Performance Comparison between OpenMP and MPI on IA64 Architecture

Lin Qi, Meiming Shen, Yongjian Chen, and Jianjiang Li

Institute of High Performance Computing, Tsinghua University
Qilin97@mails.tsinghua.edu.cn

Abstract. In order to evaluate performances of OpenMP and MPI on 64-bits shared memory architecture, we made a comparison between OpenMP and MPI on IA64 SMP with NPB2.3 benchmark suite (excluding IS kernel). Assisted by Vtune(TM) Performance Analyzer 7.0, we raised comparisons in execution time, detailed performance data and overheads between these two paradigms, and managed to optimize some of these kernels.

1 Introduction

The opportunities of shared-memory programming paradigms such as OpenMP for parallel programming on distributed shared memory architectures like SMP clusters are still not clear. Although experiments of mixed mode programming for SMP clusters, often using MPI and OpenMP, have been done on several benchmarks and applications, the results don't directly deny or encourage such combinations. To achieve the final goal, our current works focus on investigating this problem by studying the overhead characteristics of OpenMP and MPI.

Recently, detailed overhead analysis technologies have shown that the overhead ratios and characteristics of OpenMP and MPI programs are quite different, which means that the program models used by MPI and OpenMP may affect program behaviors in different ways, and the combination of these two models may compensate each other thus to reduces overheads. So our works concentrated on how to use overhead analysis technologies to identify overheads in both models in order to find out opportunities for mixed mode paradigms on SMP clusters. After trying several software tools we finally used Vtune from Intel to assist our performance analysis work.

2 Experimental Conditions

2.1 Programming Model

OpenMP

Along with the appearance of scalable shared-memory multiprocessor, people realized that there is more direct and simple way to program on SMP. OpenMP was

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 388–397, 2004.

proposed for a transplantable standard to substitute for MPI. The original objective of OpenMP was to provide a transplantable and scalable programming mode on shared memory platform and to offer a simple and flexible programming interface for development of parallel applications. It is composed of directives, environmental variables and runtime library. It supports incremental parallelization so it is very easy to transform a serial application to a parallel one and gain considerable performance enhancement on shared memory architecture.

MPI

In a long time people considered the only way to develop parallel programs is utilizing message passing model. Message passing programming model is a widely used programming method explicitly controlling parallelism on distributed memory system. All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs. The most familiar programming style is SPMD. MPI (Message Passing Interface) defines a whole set of functions and procedures to implement message passing model.

2.2 Methodology

We mainly concentrated on program performances in this paper. Each of NPB2.3 kernels was tested on four threads or processes when it monopolized the IA64 SMP. At first we measured the execution time of each kernel with OpenMP and MPI versions respectively. We call the execution time measured by function w_time wall clock. They are listed in Fig. 1. Then we used Vtune(TM) Performance Analyzer 7.0 to obtain more detailed performance data in each thread or process of the kernel. In this paper we focused on CPU_CYCLE events, CPU_CYCLE %, IA64_INST_RETIRED-THIS events, IA64_INST_RETIRED-THIS % and IPC (IA64_INST_RETIRED-THIS per CPU_CYCLE). These performance data obtained by Vtune were on function level, that is, they gave amount of CPU cycles and instructions in every function. We classified these functions into two categories, one is explicitly existent functions programmed by developers and the other is runtime library calls which are implicitly called by the system, we call them runtime library overheads. Theoretically, with the same algorithm the execution times of explicitly existent functions should be similar between OpenMP and MPI versions, and the different parts should come from the runtime library overheads.

2.3 Benchmark

We chose NPB2.3 suite excluding kernel IS to launch our experiments. We parallelized NPB2.3 serial version with OpenMP directives in Fortran referring to the OpenMP version in C by NASA. Noticing that the corresponding MPI version is highly tuned in Fortran, comparisons between them should be challenging. We tested all of these kernels in their default class. BT, LU and SP are in class W, the others are in class A.

2.4 Platform

Our experiment environment was established on IA64 platform with a single SMP. With total 4 GB memory, it had four Itanirum2 processors with 900MHz which were connected by the system bus with bandwidth up to 6.4 GB/s. For each processor, the caches were 16 KB of L1I, 16KB of L1D, 256KB of L2 and 3MB of L3 on die with 32 GB/s bandwidth. The Itanium processor's EPIC features made higher instruction level parallelism up to 6 instructions per cycle.

