPU cores waiting for us to use, if we want 48 process, just let it run in two machine nodes. The experiment shows that it is worthy.

Because every time the program wants to write the netcdf fill in order, the more process we have, the longer time we have to wait which results in the more cost time.

Table 16 The final result of different number of nodes

Nodes	hpc0	hpc1	hpc2	hpc3	Total	Initial Time(s)	Optimization Time(s)	Speed up
4	24	24	24	24	96	30.58	23.12	34.21%
3	24	24	24		72	31.01	21.83	29.60%
2	24	24			48	36.62	25.73	29.74%
1	24				24	69.19	37.11	58.49%
1	48				48	52.69	28.72	

Due to the reason, we have thought out some ideas to deal with the slow I/O problem. By analyzing the concrete implementation of data output procedure in MASNUM model, we find that the reason of low I/O bandwidth is the random discrete disk access which is large and low efficiency lead by the data output. The origin version of this model output in queue, shown in the following algorithm. Because every output netcdf file are shared by all processes and output the data into file in orders, every process have to wait the former processor, which have been allocated a number in the initialization progress.

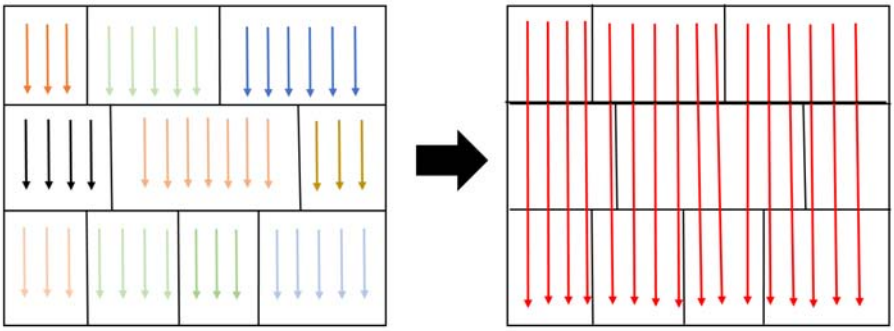


Figure 25 model for the date writing into netcdf

After the algorithm above, each process outputs its local data into the file according to the priority. However, it will result in plenty of random accessing in the disk. Unlike the memory accessing, the random disk accessing is far bigger than the sequential disk accessing. Because of it, we decide to run our own algorithm to output the file. First we gather all of the data from all processors, and then reorganize the data the data. Later, we write the data into the file in sequential accessing which is obviously speed up the I/O. Our algorithm is as follows. We let root process be the server and other process be the client.

Server Side	Client Side
for id = each client's ***id***	
blocked receive ***length*** from ***id***	***data*** = **compress**(***matrix***)
blocked receive ***data*** from ***id***	unbloked send ***length*** to ***server***
insert **decompress**(***data***) into ***matrix***	unblocked send ***data*** to ***server***
end for	

In this way, the disk accessing is more quick than before for the disk accessing follows the law of the sequential order.

3.3.7 Data Alignment

Unaligned access leads to ineffective load and store operations. Thus, we align the data for better performance by using pragmas and directives. There are 58 unaligned multi-demission arrays. MASNUM accesses these data frequently. We tested the program that applied data alignment optimization on our local testing platform. The data alignment has a positive effect on the program, although the effect is not obvious.

3.4 Result & Performance

All of the following result are running under the 48 processors in 1 node. All of the above optimization is confirmed to be correct. And I would to give a screenshot here to make it more

realizable.

```
[wlm@buaahpc0 exp1]$ ls
compare_exp1      Handle_err.f90  pac_ncep_wav_20090228_stardard.nc
compare_exp1.f90  Handle_err.o
compare_exp1.o    makefile
[wlm@buaahpc0 exp1]$ ./compare_exp1
 compare HS between ../../exp/exp1/pac_ncep_wav_20090228.nc and
 ./pac_ncep_wav_20090228_stardard.nc
 Step 1: Open file ../../exp/exp1/pac_ncep_wav_20090228.nc
 Step 1 Success
 Step 2: Open file ./pac_ncep_wav_20090228_stardard.nc
 Step 2 Success
 Step 3
 Step 3.1 Compare missing_value
 Step 3.1 Success
 Step 3.2 Compare scale_factor
 Step 3.2 Success
 Step 3.3 Compare HS
 Step 3.3 Success
 Compare Success
```

```
[wlm@buaahpc0 exp2]$ ./compare_exp2
 compare HS between ../../exp/exp2/global_ncep_wav_20090630.nc and
 ./global_ncep_wav_20090630_stardard.nc
 Step 1: Open file ../../exp/exp2/global_ncep_wav_20090630.nc
 Step 1 Success
 Step 2: Open file ./global_ncep_wav_20090630_stardard.nc
 Step 2 Success
 Step 3
 Step 3.1 Compare missing_value
 Step 3.1 Success
 Step 3.2 Compare scale_factor
 Step 3.2 Success
 Step 3.3 Compare HS
 Step 3.3 Success
 Compare Success
```

Figure 26 Validation of exp1 and exp2 (optimized program)

3.4.1 Top Hotspots

Top Hotspots

This section lists the most active functions in your application.
overall application performance.

Function	Module	CPU Time
implsch	masnum.wam.mpi	416.050s
propagat	masnum.wam.mpi	304.316s
pmpi_waitall_	libmpifort.so.12	139.941s
pmpi_recv_	libmpifort.so.12	81.250s
pmpi_sendrecv_	libmpifort.so.12	64.134s

Figure 27 The screen snapshot from the optimized program

Compared with the former graph, the implsch seems cost longer time. But actual it is not. Because the CPU time here is the total time of all CPUs. And we could get a conclusion that what we do in the porpagat is excellent. All of the other part of hotspots cost less CPU time.

3.4.2 CPU Usage

From Figure 28, we could see that the average CPU usage are good which is quite near the ideal stage.

