International Journal of Hybrid Information Technology
Vol. 9, No.11 (2016), pp. 239-254
http://dx.doi.org/10.14257/ijhit.2016.9.11.21

Performance Evaluation and Optimization of HPCG benchmark on CPU + MIC platform

Qingyi Pan[1] and Xiaoying Wang[1,*]

[1]*State Key Laboratory of Plateau Ecology and Agriculture, Department of Computer Technology and Applications, Qinghai University, Xining, China, 810016*
[*]*wxy_cta@qhu.edu.cn*

Abstract

High-performance conjugate gradient (HPCG) is the latest benchmark adopted by the TOP500 organization, and thus how to optimize the HPCG source code for different heterogeneous computing platforms to achieve a higher floating-point computation rate has already become a new hot issue in HPC field. In the paper, we used the CPU + MIC heterogeneous computing platforms, and successfully ported HPCG to the platform. Through the analysis of HPCG source code and optimization for CPU + MIC platforms, practical significance and the value of further research is put forward. Results of performing the benchmark indicate that the design of optimization methods is reasonable and has facilitated the speedup of HPCG benchmark.

Keywords*: HPCG benchmark, HPL benchmark, MIC co-processor, heterogeneous computing*

1. Introduction

With the development of a series of technologies related to the supercomputers, high-performance computing plays an increasingly important role in many fields, such as the South African SKA radio telescope, the analysis of financial model, we all need the high-performance computing model to support them, so the supercomputers running high-performance computing also take an increasingly essential responsibility. Thus, how to judge the supercomputer performance effectively and scientifically has become a central topic.

In various performance benchmarks of supercomputer, the most famous test is HPL 0, using the peak floating-point calculation performance as metric. Nevertheless, the problem of HPL in most systems is that it’s easy to be vectorized, and thus the floating-point computation rate tested in the actual application may be only a tenth of peak performance. Hence, only using the results obtained by HPL metric to measure performance of the multi-processor computer systems is incomplete [2]. On the basis of HPL, HPCC provides far more metric results than single performance metric routine. However, since the test procedure and test results are too complex, it has not been widely adopted. Subsequently, the University of Tennessee has proposed a latest metric to better reflect the supercomputer system performance again - HPCG. It is served as an alternative standard for ranking the Top500 list and applied widely.

In this paper, we conduct discussion and research based on the latest performance benchmark of TOP500 supercomputer, and optimize metric results upon a real platform. The benchmark version used in this paper is HPCG-3.0, which was released at the end of 2015 and became the latest benchmark for peak performance evaluation of supercomputer systems.

2. HPCG: The Benchmark for Supercomputer

The reference HPCG is written in C++, using some C++ standard libraries and container classes. Also, MPI and OpenMP can be selected by user for parallelization at the compile time.[3] HPCG benchmark generates a synthetic three-dimensional partial differential equations model problem, and computes sparse linear system generated by gradient iterative pretreatment. In the establishment, the program will construct a logically global, physically distributed sparse linear system. The special sparse matrix system used is a simple elliptic partial differential equations and it can also be described as the 27-point stencil at each grid point in the 3D domain (as Figure 1 shows), such that point equation (*i, j, k*) depends entirely on the value of its location and other surrounding points.

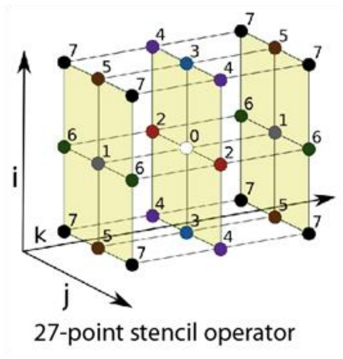

Figure 1. Sparse Linear Matrix Model

Defining the sparse linear system is to provide easy access to execute a collection of important computational kernels and the metric mainly uses the symmetric Gauss-Seidel preconditioned iterative method to compute the higher order linear sparse matrix, which is the most expensive routine in the metric.

Assume the *i*-th equation of the sparse linear system Ax=b [4] is:

$$\sum_{j=1}^{n} a_{ij}x_j = b(i=1,2...,n)$$

After proper formula deformation of above formula, we can obtain the symmetric Gauss-Seidel preconditioned iterative formula as follows:

$$\mathrm{x}_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)})(i=1,2,...,n)$$

Take the proper $x^{(0)}$ as the initial vector, can we obtain the approximate solution vector $\left|x^{(k)}\right|$ by the iterative format. In HPCG, the solution to the equations make further improvement on the basis, it performs two steps: forward and backward sweeps.[5] Firstly, using the initial value $x^{(0)}$ to get corresponding $\mathrm{x}^{(k+1)}$ values by the above-described algorithm according to the order of the line. Continuously using $\mathrm{x}^{(k+1)}$ as an initial value to sweep back. With the above algorithm by the reverse order of the line, continuing to iterate to get the final result. Therefore, more accurate computational results are obtained. Specific formula is as follows:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{\substack{j=1 \\ \mathrm{i}\neq j}}^{n} a_{ij}x_j^{(k+1)})\ (i = n, \ldots, 2,1)$$

Meanwhile, it needs to be paid attention to that the sparse matrix used in the HPCG test is not necessarily symmetrical. So the optimized code should be able to handle the improved general sparse matrix structure instead of particular matrix structures.