We used mpich-1.2.5 for the MPI versions. This shared memory version of the MPI library made performance of MPI versions on SMP scale as high as it could.

In our experiments all of the kernels were compiled with intel's efc and the compiler options are configured as:

F77 = efc –openmpP –O3 –g –w for OpenMP version; MPIF77 = mpif90 –O3 for MPI version. The –g option was to obtain debug information for linux binary during Vtune's source drill down.

3 Understanding the Performance Results

3.1 Execution Time Comparison

Fig. 1 demonstrates the seven NPB2.3 kernels's performance differences between MPI and OpenMP on the IA64 SMP.

Among the seven kernels, BT, CG, EP and FT exhibited a slightly superior performance with OpenMP version compared with their MPI version; two versions of SP showed almost the same performance, while LU and MG obviously did better with MPI. Being highly tuned with shared memory version of MPI library, these seven kernels all showed fairly high performance with MPI on SMP. However, except LU and MG the other five kernels also exhibited competitive performances with OpenMP. Optimizing LU and MG will be a part of our works in the near future.

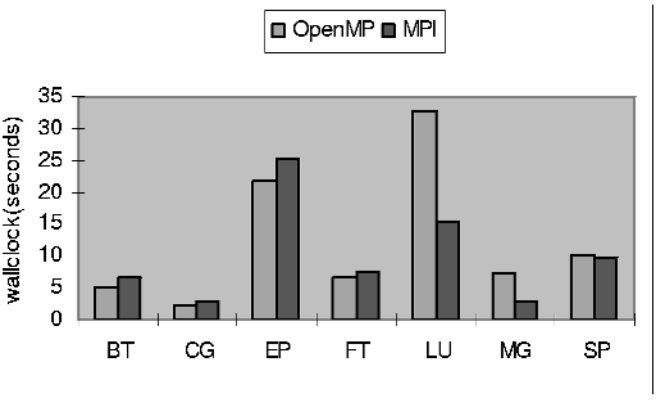

Fig. 1. Execution time of NPB2.3 suite on IA64 SMP

Table 1. Execution time table

	OpenMP(s)	MPI(s)
BT	5.06	6.63
CG	2.28	2.80
EP	21.61	25.07
FT	6.53	7.66
LU	32.81	15.44
MG	7.14	2.97
SP	9.94	9.67

3.2 Comparison of Detailed Performance Data between OpenMP and MPI

To find the causes of such performance distinction, we used Vtune(TM) Performance Analyzer7.0 to measure more detailed data which could indicate program and system behaviors in both programming paradigms.

In figures below, we showed two basic events and their proportion of BT and MG. They represented different performance comparison results. The basic events were CPU_CYCLE events, IA64_INST_RETIRED-THIS events and IPC respectively. Because we tested every kernel on four threads or processes only, the 0#, 1#, 2# and 3# mean to four threads or processes of OpenMP or MPI. The precise data were listed below in the tables following the corresponding figures.

Upon further analysis of Vtune results, we found although the workloads were not distributed averagely OpenMP performs better than MPI. According to the first three threads, retired instructions of OpenMP version were less than those of MPI version. Therefore with the similar IPC, the number of CPU cycles was also less than that of MPI version. However the fourth thread has much more retired instructions than the first three. Vtune Analyzer collected 9.150E+9 retired instruction events, outweighing 8.003E+9, retired instruction events of the 3# thread of MPI version. In this thread __kmp_wait costed 4.088E+9 instructions, 2.206E+9 CPU cycles, and it accounted for 44.69% of the total retired instructions, 42.60% of the total CPU cycles. Because the IPC of __kmp_wait in this thread was 1.85, it pulled the average IPC of all functions in 3# thread up to 1.767. Finally the higher IPC makes the 3# thread's CPU_CYCLES events of OpenMP version less than that of MPI version.

For MG, IPCs of OpenMP code were higher than IPCs of MPI code which showed better instruction level parallelism with OpenMP. However we also observed that its workloads were not very balanced among the four threads as the MPI version did as it was shown in Table 3. More instructions and lower IPCs in function reside(), vranlc(), zran3() of 3# thread induced more CPU cycle events in this thread and longer execution time of the whole program.

