Performance Analysis of Pure MPI Versus MPI+OpenMP for Jacobi Iteration and a 3D FFT on the Cray XT5

Glenn Luecke, Olga Weiss, Marina Kraeva,
James Coyle, James Hoekstra, *Iowa State University's*
High Performance Computing Group

ABSTRACT: *Today many high performance computers are collections of shared memory compute nodes with each compute node having one or more multi-core processors. When writing parallel programs for these machines, one can use pure MPI or various hybrid approaches using MPI and OpenMP. Since OpenMP threads are lighter weight than MPI processes, one would expect that hybrid approaches will achieve better performance and scalability than pure MPI. In practice this is not always the case. This paper investigates the performance and scalability of pure MPI versus hybrid MPI+OpenMP for Jacobi iteration and for a 3D FFT on the Cray XT5.*

KEYWORDS: MPI, OpenMP, hybrid programming, Cray XT5

1. Introduction

Since fast parallel execution is a critical requirement for many scientific applications, it is important to parallelize using methods that achieve the highest possible performance. In the past, using the MPI [1] passing library gave the best possible performance for distributed memory parallel computers. Today, most high performance computers, including the Cray XT5, are a collection of shared memory nodes interconnected by a communication network for message passing. This architecture allows one to parallelize applications using either only MPI or using a combination of MPI and OpenMP [2, 3]. Programming with both MPI and OpenMP is often called "hybrid" programming, see [3, 4]. Numerous papers have been written, e.g. [4, 5, 6, 7], comparing the performance of various applications using pure MPI and MPI+OpenMP. The purpose of this paper is to investigate the performance of pure MPI versus MPI+OpenMP for Jacobi Iteration and for a 3D FFT on the Cray XT5.

The Cray XT5 has been designed for maximum performance and scalability [8] with each compute node consisting of two 2.4 GHz AMD Istanbul Opteron processors (often referred to as "sockets") with each 6-core processor having a 6 MB shared L3 cache and 16 GB of memory. Thus, each compute node has 32 GB of memory and is a cache coherent Non-Uniform Memory Access (ccNUMA) node so that memory accesses from one socket to the memory of the other socket will be slower than memory accesses to the socket's local memory [4]. Therefore, to achieve maximum performance on a ccNUMA node, the method of parallelization should use memory accesses local to the executing socket. When an application is parallelized using MPI with one MPI process per socket instead of 1 MPI processes per node, memory accesses are local to the executing socket since all data is private to the executing MPI process, assuming the MPI process is not moved from one socket to another during execution.

When running an OpenMP program on a ccNUMA node, data is normally distributed using "first touch", i.e. the data will reside in the memory of the socket where the data is first used. The primary method of parallelization in OpenMP is the parallelization of loops. Loop iterations are scheduled for execution among threads according to the scheduling method specified in the OpenMP program, i.e. static, dynamic or guided (a "chunksize" can be

specified for each of these options). When an OpenMP program is executed on more than one socket, using dynamic or guided schedule will lead to the data distribution performance problem due to non-local data accesses. When an OpenMP program is executed on a single socket instead of on multiple sockets, then the data distribution performance problem goes away and one is free to use any of the scheduling options without a performance penalty due to non-local data accesses. The programming environment for the Cray XT5 allows the scheduling of MPI programs with one or more MPI processes for each socket and thus guaranteeing local memory accesses for the OpenMP in the hybrid program. Of course, performance of an application involves many different factors and data distribution on ccNUMA nodes is only one of these factors.

2. The Jacobi Iteration

The Jacobi iteration used in this paper is based on the description of the Jacobi iteration found in chapter 2 of volume 1 of [1] that uses the mpi_sendrecv routine for parallelization. Data is distributed using the 1D block partitioning shown in Figure 2.4 on page 58. This Jacobi iteration comes from solving the 2D Laplace equation in a rectangle with values of the solution known on the four boundaries.