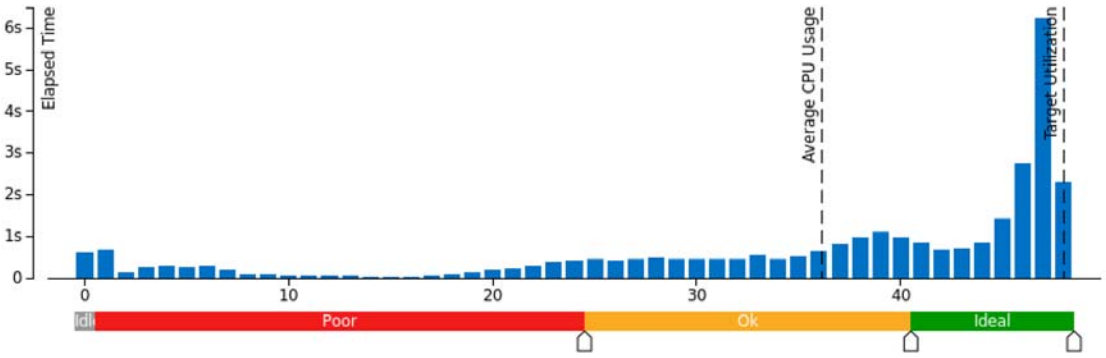


Figure 28 Simultaneously Utilized Logical CPUs

From Figure 29, we could know that at the two ends, what CPU does here is for MPI communication. The left end means that the root process sends the netcdf data that it reads, and right end is to waiting time. In the waiting process, each process writes their local data one by one. That is the reason why the time of mpi communication is so long and even CPU's spinning occurs.

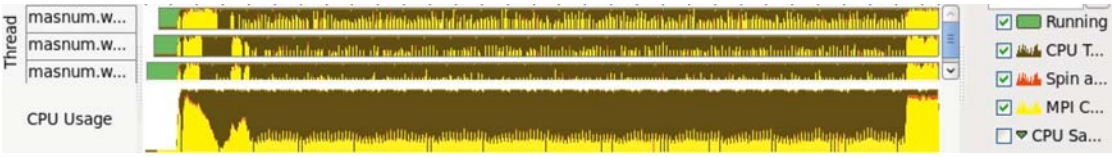


Figure 29 CPU usage

4. Parallel Design and Optimization of DNN on CPU+MIC platform

4.1 Algorithm and Program Analysis

4.1.1 Brief Introduction of DNN

Deep Neural Network (DNN) is an Artificial Neural Network (ANN) with multiple hidden layers of units between the input and output layers. Usually, the algorithm includes three major parts: the forward calculation, the error back propagation, and update of weight and bias. Similar to shallow ANN, DNN can model complex non-linear relationships. As one of typical algorithms for deep learning, DNN has been successfully applied in many domains in recent years, such as speech recognition and image recognition.

4.1.2 Analysis of the Original DNN Program

The DNN used in ASC16 consists of eight layers, including an input layer, six hidden layers and an output layer. The detailed network structure is shown in Figure 30 .

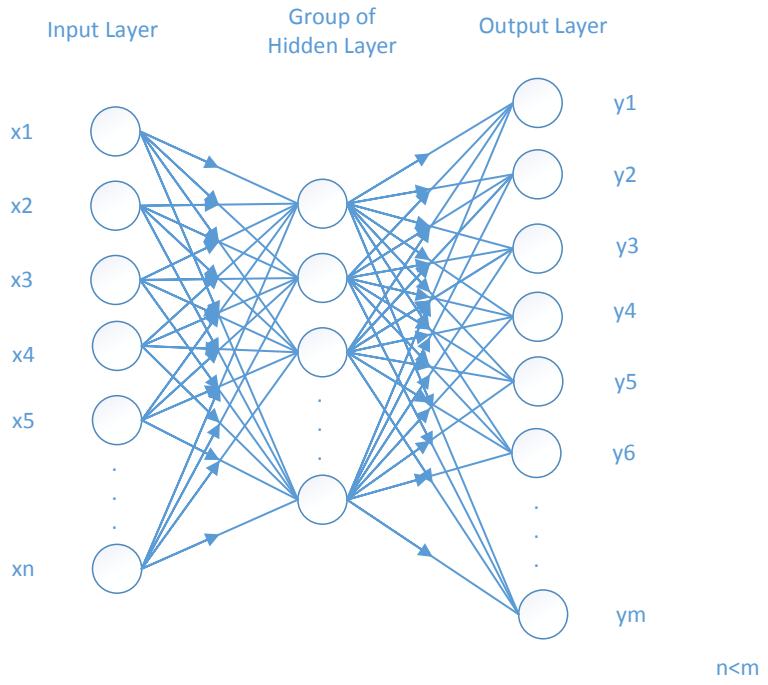


Figure 30 Detailed network structure

In order to make our description clear, here we simplify the representation of the parameters in *nodeArg*, as listed in the following table.

Table 17 Simplified representation of parameters in *nodeArg*

Original	Simplified representation	Meaning
nodeArg.numN	numN	Size of minibatch
nodeArg.dnnLayerNum	LayerNum	layer numbers
nodeArg.d_X	X	Input data of the input layer
nodeArg.d_T	T	Target label of input, used in backward
nodeArg.d_Y[i]	Y[i]	The neurons state of the layer i+1 (hidden layers and output layer, 0<=i<NUM_LAYERS)
nodeArg.d_W[i]	W[i]	The weight matrix of layer i (0<=i< NUM_LAYERS)
nodeArg.d_B[i]	B[i]	The bias of the Neurons on layer i+1 (hidden layers and output layer, 0<=i<NUM_LAYERS)
nodeArg.d_E[i]	E[i]	The training error of layer i+1 (hidden layers and output layer, 0<=i<NUM_LAYERS)
nodeArg.d_Wdelta[i]	Wdelta[i]	The weight delta matrix of layer i (0<=i<NUM_LAYERS)
nodeArg.d_Bdelta[i]	Bdelta[i]	The bias delta of the Neurons on layer i (hidden layers and output layer, 0<=i<NUM_LAYERS)

4.1.2.1 Analysis of Forward

(1) Basic idea

During forward calculation, DNN mainly use the data reading from the data file, the bias of the hidden layers and output layer, and the weight matrix to get the output of the hidden layers and output layer in order.

(2) Algorithm

The algorithm of Forward can be described as follows:

for i=0 to LayerNum-2 do

1. *Initialize **every row** of Y[i] with B[i],*

$$Y[i][j] = B[j\%col]$$

2.

$$Y[i] = \begin{cases} X \times W[i] + Y[i] & i = 0 \\ Y[i-1] \times W[i] + Y[i] & i > 0 \end{cases}$$

3. ***Sigmoid** on hidden layers or **Softmax** on output layer*

if(i==LayerNum-2) //output layer

$$Y[i][j] = \frac{e^{max\{Y[i][k]\}-Y[i][j]}}{\sum_{l=\frac{j}{col}}^{l=\frac{j}{col}+col-1} e^{max\{Y[i][k]\}-Y[i][l]}} \quad \frac{j}{col} \le k \le \frac{j}{col} + col - 1$$

else //hidden layer

$$Y[i][j] = \frac{1}{1+e^{-Y[i][j]}}$$

(3) Analysis

If we do not consider the difference inside *add*, *sub*, *multiply*, *div* and *exp* operation, it can be concluded that the computing complexity of this part is O (LayerNum $\times n^3$).