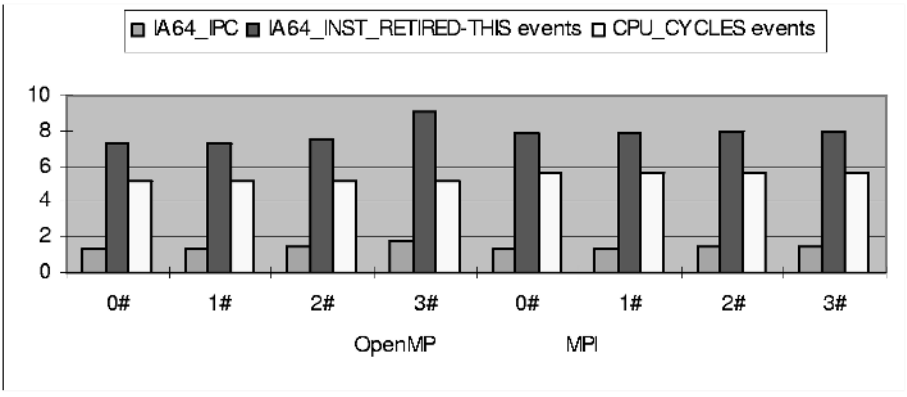

Fig. 2. BT – class W

Table 2. BT – class W

version	proc/thr No.	IPC	INST_RETIRED	CPU_CYCLES
OpenMP	0#	1.395	7.255E+9	5.202E+9
	1#	1.396	7.273E+9	5.209E+9
	2#	1.450	7.531E+9	5.193E+9
	3#	1.767	9.150E+9	5.179E+9
MPI	0#	1.395	7.854E+9	5.629E+9
	1#	1.403	7.891E+9	5.624E+9
	2#	1.421	7.958E+9	5.602E+9
	3#	1.432	8.003E+9	5.588E+9

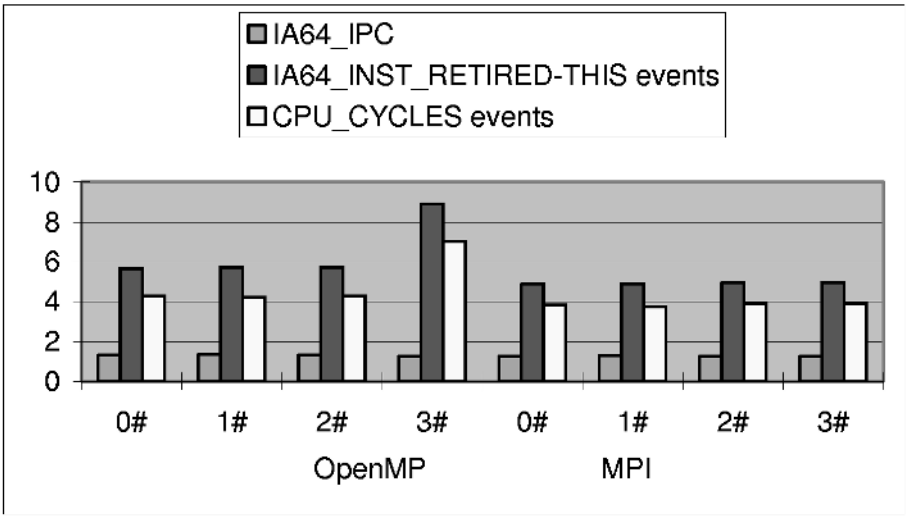

Fig. 3. MG - class A

Table 3. MG – class A

version	proc/thr No.	IPC	INST_RETIRED	CPU_CYCLES
OpenMP	0#	1.319	5.662E+9	4.292E+9
	1#	1.359	5.729E+9	4.216E+9
	2#	1.331	5.730E+9	4.306E+9
	3#	1.267	8.891E+9	7.016E+9
MPI	0#	1.275	4.896E+9	3.841E+9
	1#	1.303	4.908E+9	3.765E+9
	2#	1.272	4.955E+9	3.896E+9
	3#	1.267	4.971E+9	3.922E+9

3.3 Breakdowns of Execution Time

When testing with a benchmark program, Vtune provides performance events on function level. These functions are not only composed of those which are programmed by developers but also some runtime library functions which are implicitly called. We call the latter ones overheads.