The HPCG execution procedure is shown in Figure 2. The focus of the program is operation sparse matrices [3]. To achieve faster floating-point computational rates, we need to optimize the metric for particular co-processor. In the paper, we select many-core Intel Architecture co-processor (MIC) to optimize the source code of the HPCG. The executing process of HPCG benchmark is shown in Figure2.

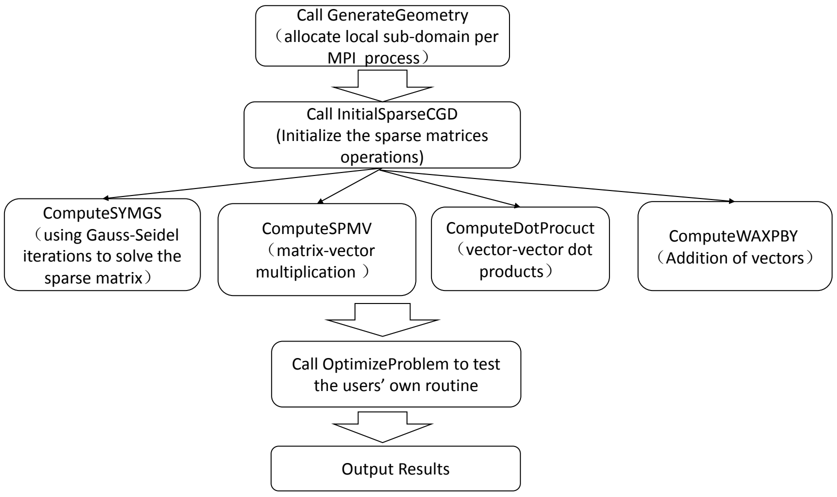
Call GenerateGeometry （allocate local sub-domain per MPI process）
Call InitialSparseCGD (Initialize the sparse matrices operations)
ComputeSYMGS （using Gauss-Seidel iterations to solve the sparse matrix）
ComputeSPMV （matrix-vector multiplication ）
ComputeDotProcuct （vector-vector dot products）
ComputeWAXPBY （Addition of vectors）
Call OptimizeProblem to test the users' own routine
Output Results

Figure 2. The Executing Process of HPCG Benchmark

3. Optimization of the Source Code of HPCG for MIC Co-Processor

In this paper, the test is based on MIC co-processor (Intel Xeon Phi). The co-processor is compatible of code on the Intel CPU in the largest degree. Just need to add pragma tags and include directives for transferring data, the routine runs on the multi-core platform. The programming tools and languages employed for code development for a MIC are the same as those used for CPUs. The MIC co-processor owns independent memory, power supply and other equipment MIC co-processor owns 61 core totally, and single-core computational capability is stronger than that of GPU, and per core can provides four threads, being able to provide more than 230 threads in total.

Although porting the source code on the CPU side to MIC almost need no adjustment, but in order to utilize the computational capacity of MIC co-processor, the optimization of code is essential. In this paper, the metric is optimized for CPU+MIC platform. The reason why we did not choose the GPU + CPU heterogeneous computing platform is that, it is difficult to port the code on GPU platform to other platform, and the parallel computational patterns of routines are too little.

3.1. Offload Programming Mode

In the process of porting the HPCG benchmark to the MIC side, the offload programming mode is mainly used[6] , which is the CPU (Host side) offload computing kernels on the MIC co-processor (Device side) for high performance parallel computation, then CPU continues dealing with other matters, meanwhile MIC co-processor continues computing the hot spot kernel. Finally, the computational results will be transferred to the CPU side when finished.

The most common used key words in offload programming mode is: offload. It only need to add a small amount of guide statements, such as:

```
#pragma offload target( mic:0)   // port the source code to No.0 mic cards
    for(int i=0; i<N; i++ )
        a[i] = b[i] + c[i];
```

But at this time, only transplanting the source code to the MIC co-processor, the routine still executes serially. Combining with OpenMP parallel programming mode makes the program execute in parallel and improve the execution rate.[7]

```
#pragma offload target(mic:0)
#pragma omp parallel for
for(int i=0; i<N; i++ )
    a[i] = b[i] + c[i];
```

The above is the offload mode of original program, only using a line of instruction ports the routine to MIC side, the implement on the MIC is simple and high efficiency. As for the optimization for the MIC co-processor, it also need other keywords (*i.e.*, in, out, nocopy *etc*.) to control the transfer between the Host side and Device side.

3.2. Vectorization

In the OpenMP parallel programming mode, the vectorization is also of great significance in optimize the routine on MIC side[8]. For the MIC routine, -vec is needed to switch on the vectorization. At the compiling time, the Intel compiler will vectorize the internal vector automatically. When it is determined that the part to be vectorized does not have the data dependence, the (#pragma ivdep) option can be annotated out of the circulation, and we even can annotated the code with(#pragma simd) to guide vectorization, ignoring the warning. When it is uncertain whether to be able to vectorize or not, (#pragma vector always) can be used to avoid some operations which is not memory alignment to be vectorized. In addition, SIMD can be modified to vectorize the program more effectively, but the transportability of SIMD is relatively poor too complex and relies too heavily on hardware architecture. In this paper, we mainly use the ICC compiler to vectorize the routine automatically.