Let p be the number of MPI processes used and m = n/p. For simplicity, we assume p divides n evenly. Let "dp" be mpi_double_precision, "rank" be the rank of the executing MPI process and let "left"/"right" be the MPI process to the left/right of the executing MPI process [1, p66]. The following is the Fortran code segment of the MPI parallelization of the Jacobi iteration. Notice there are stride one array accesses in the innermost loop. The time reported is the maximum of the times for each MPI process.

```
! The MPI Jacobi Iteration
call mpi_barrier(mpi_comm_world, ierror)
t = mpi_wtime()
do iter = 1, niter
   do j = 1, m
      do i = 1, n
         B(i,j) = 0.25d0*(A(i-1,j)+A(i+1,j) + A(i,j-1) +A(i,j+1))
      enddo
   enddo
   do j = 1, m
      do i = 1, n
         A(i,j) = B(i,j)
      enddo
   enddo
   call mpi_sendrecv(B(1,m), n, dp, right, tag, A(1,0), n, &
             dp, left, tag, mpi_comm_world, status, ierror)
   call mpi_sendrecv(B(1,1), n, dp, left, tag, A(1,m+1), n, &
          dp, right, tag, mpi_comm_world, status, ierror)
enddo
time = mpi_wtime() - t
call mpi_barrier(comm, ierror)
call mpi_reduce(time, max_time, 1, dp, mpi_max, 0, &
                comm,ierror)
```

Notice that the MPI communication exchanges vectors of length n for each MPI process. Thus, ignoring contention on the communication network, the wall-clock time to perform the MPI communication should be independent of the number of MPI processes used.

The following describes version 1 of the hybrid Jacobi iteration. Notice that a parallel region is created for each iteration and that the MPI communication is performed outside the parallel region. A and B must be declared as shared since they are both needed for the MPI communication outside the parallel region.

```
! Version 1: hybrid Jacobi iteration
do iter = 1, niter
!$omp parallel shared(A, B, m) private(i, j)
!$omp do schedule(runtime)
     do j = 1, m
        do i = 1, n
           B(i,j) = 0.25d0*(. . .)
        enddo
     enddo
!$omp end do
!$omp do schedule(runtime)
        do j = 1, m
           do i = 1, n
              A(i,j) = B(i,j)
           enddo
        enddo
!$omp end do
!$omp end parallel
   call mpi_sendrecv(B(1,m), n, dp, ..., A(1,0), n, dp,...)
   call mpi_sendrecv(B(1,1), n, dp, ..., A(1,m+1), n, dp, ...)
enddo
```

Another way to write the hybrid program is to create a single parallel region that contains all iterations including all MPI communication. Since the MPI communication is within a parallel region but not from multiple threads, one should call mpi_init_thread as shown below and the environment variable MPICH_MAX_THREAD_SAFETY should be set to 'serialized'.

Since the A = B loop does not change the values of B and does not involve the 0 and m+1 columns of A, it is okay to use the nowait clause on the second parallel loop. This should increase performance when using the **!$omp single** directive immediately following the loop.

```
!  Version 2:  hybrid Jacobi Iteration
  call mpi_init_thread(mpi_thread_serialized, provided, ierror)
   . . .
!$omp  parallel shared(A, B, m) private(i, j)
do iter = 1, niter
!$omp  do schedule(runtime)
         do j = 1, m
            do i = 1, n
                B(i,j) = 0.25d0*(. . .)
            enddo
         enddo
!$omp  end do
!$omp  do schedule(runtime)
         do j = 1, m
            do i = 1, n
                A(i,j) = B(i,j)
            enddo
         enddo
!$omp  end do nowait
!$omp  single
        call mpi_sendrecv(B(1,m), n, dp, ..., A(1,0), n, dp,...)
        call mpi_sendrecv(B(1,1), n, dp, ..., A(1,m+1), n, dp,...)
!$omp  end single ! implicit barrier
enddo
!$omp  end parallel
```

Since using private instead of shared variables will sometimes increase performance, we wrote version 2 of the hybrid Jacobi with B private (using only the size needed for each thread to conserve memory). The performance of the private B version was about the same as the shared B version so we only present performance results when B is shared.