Through our analysis, we could optimize the parts as shown in follows:

1) We should use parallel methods to compute Matrix multiplication.
2) In computer, we use Taylor expansion to compute e^x. The time it cost will be much, if we can compute it faster under premise of accuracy, that will be better.
3) From the **perspective of mathematical**, we could find that we do not have to

compute $e^{\max\{Y[i][k]\}}$ during softmax on output layer. Here is the detailed reason:

$$\mathrm{Y}[i][\mathrm{j}] = \frac{e^{\max\{Y[i][k]\}-Y[i][j]}}{\sum_{l=\frac{j}{col}}^{l=\frac{j}{col}+col-1} e^{\max\{Y[i][k]\}-Y[i][l]}} = \frac{e^{-Y[i][j]}}{\sum_{l=\frac{j}{col}}^{l=\frac{j}{col}+col-1} e^{-Y[i][l]}}$$

4.1.2.2 Analysis of Backward

(1) Basic idea

During error back propagation, we mainly use target label of input, the output of the hidden layers and output layer got in the forward, and weight matrix to get the training error in reverse order.

(2) Algorithm

- **Firstly**, DNN uses the state of the output layer and the target label of input to get the training error of the output layer.

$$\mathrm{E}[LayerNum-2][\mathrm{j}] = \begin{cases} \mathrm{Y}[\mathrm{LayerNum}-2][\mathrm{j}] - 1.0 & j\%col = T[j/col] \\ \mathrm{Y}[\mathrm{LayerNum}-2][\mathrm{j}] & j\%col \neq T[j/col] \end{cases}$$

- **Secondly**, DNN uses the Back propagation to get the training error of the hidden layer in order.

  For i=LayerNum-3 to 0 do

  1. Propagate training error

  $$\mathrm{E}[\mathrm{i}] = \mathrm{E}[\mathrm{i}+1] \times W[i+1]^T$$

  2. Transform error, where $\mathrm{Y}[\mathrm{i}][\mathrm{j}] \times (1 - Y[i][j])$ is the derivative value of sigmoid function on Y[i][j]

  $$\mathrm{E}[i][\mathrm{j}] = \mathrm{E}[i][\mathrm{j}] \times \mathrm{Y}[\mathrm{i}][\mathrm{j}] \times (1 - Y[i][j])$$

(3) Analysis

Same to the forward part, computing complexity of this part is O (LayerNum $\times n^3$).

Through our analysis, the parts we think we could optimize are shown as follows:

1) We should use parallel methods to compute matrix multiplication.
2) Also, since here we use matrix transposition, we should pay attention to the Cache hit ratio of matrix multiplication.

4.1.2.3 Analysis of Update

(1) Basic idea

During this part, DNN mainly uses training error to compute bias delta and weight delta, as well as updating bias and weight matrix.

(2) Algorithm

- **Firstly**, DNN updates the bias and weight of the input layer.

  1. Update weight matrix

  $$\mathrm{Wdelta}[0] = \mathrm{alpha} \times X^T \times E[0]$$
  $$\mathrm{W}[0][\mathrm{j}] \mathrel{+}= \mathrm{Wdelta}[0][\mathrm{j}]$$

  2. Update bias

  $$\mathrm{Bdelta}[0][\mathrm{j}] = \mathrm{alpha} \times \sum_{l=0}^{minibatch} E[0][l * col + j]$$
  $$\mathrm{B}[0][\mathrm{j}] \mathrel{+}= \mathrm{Bdelta}[0][\mathrm{j}]$$

- **Secondly**, DNN updates the bias and weight of the hidden layers.

  For i=1 to LayerNum-2 do

  1. Update weight matrix

  $$\mathrm{Wdelta}[\mathrm{i}] = \mathrm{alpha} \times Y[i-1]^T \times E[i]$$

$$W[i][j] += Wdelta[i][j]$$

2. Update bias

$$Bdelta[i][j] = alpha \times \sum_{l=0}^{minibatch} E[i][l * col + j]$$

$$B[i][j] += Bdelta[i][j]$$

(3) Analysis

Similar to the forward part, the computing complexity of this part is also O (LayerNum × n^3).

Through our analysis, the parts we think we could optimize are shown as follows:

1) We should use parallel methods to compute Matrix multiplication.
2) In fact, we do not need to compute Wdelta and Bdelta separately which will waste our time.
3) Also, we need pay attention to matrix transposition and the computation of bias, here we need to notice the Cache hit ratio.
4) Interestingly, different from the forward calculation and the error back propagation, there is no data dependencies between adjacent layers in this part. So we can take parallelization strategy to update bias and weight of different layers.

4.1.2.4 Overall Analysis

Based on the previous analysis, it can be concluded that:

(1) Major computations inside DNN algorithm is ***matrix multiplication***. To achieve better performance, it is really necessary to use parallel methods to implement it;
(2) When updating *W[i]* or *B[i]*, it only needs to get *E[i]* in the backward. Therefore, it is possible to parallelize these two parts.

4.2 Parallelization and Optimization on CPU platform

4.2.1 Introduction

Based on previous analysis to DNN algorithm and the original program, we parallelize and optimize the DNN application on CPU platform in following aspects:

(1) We implement data parallelism for DNN application, that is, processing multiple mini-batches parallelly in multiple processes under restrictions of accuracy;
(2) Implement multithreading inside process to achieve fine-grained parallelism by using OpenMP;
(3) Use general-purpose optimizing approaches, such as data alignment, auto-vectorization, etc;
(4) Exploit optimized parameters and options for compilers and libraries.

Note: there are two workloads: one for debug and one for training. As the debug workload is suitable for analyzing the performance feature of the DNN application, the following experiments are based on the debug workload. And we only use training workload to compute speedup.

4.2.2 Data Parallelism

4.2.2.1 Data Parallelizability of DNN

The training process of DNN presents a non-convex optimization problem. Mini-batch gradient descent is used during this process. However, mini-batch gradient descent is a sequential process in nature. Thus, DNN is not well suited to parallel architectures.

To parallelize DNN, we use Asynchronous Stochastic Gradient Descent(ASGD) algorithm. The algorithm of ASGD is shown in Algorithm 3.