4. HPCG Acceleration Methods

4.1. Basic Approach

In order to port the routine on CPU to MIC side conveniently and accelerate the algorithm, the basic acceleration scheme is to use the OpenMP parallel programming mode, without considering the details of the algorithm.[9] Just using the advantage of multithreading of MIC co-processor accelerates the part of cycle,[10] like the following pseudocode, but the acceleration has not fully utilized the computing powers of MIC, and the computational results only obtain 1.x speedup.

```
 #pragma omp parallel for
for(i=0; i<LEN; i++)
{
    a[i]=b[i]+c[i];
}
```

4.2. Parameter Tuning

In general, the larger size of the matrix is, the larger percent of the valid computation and the higher floating-point computation rate are[11] . But at the same time, larger size of the matrix will lead to the increase of memory consumption. Once the memory of system is insufficient, the computing capacity will reduce obviously. While increasing the size of matrix, the actual capacity of memory needed to be considered[12] .

In the latest HPCG benchmark, the problem size is arbitrary (The value in file *hpcg.dat*). Firstly, by modifying and getting the appropriate size of matrix, let the MIC co-processor approach the peak performance of floating-point computation rate. In HPCG, users can set the size of $N_x \times N_y \times N_z$ in *hpcg.dat* file, which is the whole size of matrices to be assigned to each MPI process. While the number of processor is automatically detected at run-time, and be assigned to $P_x \times P_y \times P_z$, the whole number of processor. Then the corresponding parameters $G_x \times G_y \times G_z$ is obtained, which is $G_x = P_x N_x$, $G_y = P_y N_y$ and $G_z = P_z N_z$. Finally, it is required that the matrices processed is like a cube as much as possible, in that the method can improve the floating-point capability of processor, so it is essential to select the appropriate size of sparse matrix.

4.3. Further Acceleration based on MIC

4.3.1. Analysis of the Hot Spot of HPCG

Although porting the routine on CPU side to MIC side only needs a line of guidance statement, but in order to reach the peak performance of processing, it must be combined with the characteristics of MIC to optimize the source code of HPCG.[13] By monitoring the serial code using the VTune tool, we found that *ComputeSYMGS*, *ComputeWAXPBY*, *ComputeDotProduct* and *ComputeSPMV*, which occupy over 80% of whole time of algorithm, in another word, which is the computational kernel of the algorithm. They represent several aspects of matrix operations respectively, meanwhile, the cooperative mode between CPU and MIC is offload. The CPU side is responsible for low degree of parallelism program (i e,. the initialization of the matrix, output the result of computing and *etc*.).[14] In the way of asynchronous computing, it will unload the hot spot kernels in high degree of parallelism on MIC side. CPU is waiting until the computational results are transferred. Finally it outputs the metric result, which greatly increase the computational efficiency of the program[15] . The offload programming mode is shown in Figure 3.

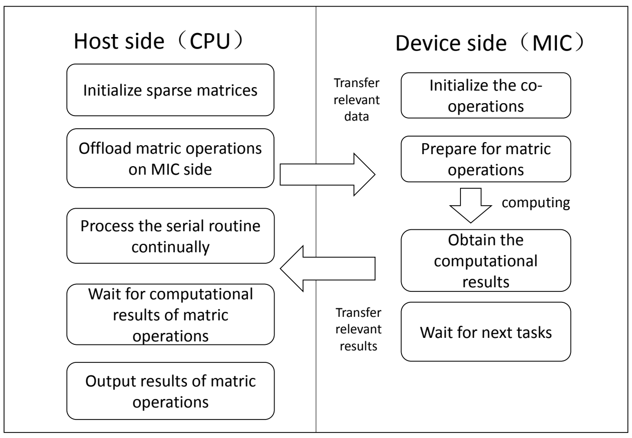
Host side（CPU）
Device side（MIC）
Initialize sparse matrices
Transfer relevant data
Initialize the co-operations
Offload matric operations on MIC side
Prepare for matric operations
computing
Process the serial routine continually
Obtain the computational results
Wait for computational results of matric operations
Transfer relevant results
Wait for next tasks
Output results of matric operations

Figure 3. Executing Process of HPCG Metric

4.3.2. The Implementation of HPCG Parallel Algorithm

(1) vector multiplication

In the sparse matrix manipulation, the vector multiplication is essential, which is mainly implemented in the *ComputeDotProduct* file. Assuming the n-dimensional vectors x and y, the computational result is marked as $result = x \times y$,which is obtained by multiplying the *i*-th element of x and *i*-th element of y. It can be expressed as:

$$result = \sum_{i=1}^{n} x_i \times y_i (i = 1,2,...,n)$$

Serial pseudo-code 3.1:

```
Begin
   For i=0 To n
       Result+=x[i]*y[i]
   End For
End
```

The analysis of serial pseudo-code shows that the loop has no data dependency and good parallelizability. By adding guidance statements, porting it to the MIC side, and takes multi-threaded execution, then taking advantage of auto-vectorization of Intel compiler, make the program execute in parallel.[16] When the loop begins, call the statement ‘in’ to transfer the value of vector *x* and *y* on CPU side to the MIC side and call ‘allocif()’ statement to apply dynamically for the space on MIC side. After the loop finished, using ‘out’ statement to transfer the experimental result back to the CPU side and call ‘freeif()’ statement to free the space. Meanwhile, in the inner calculation kernel, the compiler will vectorize the program automatically to improve parallelism.