Notice that the two calls to mpi_sendrecv are independent of each other and therefore can be executed in parallel. This can be accomplished using the **!$omp sections** directive as shown below. To execute version 3 of the hybrid Jacobi, one must link with the thread safe version of the MPICH2 library, libmpich_threadm; one must call mpi_init_thread as shown below and the environment variable MPICH_MAX_THREAD_SAFETY must be set to 'multiple'.

```
!  Version 3:  hybrid Jacobi Iteration
  call mpi_init_thread(mpi_thread_multiple, provided, ierror)
. . .
!$omp  parallel shared(A, B, m) private(i, j)
do iter = 1, niter
!$omp  do schedule(runtime)
         do j = 1, m
            do i = 1, n
                B(i,j) = 0.25d0*(. . .)
             enddo
         enddo
!$omp  end do
!$omp  do schedule(runtime)
         do j = 1, m
            do i = 1, n
                A(i,j) = B(i,j)
            enddo
         enddo
!$omp  end do nowait
!$omp  sections
!$omp  section
        call mpi_sendrecv(B(1,m), n, dp, ..., A(1,0), n, dp,...)
!$omp  section
        call mpi_sendrecv(B(1,1), n, dp, ..., A(1,m+1), n, dp,...)
!$omp end sections ! implicit barrier
enddo
!$omp  end parallel
```

3. Jacobi Iteration Performance Results

All runs were made stand alone on Cray's internal XT5 named "koi" and the minimum of 10 trials is reported. Since the XT5 used for this paper had Istanbul processors, the "xtpe-istanbul" module was loaded. The problem size, n, was taken to be 24*1024, i.e. large enough to be interesting, small enough so the problem would fit into the memory of two nodes and a multiple of 12 so all 12 cores on each node would have the same amount of computation. The number of OpenMP threads used for the hybrid programs was always equal to the number of cores available to each MPI process. For example, when using 2 MPI processes per node, each MPI process used 6 OpenMP threads. The number of iterations, niter, was taken to be 300 so that the total time for all iterations could be accurately measured (with niter=300, there was little variation in timings). The following options were used on the aprun command:

- The "-cc cpu" option was used to "bind processing elements to CPUs", i.e. to bind MPI processes and OpenMP threads to cores. This is sometimes called "CPU affinity".
- The "-ss" option was used to ensure processing elements only allocate memory local to its assigned NUMA node (i.e. socket). This is sometimes called "memory affinity".
- The "-N #" option was used to indicate the number, #, of MPI processes for each compute node.
- The "-S #" option was used to indicate the number, #, of MPI processes per socket.
- The "-n #" option was used to indicate the number, #, of MPI processes used when executing the application.
- The "-d #" option was used to indicate the number, #, of OpenMP threads used for each MPI process.

All programs were run using the PGI compiler. To be able to run the hybrid programs, it was necessary to setenv OMP_STACKSIZE 1G. The following options were used for the PGI compiler:

- "-mp=nonuma -fast" compiler options were used for hybrid programs
- "-fast" was used for pure MPI programs.

MPI time = 149.07				
hybrid 1	**dynamic**	**dynamic 1024**	**static**	**guided**
1 MPI proc/node	394.35	278.12	222.65	253.11
2 MPI proc/node	220.01	148.38	148.57	148.13
4 MPI proc/node	221.30	147.55	149.08	148.22
6 MPI proc/node	185.78	147.24	148.34	148.01
hybrid 2				
1 MPI proc/node	390.74	252.37	221.21	257.42
2 MPI proc/node	219.04	148.37	147.70	147.99
4 MPI proc/node	220.21	148.05	147.45	148.54
6 MPI proc/node	182.98	148.25	147.82	148.34
hybrid 3				
1 MPI proc/node	394.33	235.91	220.10	250.74
2 MPI proc/node	219.93	148.61	148.31	148.54
4 MPI proc/node	229.26	147.73	148.24	149.63
6 MPI proc/node	186.20	149.75	148.06	150.34

Table 1. Jacobi iteration timings in seconds for 2 nodes.

Table 1 shows timings in seconds using two nodes for the pure MPI, hybrid 1, hybrid 2 and hybrid 3 programs using various numbers of MPI processes per node and various scheduling options. Notice that 1 MPI process per node gave poor performance results. This was expected because of the ccNUMA architecture of each node. Notice that dynamic scheduling gave poor performance. This is likely due to the fact that the default chunksize is 1. Chunksize=1024 gave good performance; however, the problem is that the optimum chunksize will vary depending on the problem size and the number of nodes used. For this reason, the scalability comparisons in table 2 only use the static and guided scheduling options. Notice that the pure MPI and all hybrid methods with at least 1 MPI process/socket and using static and guided scheduling performed about the same. Because of this the scalability comparisons were done only using 1 MPI process/socket.