Algorithm 3 Asynchronous Stochastic Gradient Descent

Client	Server
Loop ***X*** = get next mini-batch ***W*** = fetch latest parameters from server **if *X*** is empty, **then** exit ***Delta*** = calculate gradient based on ***X*** and ***W*** Upload ***Delta*** **end loop**	**Loop** ***request*** = Wait for request from client **if *request*** is fetching data, **then** send ***X*** and ***W***. **if** request is upload ***delta***, **then** update ***W*** based on the uploaded ***delta***. **end loop**

The program runs faster when applying this algorithm. However, the accuracy rate decreases when we increase the number of processes. According to *On Parallelizability of Stochastic Gradient Descent for Speech DNNs*, there are two factors affecting the parallelizability: mini-batch size and data compression[2]. Both increasing mini-batch size and decreasing data compression can increase parallelizability. For us, we explore the former.

The main method to increase mini-batch is "*AdaGrad*", which leads to more mature models earlier, thus allowing for larger N. Our implementation is based on the description in *Large Scale Distributed Deep Networks*[3]. We applied AdaGrad to server side. To be more specific, we use a separate learning rate for each parameter. Let $\mu_{i,K}$be the learning rate of the i-th parameter at iteration K and $\Delta w_{i,j}$ is its gradient. We set $\mu_{i,K} = \frac{\gamma}{\sqrt{\sum_{j=1}^{K} \Delta W_{i,j}{}^2}}$. The value of γ is the constant scaling factor for all learning rates. We choose a suitable γ from a lot of valid values by using an automatic script.

4.2.2.2 Implementation of Asynchronous SGD (with AdaGrad) Based on MPI

Message Passing Interface (MPI) is widely used technology in parallel software field. It enables programmers to write programs that run on distributed systems. The first version of our asynchronous stochastic gradient descent implementation is based on MPI.

In our implementation (its concept architecture is shown in Figure 31), there are two kinds of processes: server and client. The process whose process identifier(PID) is zero behaves as a server process. It reads chunk data, distributes mini-batch to clients and

[2] H. F. J. D. G. L. D. Y. Frank Seide, "ON PARALLELIZABILITY OF STOCHASTIC GRADIENT DESCENT FOR SPEECH DNNS," from IEEE International Conference on Acoustic, Speech and Singal Processing, 2014

[3] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M.A.Ranzato, A. Senior, P. Tucker, K. Yang, A. Y. Ng, "Large Scale Distributed Deep Networks," NIPS, 2012

maintains the latest value of parameters. Other processes are client processes, who compute the data received and transfer the data to server and get new mini-batch and new references. When there is no data, client processed end and server writes result.

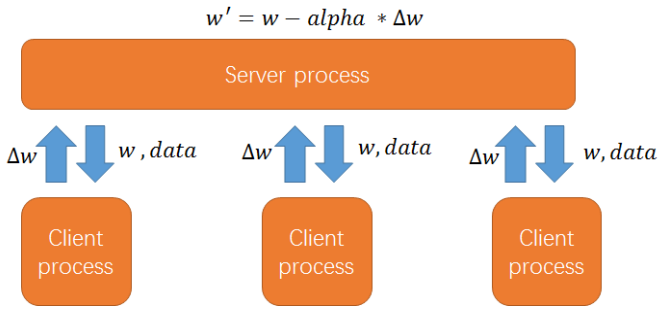


Figure 31 Data parallelization of MPI model

We tested our MPI version of ASGD, and collected the accuracy data of it. The result can be seen from Figure 32 and Figure 33.

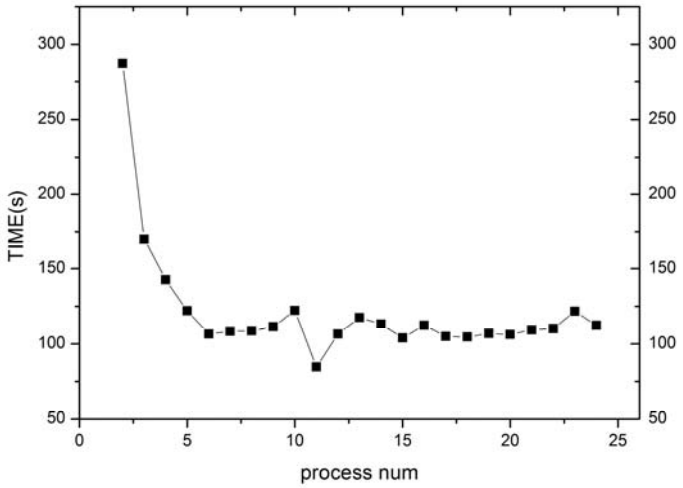


Figure 32 time of ASGD using MPI with different work threads

Figure 32 shows that data parallelism could speed up the original program remarkable. More specifically, the time which program costs reduce sharply when the process number is less than 6. At the same time. the performance of the program keeps stable when the process number is greater than 5.

According to Figure 33, the accuracy is unsatisfactory when work threads exceeds 8. In summarize, our experiment validate that data parallelism is an effective method to boosting the performance of the program under the condition that the number of threads cannot exceed the parallelizability of the program.

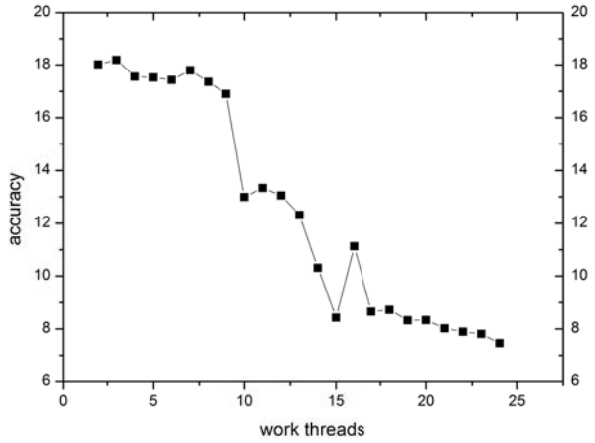


Figure 33 accuracy of ASGD using MPI with different work threads

4.2.2.3 OpenMP Implementation of *ASGD*

After tuning the performance of MPI version, we find that the necklace of our implementation is communication cost. In our design, parameter server sends bunch data to clients, which causes several copy operations. To avoid unnecessary copy operations, we rewrite *ASGD* in OpenMP. Here is our Data parallelization of OMP model.