Parallel pseudo-code 3.1

```
#pragma omp parallel for
#pragma offload target(mix:0)
  in(N)
in(x:length(N)) allocif(1) ,freeif(0)
in(y:length(N)) allocif(1) ,freeif(0)
out(result) allocif(1) ,freeif(0)
 Begin
     #pragma ivdep
       For i=0 To n
         Result+=x[i]*y[i]
       End For
 End
```

(2) Addition of vectors

The addition of vectors is also essential in the sparse matrix computation, which is mainly implemented in the *ComputeWAXPBY* file. Firstly, initialize vectors x and y and two coefficients α and β respectively, the computational result is recorded as $w = \alpha \times x + \beta \times y$. In the process, the components of the vectors x and y will multiply the corresponding coefficient and are added up, it can be expressed as:

$$w[i] = \alpha \times x[i] + \beta \times y[i] (i = 1,2,...,n)$$

Serial pseudo-code 3.2:

```
Begin
      For i=0 To n
          If alpha ==1    w[i]=x[i]+beta*y[i]       End if
          if beta == 1    w[i]=alpha*x[i]+y[i]       End if
          Else            w[i]=alpha*x[i]+beta*y[i]   End else
   End For
```

```
End
```

In the loop, if-else judgment might make the compiler mistaken that the code might exists data dependency, which may leads to the failure of vectorization, so it needs to add compulsory guidance statement. Here the sum of two vectors is able to be processed in parallel typically. Similarity, using 'in' and 'out' statement to port the input and output vectors to the MIC side, meanwhile using nocopy() to reduce the I/O interaction between device side and host side, and vectors which have nothing to do with the results are no longer transferred out of the MIC side. Utilizing the multi-thread capability of MIC fully can greatly improve the efficiency of vector addition.

Parallel pseudo-code 3.2

```
//transfer the variable needed by the MIC side in order to initialize the sparse matrix.
    …….
    Begin
      #pragma simd
          For i=0 To n
              if  alpha==1   w[i]=x[i]+beta*y[i]        End if
              if  beta==1   w[i]=alpha*x[i]+y[i]        End if
              Else          w[i]=alpha*x[i]+beta*y[i]   End else
      End For
    End
      //release the space on MIC side
        ……
    nocopy((x:length(n)) allocif(0), freeif(1))
    nocopy((y:length(n)) allocif(0),freeif(1))
    out(w:length(n))allocif(0) freeif(1)
```

(3) Matrices and vector multiplication

In HPCG, ComputeSPMV file implements the multiplication between matrices and vectors, which is the key point of optimizing the source code of HPCG. In HPCG, assuming that $n \times n$ coefficient matrix A and $n \times 1$ vector is known, the result is recorded as $n \times 1$ output vector, which is $y = Ax$, it can be expressed as the following:

$$y_{i1} = \sum_{j=1}^{n} (a_{ij} \times x_{j1})$$

The serial pseudo-code 3.3:

```
For y=0 To Width
....
  For  i=0  To  Height
      // execute the computation of sparse matrix
      sum+=x[j]*a[i][j]
      End For
End For
//CPU continues running
```

As it can been seen from the above serial code, the main part of the algorithm focuses on a nested loop body. We found no data dependencies in Sparse matrix operations, which indicates the matrix multiplication has good parallelizability. Matrix and vector multiplication uses the asynchronous computation mode. At first, the matrix A and vector is transferred to the MIC side for computing. Meanwhile, CPU continues running routines in high degree of serial. However, when CPU will run the next step relevant to matrix multiplication, it needs the wait for the computational results of matrix calculations transferred back from the MIC side. By using the statement "wait" and "signal", the mode can be implemented. Under the premise of utilizing fully of CPU computational resources, the correctness of the computational results can be ensured, also the efficiency of the executing can be improved greatly.

The parallel pseudo-code 3.3:

```
   //transmit the variable needed by MIC side to initialize the sparse matrix.
   //check whether the computation is finished or not
   signal(in1)  {
   For i=0 To N
    #pragma  ivdep
      For  j=0  To  N
     sum+=x[j]*a[i][j]
            End For
            y[i] = sum
    End For
//accept the data from MIC side, which signs that the kernel
//on MIC side have already been accomplished.
//release the space on MIC
     nocopy((x:length(n)) allocif(0),  freeif(1))
     nocopy((y:length(n)) allocif(0),  freeif(1))
      out(sum:length(n)) allocif(0),  freeif(1)
        }
```

4.4. Heat Dissipation of MIC Co-Processor

Due to fact that the power consumption of MIC co-processor is high, it is necessary to handle the heat dissipation of the machine on the platform. Taken into consideration that when the working temperature is too high, it will cause the frequency reduction of co-processor, which will lead to extreme poor performance, while the wrong computational results may be obtained. We take the following method to deal with the problem of heat dissipation.