The time spent in the MPI communication portion of the pure MPI program was measured by inserting a call to mpi_barrier before the two calls to mpi_sendrecv's, and then timing only these MPI routines. These timings showed that only about 0.2% of the total time was spent in MPI communication. Since the communication time is independent (ignoring switch contention) of the number of nodes used, it does not scale at all. However, the communication time is so small that its lack of scalability should have little effect for the Jacobi iteration up to 64 nodes (64*12=768 cores). Looking at the MPI program, one would expect the computational time to drop by a factor of two each time the number of cores is doubled. Table 2 confirms the near perfect scalability for the pure MPI program and for hybrid programs using static and guided scheduling for this problem size up to 64 nodes. Table 2 also shows that the hybrid programs perform about the same as the pure MPI program from 2 nodes up to 64 nodes when using 2 (and 4) MPI processes per node.

Nodes	**MPI**	**static**		
		hybrid 1	**hybrid 2**	**hybrid 3**
2	149.07	148.57 (149.08)	147.70 (147.45)	148.31 (148.31)
4	74.96	75.07 (74.22)	74.25 (73.93)	75.18 (75.21)
8	37.65	37.42 (37.41)	37.01 (37.14)	37.34 (37.66)
16	18.94	18.88 (18.74)	18.68 (18.92)	18.76 (19.53)
32	9.37	9.58 (9.67)	9.65 (9.48)	9.51 (9.66)
64	4.91	4.98 (4.91)	4.95 (4.97)	4.91 (5.09)
		guided		
		hybrid 1	**hybrid 2**	**hybrid 3**
2	149.07	148.13 (148.22)	147.99 (148.54)	148.54 (149.63)
4	74.96	74.65 (74.51)	74.23 (74.06)	75.22 (75.59)
8	37.65	37.61 (37.52)	37.61 (37.37)	37.76 (38.20)
16	18.94	19.16 (18.98)	19.09 (18.91)	19.18 (19.40)
32	9.57	9.85 (9.65)	9.87 (9.71)	9.73 (9.96)
64	4.91	5.20 (5.11)	5.15 (5.11)	5.05 (5.29)

Table 2. Jacobi iteration timings in seconds using 2 (and 4) MPI process/node for hybrid programs.

We experimented with using less than six cores on the sockets and found that with a fixed number of nodes and for a fixed problem size, it is most efficient to utilize all six cores for each socket.

Notice that when using the hybrid Jacobi with 2 MPI processes per node instead of 12 in the pure MPI Jacobi, MPI communication decreases by a factor of 6. To take advantage of this fact we explored a rectangular Jacobi iteration where the MPI communication will be about 25% of the total time. Instead of the 2-dimensional arrays A and B being square, we made them rectangular of sizes n1 by n2 where n1*n2 = n*n so that the number of operations performed would be the same as in the above example with n = 24*1024. Table 3 presents the performance data when n1 = 24*1024*256 and n2 = 24*4 using 2 nodes. Notice that for this rectangular Jacobi iteration, all hybrid versions outperform the pure MPI with hybrid 3 with static scheduling giving a speedup of 1.17 over the pure MPI version.

MPI = 279.92	time in seconds	Speedup = MPI/hybrid
hybrid 1 (static)	258.12	1.08
hybrid 1 (guided)	259.08	1.08
hybrid 2 (static)	253.80	1.10
hybrid 2 (guided)	252.16	1.11
hybrid 3 (static)	239.79	1.17
hybrid 3 (guided)	241.21	1.16

Table 3. Rectangular Jacobi iteration timings with 2 nodes using 2 MPI process/node for hybrid.