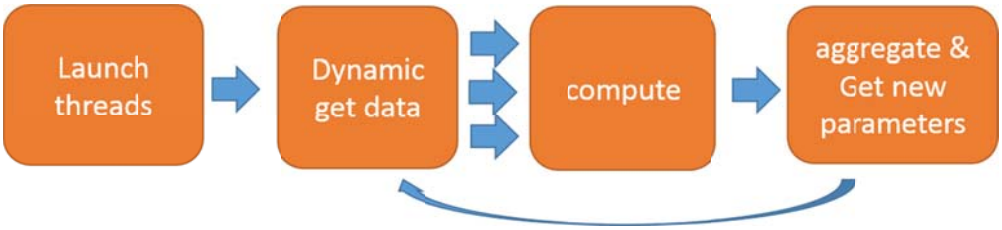


Figure 34 Data parallelization of OMP model

OpenMP based *ASGD* implementation does not have a server thread. Instead, all of threads share one central model which contains the latest parameters. In addition, each thread has its own local replica.

We tested the performance of our implementation based on OpenMP when using 1-8 to eight work threads. (Due to the results of our MPI experiments, we do not test the situation that using more than 8 threads, for its accuracy is unsatisfied.). From this, we can find that when threads are around 3, the program works well.

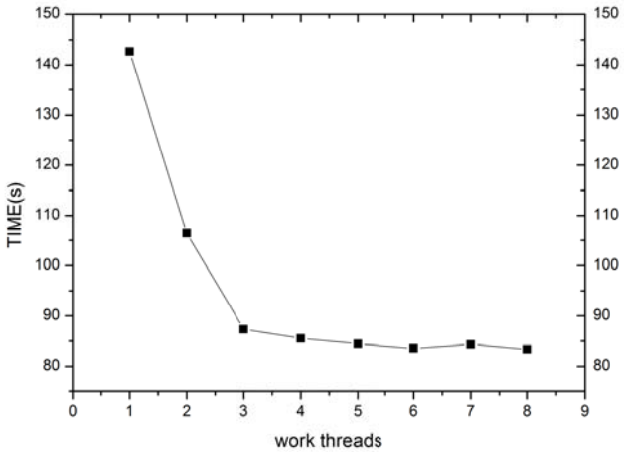


Figure 35 Time when using different number of work threads

And there is the accuracy statistics when using different number of work threads. As we can see from Figure 36, there are few differences when using 1-3 work threads. Then, the accuracy drops down rapidly when we launch more than 4 threads, which indicates the number of threads exceeds the parallelizability.

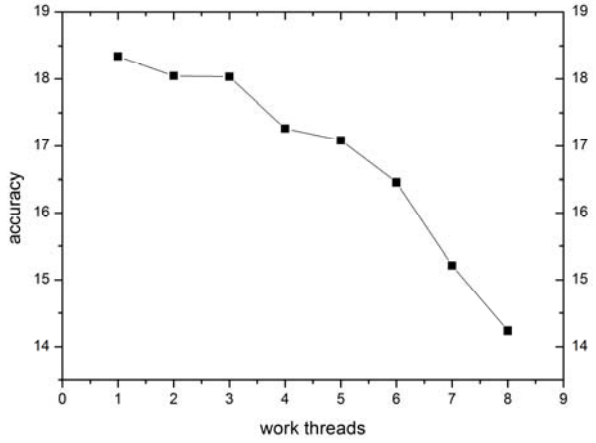


Figure 36 Accuracy when using different number of work threads

To maximize the number of threads, we use 3 work threads as it isn't exceeds the limit. Then we tunend the number of MKL work threads. Intel MKL can dynamiclly lunch suitable number of threads. However, it is not perfect. We set the number of MKL threads staticlly, testing the performance of the program. Interestingly, we find 8 MKL threads reachs the highest performance.

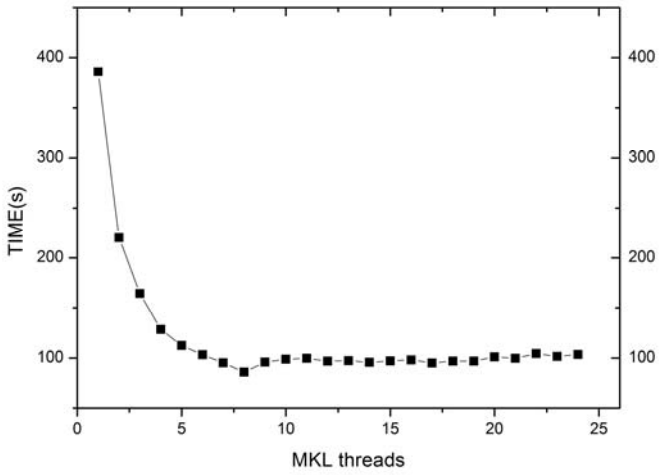


Figure 37 the time of different MKL threads

4.2.3 Compiler and Environment Related optimization

4.2.3.1 Compiler

The default compiler of the origin code is *g++* which supports various platforms. An alternate option is *ICPC*, a compiler from Intel. *ICPC* can generate highly optimized binary code on Intel platforms. We compared the performance of compiled programs of *g++* and *ICPC*. The result is shown in Figure 38(linked with sequential version of Intel *MKL*).

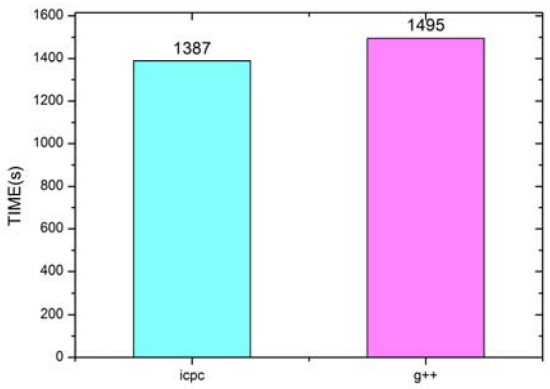


Figure 38 Execution time of the program compiled by *G++* and *ICPC*

As can be seen in Figure 38, compiling program using *ICPC* leads to performance improvement. Therefore, we replace *g++* with *ICPC* in order to produce more effective binary code.

4.2.3.2 Number of MKL Working Threads

The original program uses BLAS (Basic Linear Algebra Subprograms) for computing matrix multiplication. We choose Intel MKL (Math Kernel Library), which is a high performance library developed by intel, as the implementation of BLAS. MKL implements multiple threads matrix multiplication. Thus, the number of work threads used by MKL is an important factor that influences the speed of the program. We tested the performance of the program under different number of MKL threads. The results are shown in Figure 39.