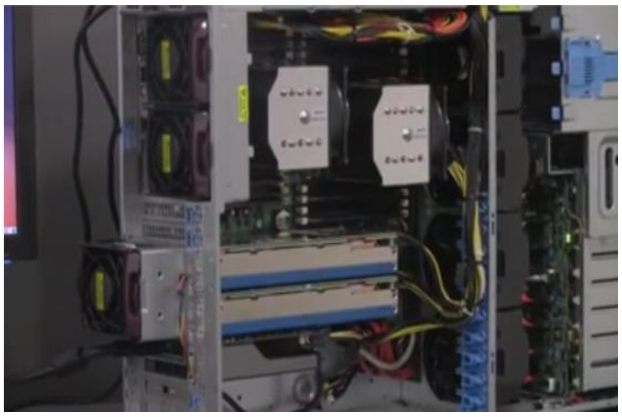

Figure 4. Design of Cooling Fan

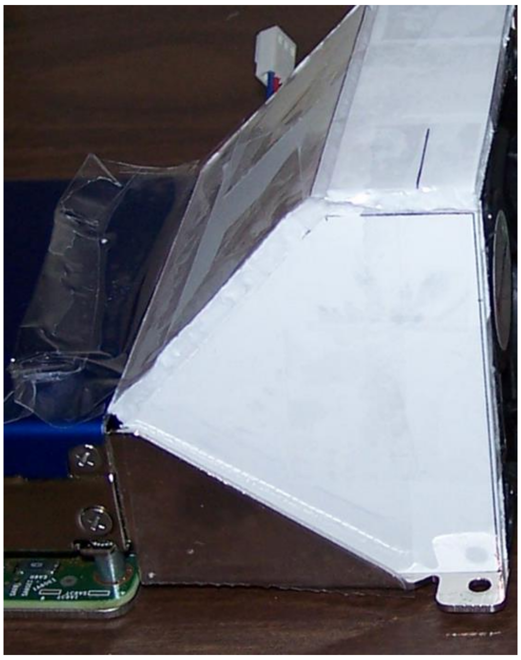

Figure 5. Funnel-Like Design of System

In Figure 4, two MIC cards are used as co-processors, and four Delta 12cm 12V 1.74A cooling fans are used for heat dissipation. Set the speed of fans maximize in BIOS to enhance the heat dissipation. Meanwhile, using more than two MIC cards can easily lead to exceeded thermal budget, in that the power consumption of MIC is too high. The four fans which encircled the co-processor are able to manufacture air convection, one of which is air-blast cools down the MIC co-processor continuously and directly. They let the MIC coprocessor stay peak performance continuously.

In Figure 5, additional fans are used for suction. According to the actual situation, we used additional card board to design the air duct of system to be funnel-like. Without affecting other equipment working, using additional low-power consumption fan induced draft from the internal system, better induced wind from the internal system. The efficiency of heat dissipation is greatly improved. Thus, inducing wind blindly from internal system is prevented, which may interfere with air convection and weaken the cooling effect.

With the above design, the heat dissipation issue is able to be solved and the stability of system can be improved effectively. In the storage section, a SSD--Plextor PX-AG256M6e (256GB) is selected, and it can increase the efficiency of reading and writing operations and reduces the power consumption of read and write elements from/to disk by 30W. Finally, the power consumption of the HPC system is 2839.52W in total.

5. Experimental Test and Analysis

5.1. Experimental Platform

(1) Hardware environment

In this paper, the test platform is designed and constructed based on high performance computing cluster of Three-River Source Data Analysis Center of Qinghai University. The power consumption of the test platform constructed is less than 3000W in total, with CPU+MIC as a heterogeneous computing mode.[17] It uses offload programming mode which CPUs dominate under assistance of MICs. The platform includes three Intel Xeon E5 CPU processor and two Intel Xeon Phi co-processor. The hardware configuration can meet the requirements of HPCG benchmark.

Table 1. Hardware Environment Configuration on Test Platform

Item	Name	Configuration
Server	Inspur NF5280M4	CPU: Intel Xeon E5-2680v3 x 2，2.5Ghz，12 cores Memory: 16G x8，DDR4，2133Mhz Hard disk: 1T SATA x 1 Power consumption estimation: E5-2680v3 TDP 120W, memory 7.5W, hard disk 10W
Accelerator Card	XEON PHI-31S1P	Intel XEON PHI-31S1P （57 cores, 1.1GHz, 1003GFlops, 8GB GDDR5 Memory） Power consumption estimation: 270W

(2) Software environment

The operating system taken in the test is RedHat Linux 7.2. In order to utilize fully computational capability of Intel co-processor, the Intel official MKL math library, ICC and Intel MPI are selected.[18] Specific parameters are shown in Table 2.

Table 2. Software Environment Configure on Test Platform

Category	Description	Version number
Operating System	Linux	RHEL 7.2
Math Library	Intel MKL	2015.0.090
Compiler	Intel Composer XE Suites	2015.0.090
MPI soft ware	Intel MPI	5.0.1.035
HPCG	HPCG	3.0
PBS	Torque	3.0.5

5.2. Optimization Results Analysis

5.2.1. Basic Approach

Firstly, the results of basis scheme are obtained. Change the matrix size, ranging from 8 to 128, and record test results which is obtained by porting HPCG metric to MIC side and run it relying solely on multithreading character of MIC. The test results show that if the routine does not use any acceleration approach, only by the advantage of multithreading character of MIC, the speedup of HPCG metric will not be obvious, just getting speedup in the range of 1.1~1.2. Sometimes, results obtained by running HPCG on MIC co-processor is even slower than that on CPU.

5.2.2. Parameter Tuning

Count the establishment time of sparse matrix in HPCG, the size of which has been growing from 16*16*16 to 128*128*128. The test results, evaluated on real data, show that the establish time of sparse matrix achieve extremely significant increase with increasing size of matrix. Even for the same size of matrix, the setup time may be measured with slightly perturbed. So the figures for the establishment of different size of matrix were calculated as the average of several repetitive runs.