For example, when we tested with kernel BT, we obtained performance data of many functions, lhsx, lhsy, lhsz, compute_rhs, x_backsubstitute and so on, all of which could be found in BT's source code. But there were still some other functions such as MPI_SHMEM_ReadControl, MPI_Waitall, kmp_wait, kmp_yield which could not be found in source code. So we classified all functions into algorithm functions and overheads. Through quantitative analysis it was clear that MPI_SHMEM_ReadControl and kmp_wait were the main overheads of MPI and OpenMP respectively. According to measurement results of all NPB2.3 kernels, CPU cycles costed by MPI_SHMEM_ReadControl were all less than, or at most similar with those costed by __kmp_wait. This was because most of overheads in MPI were resulted of communication and synchronization among processes, for example mpi_send, mpi_receive, mpi_wait, but all of these operations were explicitly controlled, so they were not included in MPI_SHMEM_ReadControl. On the other hand many causes generated OpenMP program's overheads, for example synchronization, load imbalance and improper memory access strategies. Obviously most of overheads in OpenMP rose from synchronization and load imbalance, for example omp_barrier and single, critical operations which resulted in the other threads were idle while only one thread was busy. Most of these overheads were included into __kmp_wait. Anyway, generally when overhead's quantity is large, it means the program's performance is low and should be optimized.

Fig. 4 shows execution time breakdowns of LU, the overheads of its OpenMP version account for over 50% of its execution time. We are trying to find the sources and optimize it. It will be discussed in the future papers.

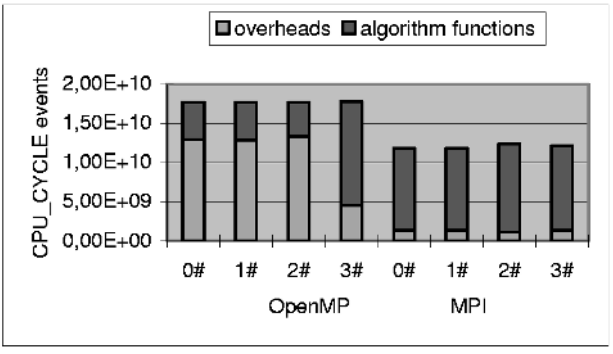

Fig. 4. Execution time breakdowns of LU with OpenMP and MPI

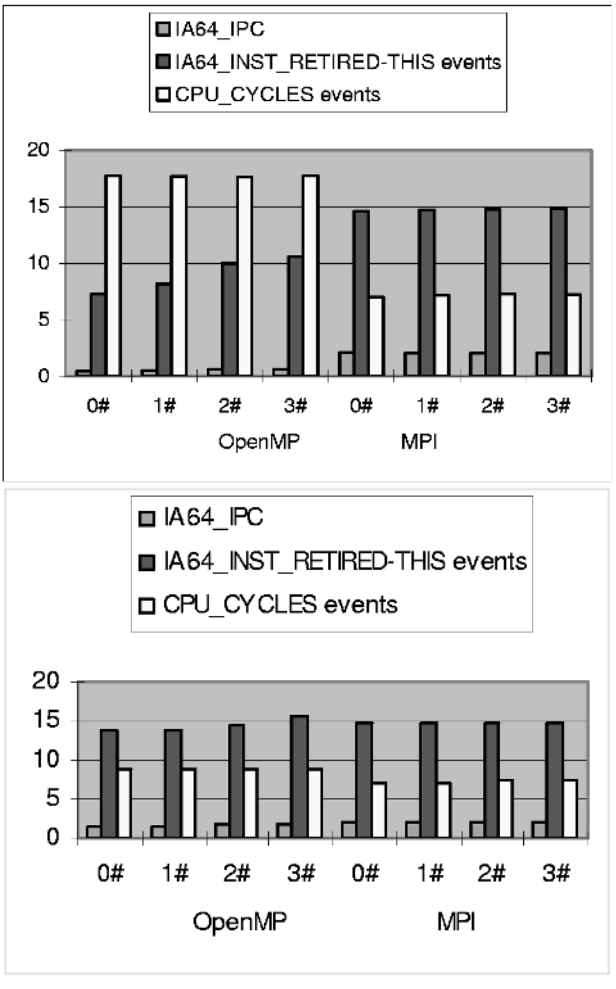

Fig. 5. Performance before and after optimization

3.4 Performance Enhancement from Optimization

The upper one of Fig. 5 is performance exhibition of SP OpenMP version before optimization. For OpenMP version, the IPC were very low and there were obviously much more CPU cycle events measured. We observed that the CPU cycle events of __kmp_wait were 7.53E+9, 7.31E+9, 8.66E+9 and 4.48E+9 in every thread respectively, being the most time-consuming function. And functions such as x_solve, y_solve, z_solve, lhsx, lhsy and lhsz accounted for the main part of execution time. The optimization focus on the six functions mentioned above and the optimizing method was simply called loop inversing and combination. We took the x-series functions for example:

x_solve.f:
The original code segment is:

```
do i = 0, grid_points(1)-3
    i1 = i + 1
    i2 = i + 2
!$omp do
    do k = 1, grid_points(3)-2
      do j = 1, grid_points(2)-2
        fac1 = 1.d0/lhs(i,j,k,n+3)
        lhs(i,j,k,n+4) = fac1*lhs(i,j,k,n+4)
        lhs(i,j,k,n+5) = fac1*lhs(i,j,k,n+5)
        …
      end do
    end do
!$omp end do
  end do
```

The optimized code segment is:

```
!$omp do
  do k = 1, grid_points(3)-2
    do j = 1, grid_points(2)-2
      do i = 0, grid_points(1)-3
        i1 = i + 1
        i2 = i + 2
        fac1 = 1.d0/lhs(i,j,k,n+3)
        lhs(i,j,k,n+4) = fac1*lhs(i,j,k,n+4)
        lhs(i,j,k,n+5) = fac1*lhs(i,j,k,n+5)
        …
!$omp end do
```

lhsx.f:
The original code segment as below:

```
do k = 1, grid_points(3)-2
   do j=1, grid_points(2)-2
!$omp do
     do i = 0, grid_points(1)-1
```

```
        ru1 = c3c4*rho_i(i,j,k)
        cv(i) = us(i,j,k)
        rhon(i) =
          dmax1(dx2+con43*ru1,dx5+c1c5*ru1,dxmax+ru1,dx1)
      end do
!$omp end do
!$omp do
      do i = 1, grid_points(1)-2
    …
      end do
!$omp end do
end do
```

This code segment was optimized as follows:

```
!$omp do
  do k = 1, grid_points(3)-2
    do j = 1, grid_points(2)-2
      do i = 0, grid_points(1)-1
      rhon(i,j,k) = dmax1(dx2+con43*c3c4*rho_i(i,j,k),
dx5+c1c5*c3c4*rho_i(i,j,k),dxmax+c3c4*rho_i(i,j,k),dx1)
      end do
      do i = 1, grid_points(1)-2
     …
      end do
!$omp end do
```

The nether one of Fig. 5 is the performance exhibition after optimization. The IPC of each thread was lifted, from 0.41, 0.46, 0.57, 0.60 to 1.57, 1.58, 1.66, 1.78 respectivly. The CPU cycle events of __kmp_wait were reduced to 1.7E+9, 1.7E+9, 2.7E+9 and 7.1E+8 respectively. And the execution times of those time-consuming functions were also cut down to some extent. Because of the enhancement of IPCs, although our optimization made retired instructions increased CPU_cycle events of the whole program were decreased.

4 Conclusion and Future Work

In this paper we made a comparison between OpenMP and MPI on IA64 SMP with NPB2.3 benchmark suite excluding IS. Our study shows that with simple OpenMP directives people can gain as good performance as shared-memory MPI on IA64 SMP. The merits of easier programming and satisfying performance make OpenMP a profitable paradigm for 64-bits shared memory architectures.

Furthermore there are some noteworthy questions in OpenMP. Although the easy programming is the obvious merit of OpenMP, improper programming is more likely to happen and that often leads to inferior performance. Another common problem is the imbalanced load distribution in OpenMP program. Furthermore, whether OpenMP can gain superior performance than shared-memory MPI on SMP is still an unsolved topic since OpenMP was proposed especially for shared memory architecture.

Our future work will continue to focus on performance evaluation and analysis of OpenMP and MPI in order to explore the possibilities in mixed mode programming on SMP cluster.

References

1. Geraud Krawezik and Franck Cappello, Performance comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors, SPAA'03, June 7-9, 2003.
2. F. Cappello and D. Etiemble, MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. Procs. of the international Conference on Supercomputing 2000: High-Performance Networking and Computing (SC2000), 2000.