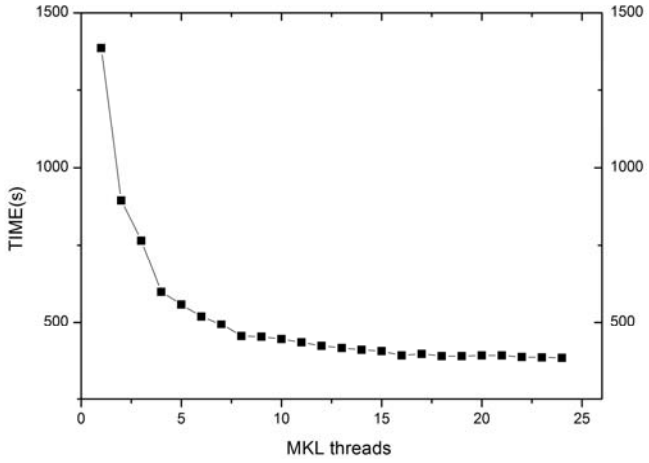


Figure 39 Execution time under different number of MKL threads

It is obvious that the more threads MKL uses, the faster the program runs. But MKL cannot archive linear speedup. The speedup is not very obvious when the number of threads exceeds around 8. So it may reach higher performance if we can execute three processes (each process has 8 threads for computing matrix multiplication) in parallel.

4.2.3.3 Compiler Optimization options

Modern compilers support various optimization strategies. Some aggressive optimizations may cause worse performance and others may lead to better performance. To find the most suitable optimization options, we did some experiments. All of the options we tested are listed in Table 18.

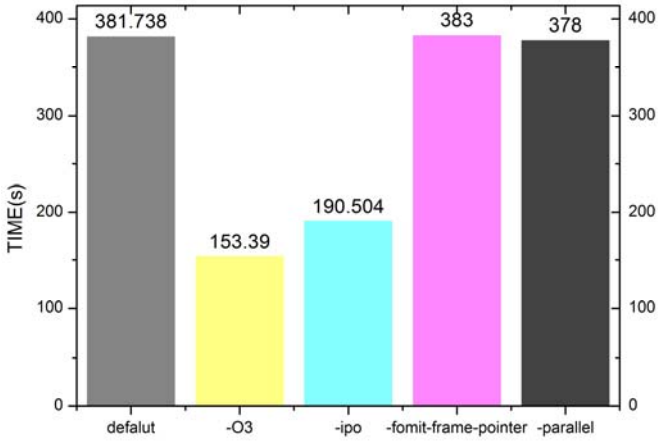
Table 18 Optimization Options and Details

Optimization Options	Description
default	Does not add any extra optimization flags.

-O3	Enables more aggressive loop transformations. Note that the O3 optimizations may not cause higher performance.
-ipo	Enables inter-procedural optimization.
-fomit-frame-pointer	Enables EBP as a general purpose register
-parallel	Auto parallelize

As we can see from Note:linked with MKL multi-threads version

Figure 40, *-O3* option cause higher performance because of its aggressive optimization. In addition, using *EBP* as a general purpose register does not lead to better performance, either. As for inter-procedural optimization, it really has obvious effects. Inter-procedural optimization focus on analyzing the possible value sets of pointers, which enables compiler to generate more effective binary code. As a result, the performance of the program is obviously improved. Interestingly, as a result of auto parallelize optimization, the performance of the program is improved although speedup is less than 1%.



Note:linked with MKL multi-threads version

Figure 40 The effects of different optimization options

In conclusion, we enable *-O3*, *–ipo* and *–parallel* options for better performance (it only takes 149.504 seconds).

4.2.4 Data Alignment and Vectorization

By analyzing source code, we find that there are lots of operation applying for space during *dnn_utility.cpp* file. To our dismay, the operation is *new()*, which doesn't make data forced alignment. In order to reduce the CPU access to memory, we change it to *_mm_malloc()*. Besides, we find that most of the loop statements don't have data dependencies, for example:

```
1  for (int i = 0;i < row*col; ++i)
2  {
3          Y[i] = 1.0f/(1.0f + expf(-Y[i]));
4  }
```

We could easily find there is no dependencies between *Y[i]* and *Y[j]* when *i! = j*. So we can take compiler options to make it automatic vectorization, like *-xHost*, so that we can make our program faster.

Based on the above analysis, we did some related experiment, the results are as follows:

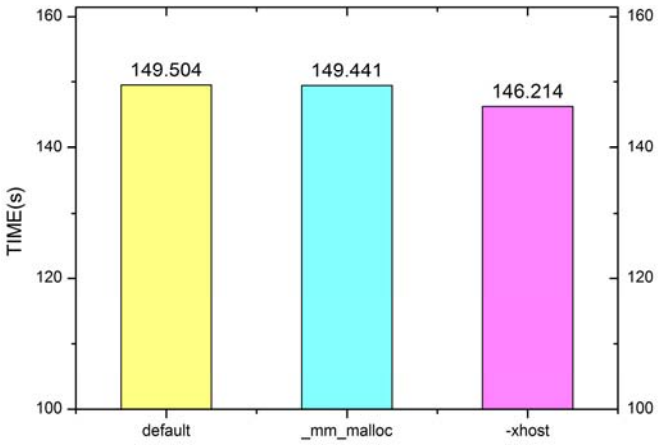


Figure 41 the effects of data alignment and vectorization

-xHost option enables all of the instruction sets supported by the host, which means it enables auto-vectorization, an optimization method that rewrite loops in SIMD instructions. As can been seen from Figure 41, auto-vectorization leads to a small performance improvement. Note that the default version used here applies *-O3*, *-ipo* and *-parallel*.

4.2.5 Final Execution Time and Speedup on ASC16 Remote CPU Platform

After all the experiments and optimizations on CPU, it takes ***87.042s*** for our CPU DNN program to execute the debug workload. It takes ***310.152s*** to execute the training workload. Comparing to the original sequential program, the final speedup of DNN on CPU platform is:

(1) Debug workload

$$\text{Speedup} = \frac{Execution\ time\ of\ original\ sequencial\ program}{Excution\ time\ of\ optimized\ parallel\ program\ on\ CPU} = \frac{1495.367}{87.042} = 17.180$$

(2) Training workload

$$\text{Speedup} = \frac{Execution\ time\ of\ original\ sequencial\ program}{Excution\ time\ of\ optimized\ parallel\ program\ on\ CPU} = \frac{4860.571}{310.152} = 15.672$$

4.3 Hybrid Parallelization and optimization on CPU+MIC platform

In this section, we will introduce our optimization on MIC. Mainly talking about the asynchronous transfer optimization and data parallelization's difference with it on CPU. Here is our model:

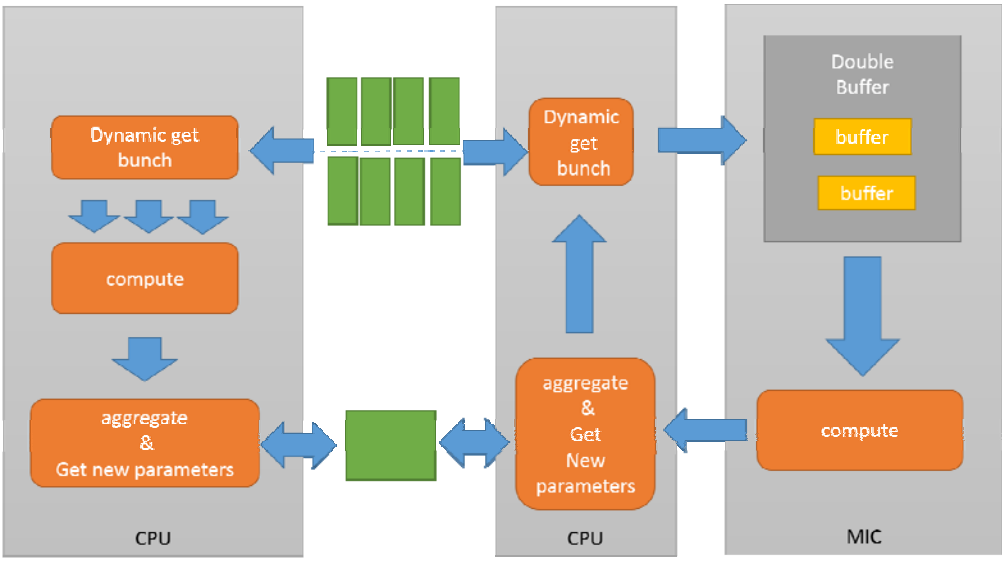


Figure 42 CPU+MIC model

4.3.1 Dynamic Load Balance

For CPU+MIC Heterogeneous computing platform, the strategy of distributing work to the work threads on CPU and MIC is an important point.

A sample strategy is dividing training data statically, which means the data that will be computed by CPU or MIC has already been decided before the training process. As a result, the workload on CPU and MIC may be unbalanced.

To avoid this problem, we apply a dynamic strategy to distribute training data. To be more specific, the training data will be requested by idle work threads. In detail, when a thread requests data, it will lock the data buffer and get a bunch of samples from it. After that, it will unlock the data buffer, enabling other threads to read data. We implement a double-buffer mechanism on MIC platform. has a double-buffer to enable asynchronous data transfer from CPU to MIC.

Here is the result of the experiment about the speed and accuracy when there are 2,3, and 4 work threads.

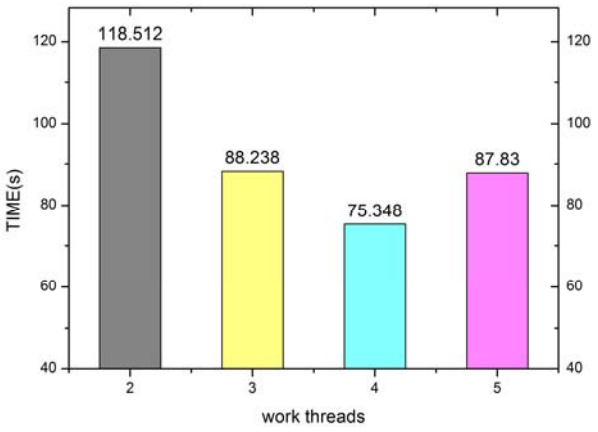


Figure 43 Performance under different number of work threads on CPU+MIC

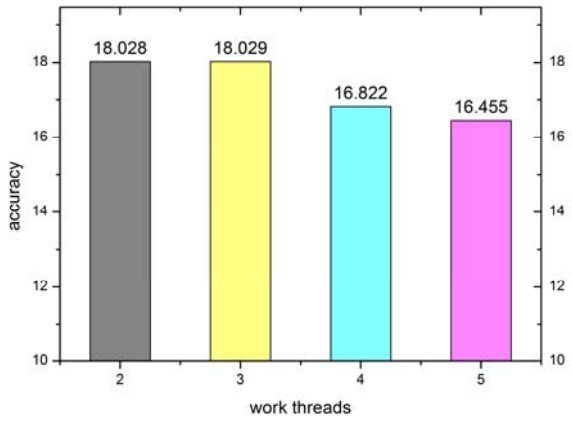


Figure 44 Accuracy under different number of work threads on CPU+MIC

4.3.2 Asynchronous Transfer Optimization

When computing on MIC, we apply two main methods to parallelizing it: setting double-buffer to get new data and parameters and merging data asynchronous. We will discuss them in the following sections.

4.3.2.1 Double-Buffer

We use a double-buffer scheme to support asynchronous data transfer between CPU and MIC. One of the buffers is used by the work thread, and the other is used for storing data that is asynchronous transferred from host. Once computing and data transfer completed, the two buffers are swapped.

4.3.2.2 Asynchronous Aggregate Gradient

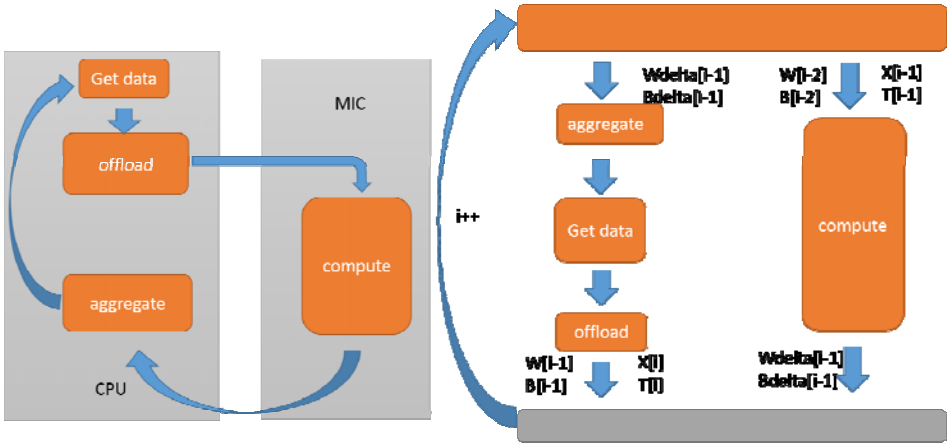


Figure 45 Difference between synchronous and asynchronous

On the left side of Figure 45 shows the synchronous aggregate gradient process. Even through the data transfer from CPU to MIC is asynchronously, we have no way but to wait the time when CPU have already get the data from MIC. And only after that time, MIC could get its new parameters from CPU. That will waste a lot of time.