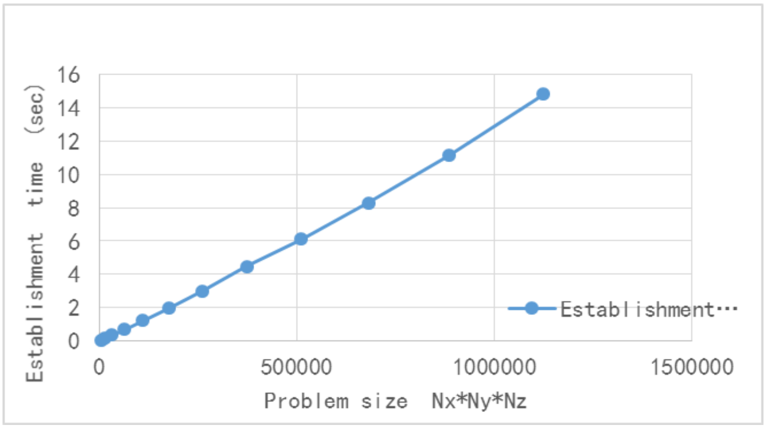

Figure 6. Establishment Time of Sparse Matrix

Figure 6 shows that, with the increasing size of the matrix, the set up time of sparse matrix grows significantly. For example, when the matrix size grow from $64 \times 64 \times 64$ to $104 \times 104 \times 104$, the size of matrix grows 4.291 times, the establishment time of sparse matrix also increases 4.932 times.

Adjusting the size of sparse matrix as described above, the HPCG benchmark runs on the CPU side. At first, the single CPU node is tested. Because the minimize size of matrix allowed in HPCG benchmark is 16, the size of matrix selected has been increased from 16 to 128. Then count the floating point peak performance measured from different matrix calculation module in HPCG respectively. Seen by the test, when the number of threads of one node is 20, the floating-computation speed is the highest. Details information are in the following Figure 7.

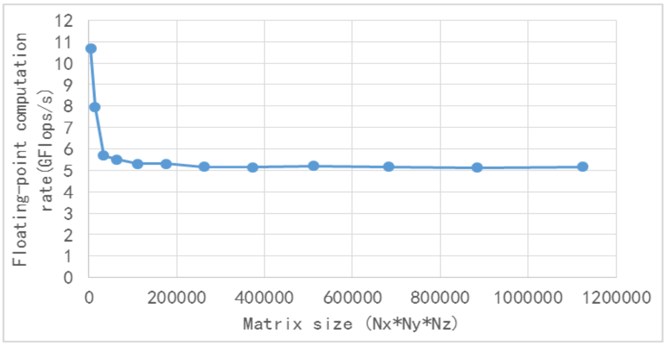

Figure 7. Floating-Point Performance in Different Matrix Size

From the Figure 7, when the size of matrix is $16 \times 16 \times 16$, the floating-point calculations rate reaches the peak, which is 10.6818 GFlops/s. With the increasing size of matrices, the floating-point computation speed does not increase, but has remained at about 5 GFlops/s.

(1) When the size of matrix is 16, the single node of CPU is not limited by memory capacity, which utilize the computing capacity fully, and achieve floating-point performance peak.

(2) With the increasing size of matrix, the floating-point computation rate measured by HPCG is remains at about 5 GFlops/s, instead of rising. It reflects that when the size of the matrices increases, although calculation related to solve the matrix equation increased significantly, it also leads to an increase in complexity of memory access patterns. The capacity of memory limits the increase of float-point computation rate of a single node,

which leads to the result that the float-point computation rate is measured no significant growth by HPCG Benchmark.[19]

Next, using the multi-node cluster accelerate the HPCG benchmark. Since it is measured in the single node that under the 20 threads condition, floating point calculation rate can reach the peak at the size of $16 \times 16 \times 16$、$24 \times 24 \times 24$ and $20 \times 20 \times 20$, so using the three size of matrices, which are in the 20 threads (allocation of threads between nodes is 7+7+8),24 threads (allocation of threads between nodes is 8+8+8) and 16 threads (allocation of threads between nodes is 5+5+6) conditions for testing, meanwhile, recording the floating-point computation rate measured by HPCG at different matrix size. Details are as follows in the Figure 8.

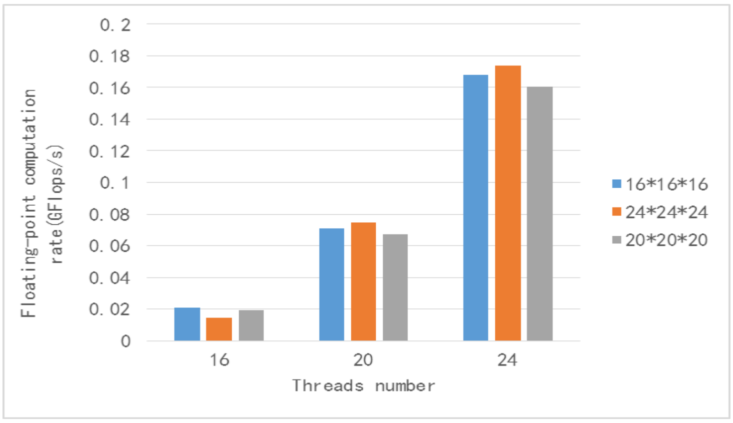

Figure 8. Floating-Point Performance in Cluster

The experimental results show that in cluster floating-point computation rate has significantly decreased in different matrix size, while the establishment time of large size of matrix increases obviously, which indicates that with the increasing number of the compute nodes, the program is able to build a sparse matrix more quickly, but because of scheduling the processes in different nodes, the communication latency between nodes , I/O operation, consumption related to process and overlapping of communication will increase obviously. The above all may greatly affects corresponding floating-point computation rate, resulting in the floating-point performance in cluster much lower than in a single node.