But when we use the model on the right, the problem has been solved. When MIC is computing, we can fetch the parameters on the MIC got in the last iteration. And when MIC has ended its computation, what it needs to do is just change its nodeArg's d_X, d_T, d_W and d_B 's pointer and let it point to the buffer where contains new data which has been transferred by CPU. And then it can start its next iteration. Thus, MIC just need to wait the pointer changed. It will save much time!

But there is also a problem that we can found. It is that our d_W, d_B is a little more delayed than it on CPU. So when we aggregate the d_Wdelta and d_Bdelta to the nodeArg shared by all threads, it will lose some accuracy. So we aggregate the d_Wdelta and d_Bdelta computed by CPU a little different from MIC. According to the results of our experiments (shown in Figure 46), it works better.

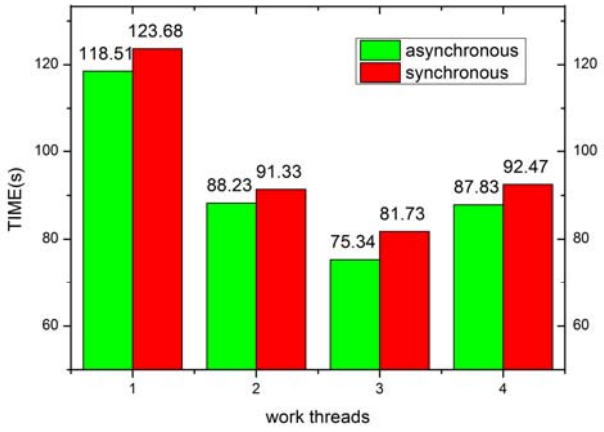


Figure 46 Comparison between synchronous and asynchronous aggregate gradient

4.3.3 Miscellaneous Optimization

4.3.3.1 Reuse Memory

Through a series of experiments, we found that memory allocation on MIC is an expensive operation. If the code requires memory allocation in each iteration, the benefit of computing data on MIC would be affected. To avoid this problem, we only execute memory allocation just the first iteration and reuse them during the following iterations.

4.3.3.2 Cache Optimization

The cache architecture of MIC is different from that of CPU. Each core of MIC has 32KB L1 instruct cache and 32KB L1 data cache, the same as CPU. In addition, MIC has 512KB L2 cache shared by all cores, which is bigger but slower than L2 cache on CPU. So, we optimize cache hit ratio on MIC, getting remarkable effect on MIC. The result can be seen from Figure 47 and Figure 48.

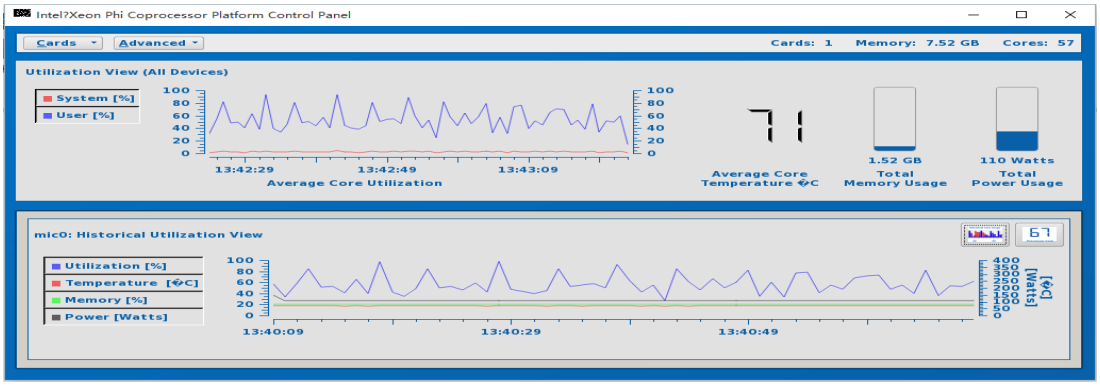


Figure 47 Before applying cache optimization

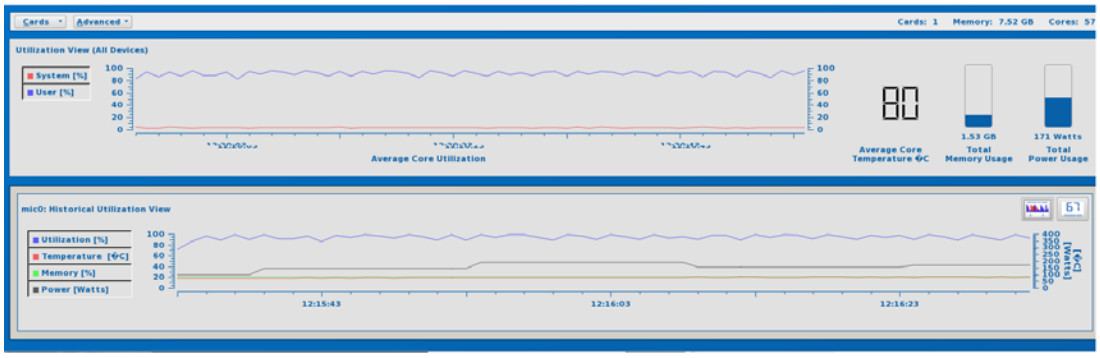


Figure 48 After applying cache optimization

From Figure 47 and Figure 48, we can find that improving cache hit ratio is a good way to optimize the performance of MIC.

4.3.4 Final Execution Time and Speedup on Remote CPU+MIC Platform

After all the experiments and optimizations, it takes ***75.348s*** for our CPU+MIC DNN program to execute the debug workload. As for the training workload, it takes ***285.265s***. Comparing to the original sequential program, the final speedup of DNN on CPU+MIC platform is:

(1) Debug workload

$$\text{Speedup} = \frac{Execution\ time\ of\ original\ sequencial\ program}{Excution\ time\ of\ optimized\ parallel\ program\ on\ CPU+MIC} = \frac{1495.367}{75.348} = 19.846$$

(2) Training workload

$$\text{Speedup} = \frac{Execution\ time\ of\ original\ sequencial\ program}{Excution\ time\ of\ optimized\ parallel\ program\ on\ CPU+MIC} = \frac{4860.571}{285.265} = 17.039$$