5.2.3. Acceleration of HPCG based on MIC

According to the above optimization of source code of HPCG based on MIC, modify the size of matrix from 16 to 128. Under the condition of 20 thread, the floating-point computation rate is recorded. The computational results are judged by the percentage of accelerate rate. Details are as follows in Figure 9.

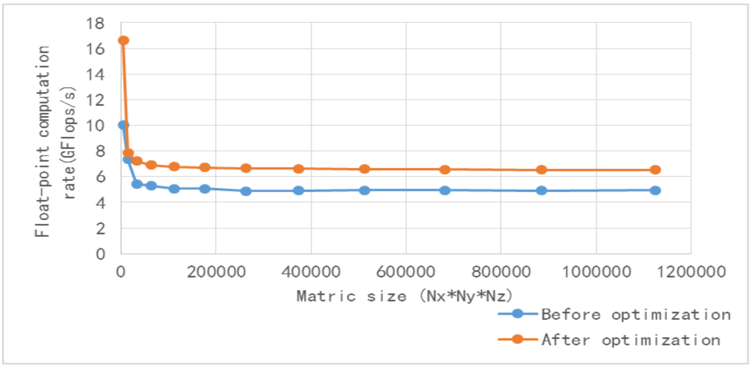

Figure 9. Floating-Point Performance after Optimization

According to the Figure 9, after the optimization of the source files of HPCG related to matrix operation, the performance has been greatly improved. When the size of matrix is $16 \times 16 \times 16$, the increasing percent of floating-point performance is the highest, which is 65.39526 %. With the increasing of the size of matrix, it is obvious that the floating-point rate increases, but optimized source file may substantially increase the percentage about 30 %. The main reasons are the following aspects:

(1) When the size of matrix is 16, the valid computational time related to matrix operation increases. Due to the small size of the matrix, it has adequate streaming memory systems, so the optimization effect is the most obvious.

(2) With the increasing of the size of matrix, more numbers of threads on MIC are put into calculation. However, when the numbers of threads increase to a certain degree, the competition of memory between threads will happen, so the memory capacity will limit the acceleration percent. In another word, valid computation is proportional to the memory consumption[20] , which may lead to the fact that the acceleration percentage keeps at a constant level.

The optimized metric results for the particular platform, evaluated on real data, shows that the speedup of HPCG is obvious. The floating point computation rate has been significantly improved. To some extent, the drawback of insufficient computation capability of CPU is greatly alleviated, and the peak experimental value is also improved significantly.

5.2.4. The Influence of Heat Dissipation to Computation Capability of MIC

The innovative heat dissipation technologies is adopted in the article, which plays an important role in maintaining floating-point performance peak. The following content is taking the size of matrix is $16 \times 16 \times 16$ as an example, and record the floating-point performance peak of MIC at different times.

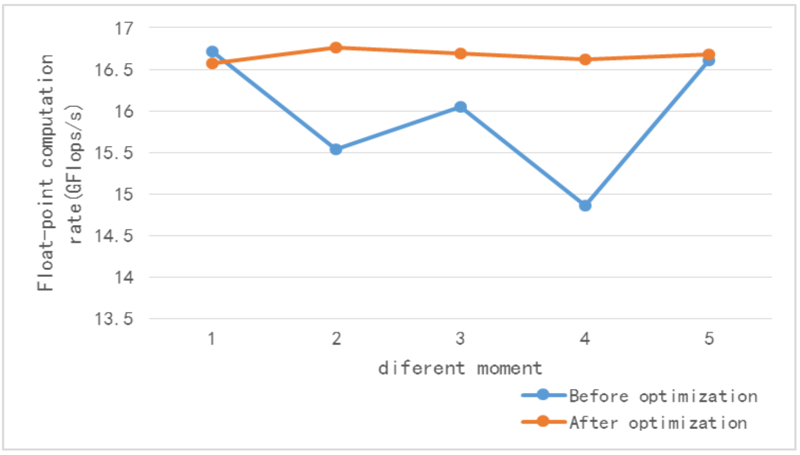

Figure 10. Floating-Point Performance of Heat Dissipation

As is shown in Figure 10, before the heat treatment, the peak performance of MIC system measured by HPCG is unstable and fluctuated obviously, which may cause the wrong computational results. HPC system will have great volatility, which will lead to a significant decline in floating-point performance of MIC.

After using the above method to solve the heat dissipation, the floating point calculation rate measured by HPCG at different times almost stays at the same rate, and the peak of system reached a stable level of about 16.6 GFlops/s, which shows that the design of heating dissipation is feasible, [21]and it can maintain the computing performance of MIC co-processor, and it will not occur that the calculation errors or a significant decline in performance due to the poor heat dissipation. It proves that heat dissipation system is essential in the stability of the peak of system.

6. Conclusion and Future Work

In this paper, the HPCG benchmark is optimized for the MIC co-processor architecture. Using a variety of optimization techniques, mainly optimizing deeply the source code related to matrix operations, the results, evaluated on real data, show the feasibility and effectiveness of the optimization. In this paper, the HPCG is mainly tested on a single MIC co-processor. If more than one cards are used under test, we need to consider the communication between different nodes[22]. If not handled properly, it is likely to offset the efficiency improvements after optimized, which may cause that multi-node results to be even worse than the results of a single node[23]. Therefore, the future work will focus on the further measurement and optimization for the interaction between multiple nodes.

Acknowledgments

This paper is partially supported by The National Natural Science Foundation of China (No. 61363019, No. 61440021, No. 61272087 and No.61073008); National Natural Science Foundation of Qinghai Province (No. 2014-ZJ-718, No. 2015-ZJ-725).

References

[1] J.J. Dongarra, P. Luszczek and A. A. Petitet, "The LINKPACK Benchmark: Past, Present, and Future", Concurr Comp-Pract E., vol. 9, no. 15, (**2003).**

[2] X. Wang and S. Li, "HPCC a Memory Access Model Oriented Benchmark – a Potential Test Method to Replace HPL in TOP500", MINI-MICRO SYSTEMS, vol. 5, no. 27, **(2006)**. In Chinese.

[3] J. .Dongarra, A. Michael, Heroux and Luszczek, "A New Metric for Ranking High Performance Computing Systems", National Science Review Advance Access, vol. 1, **(2016)**.

[4] A. Michael, Heroux, J. Dongara and P. Luszczek, "R. Hpcg benchmark technical specification", Technical report, vol. 10, **(2013)**.

[5] J. Dongarra and M. A. Heroux, "Hpcg: One year later", ISC 2014, Leipzig, Germany, **(2014)**.

[6] C. Wang and S. Zhu, "A Design of Fpga-Based System For Image Processing", Review of Computer Engineering Studies, vol. 1, no. 2, **(2015)**.

[7] Y. Fan, "Semantic Annotation and Storage for Tourism Information", Review of Computer Engineering Studies, vol. 2, no. 2, **(2015)**.

[8] P. Shen, H. Wang, Z. Meng, Z. Yang, Z. Zhi, R. Jin and A. Yang, "An Improved Parallel Bayesian Text Classification Algorithm", Review of Computer Engineering Studies, vol. 1, no. 3, **(2016)**.

[9] M.M. Guo, S.L. Yang, H. Yan, L.F. Kan and B. Yang, "Mobile Video Alarm System Based On Cloud Computing", Review of Computer Engineering Studies, vol. 2, no. 1, **(2014)**.

[10] B. Shen, G.-Y. Zhang, S.-H. Wu, X.-W. Lu and Q. Zhang, "Research of Offload Parallel Method Based on MIC Platform", Computer Science, vol. 6A, no. 41, **(2014)** in Chinese.

[11] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu and Y. Wang, "MIC High Performance Programming Guide", China WaterPower Press, Beijing, **(2012)** in Chinese.

[12] M. Xiong and Y.X. Wang, "Parallel optimization of the seismic wave PKTM algorithm on CPU + MIC heterogeneous platform", Computer Engineering & Science, vol. 1, no. 37, **(2015)** in Chinese.

[13] H. Zheng, A.P. Song and Z. Wu, "MIC-GaBp: A New Algorithm To Solve Large Scale Sparse Linear System", Journal on Numerical Methods and Computer Applications, vol. 1, no. 36, (**2015**) in Chinese.

[14] Z. Mo, "J. High-performance programming frameworks for numerical simulation", Computer Science, vol. 1, no. 3, (**2016**) in Chinese.

[15] A. Zhang, H. An, W. Yao, X. Jiang and F. Li, "Efficient Sparse Matrix-vector Multplication on Intel Xeon Phi", Journal of Chinese Computer Systems, vol. 4, no. 37, (**2016**) in Chinese.

[16] W. Yao, Y. Du, Q. Wu and C. Yang, "Research of LARED-P on Intel Xeon Phi", Computer Engineering & Science, vol. 5, no. 36, **(2014)** in Chinese.

[17] Y. Song, L. Wang and X. Meng, "An Accelerated Ray Tracing Algorithm for the Intel Xeon Phi^{TM} Architectures", Journal of Computer-Aided Design & Computer Graphics, vol. 12, no. 27, **(2015)** in Chinese.

[18] J. Qi, K. Li, C. Yang and Y. Du, "Accelerating 3D GVF computation on Xeon Phi using stencil optimization", Computer Engineering & Science, vol. 8, no. 36, **(2014)** in Chinese.

[19] B. Song, J. Zhou, C. Hua, X. Liu and X. Zhou, "Design And Implementation Of Mrg32k3a Parallelisation Based On Mic", Computer Application and Software, vol. 2, no. 33, **(2016)** in Chinese.

[20] Y. Che, L. Zhang, Y. Wang, C. Xu and X. Cheng, "Open MP Performance Analysis of CFD Application on Intel Multicore and Manycore Architectures", Journal of Frontiers of Computer Science and Technology, vol. 9, no. 10, **(2015)** in Chinese.

[21] X. Yang, J. Jin, P. Wang G. M. Zhao and Y. Kang, "Wave equation prestack depth migration based on Xeon Phi platform", vol. 5, no. 37, **(2015)** in Chinese.

[22] F. Gao and Y. Liu, "Acceleration of video caption retrieval algorithm based on Intel MIC", Computer Engineering & Science, vol. 4, no. 37, **(2015)** in Chinese.

[23] M. Fang, W. Zhang, L. Zhang, L. Zeng, X. Liu and L. Yin, "A 3—Dimension Monte Carlo simulator for semiconductor devices based on many integrated core", Computer Engineering & Science, vol. 4, no. 37, **(2015)** in Chinese.