4. The 3D FFT

Fast Fourier Transform [9] is widely used in many engineering and scientific applications. The 3D FFT presented in the paper is a straightforward parallel implementation of the 3D FFT formula (1) for a problem size n*n*n that calls multiple 1D FFTs.

$$y(k_1,k_2,k_3)=\sum_{j_3=0}^{n_3-1}\sum_{j_2=0}^{n_2-1}\sum_{j_1=0}^{n_1-1}x(j_1,j_2,j_3)\omega_1^{j_1k_1}\omega_2^{j_2k_2}\omega_3^{j_3k_3} \quad (1)$$

where $\omega_r = e^{-2\pi i/n_r}, r = 1,2,3, i = \sqrt{-1}.$

Let p be the number of MPI processes. Let m = n/p and assume that p divides n evenly so that we can use mpi_alltoall instead of mpi_alltoallv for the global transpose portion of the FFT. Let x and y be double complex, one-dimensional arrays of size n*n*m where x is the input data and the output data is stored in y in normal form.

One can perform multiple 1D FFTs by calling ZFFT1D from AMD's Core Math Library within a loop or one can make a single call to ZFFT1M(). We tried both methods and found that the timings were the same within the accuracy of our timings, so we used ZFFT1M to simplify the program.

The following describes the hybrid program used for this study. The pure MPI program was obtained by compiling the hybrid program without OpenMP support. Notice that there are 8 loops. Loops 1, 3 and 6 each perform m*n 1D FFT's and loops 2, 4, 5, 7 and 8 all rearrange data in x and y. We selected the order of the loops based on stride with the smallest stride for the inner most loop and the largest stride for the outer most loop. However, it was not obvious how to arrange the loops for loop 4, so we experimented and measured times. Parallelizing loop 4 with OpenMP using dynamic scheduling on the j loop instead of the k loop dropped the time from 9.4 seconds to 4.7 seconds when using 4 nodes, so we took the outer loop to be the j loop. Experiments for loops 2, 4, 5, 7 and 8 were also performed to determine if loop collapsing and specified scheduling would help performance. The cases where this helped performance are indicated in the pseudo-code below. We tried fusing loops 1, 2 and 3 in the MPI program and found there was no increase in performance. We also tried to do loop collapsing for the 1D FFT loops and found there was no performance improvement. Based on our experiments, the version below performs the best.

```
! hybrid 3D FFT
. . .
integer,parameter :: n = 12*128
double complex,allocatable :: x(:), y(:)
call mpi_init_thread(mpi_thread_multiple, provided, ierror)
call mpi_comm_size(mpi_comm_world, p, ierror)
m = n/p  !  we assume p divides n evenly
allocate(x(0:n*n*m-1), y(0:n*n*m-1)
        . . .
call mpi_barrier(comm, ierror)
t = mpi_wtime()
!$omp  parallel shared(x, y, m) private(i, j, k, comm)
!$omp  do schedule(runtime) ! LOOP 1:  m*n 1D FFTs
       do k = 0, m-1
          call ZFFT1M(-1, n, n, x(k*n*n), comm, info)
       enddo
!$omp  do schedule(guided) collapse(2) ! LOOP 2:  transpose
       do k = 0, m-1
          do j = 1, n-1
             do i = 1, n-1
                y(i + j*n + k*n*n)  =  x(j + i*n + k*n*n)
             enddo
          enddo
       enddo
!$omp  do schedule(runtime) !  LOOP 3:  m*n 1D FFTs
       do k = 0, m-1
          call ZFFT1M(-1, n, n, y(k*n*n), comm, info)
       enddo
!$omp  do schedule(dynamic) LOOP 4:  transpose
       do j = 0, m-1
          do k = 0, n-1
             do i = 0, n-1
```

```
                x(i+j*m+k*n*m) = y(k+j*n+i*n*n)
              enddo
            enddo
          enddo
!$omp  single
       call mpi_alltoall(x, n*m*m, mpi_double_complex, y, &
       n*m*m, mpi_double_complex, mpi_comm_world, ierror)
!$omp  end single ! implicit barrier
!$omp  do schedule(runtime) !  LOOP 5:  rearrange data
          do k = 0, m-1
             do j = 0, n-1
                do ip = 0, p-1
                   do i = 0, m-1
                      x(i + ip*m + j*n + k*n*n) = &
                                   y(i + j*m + (ip*m+k)*m*n)
                    enddo
                  enddo
               enddo
          enddo
!$omp  do schedule(runtime) ! LOOP 6:  m*n 1D FFTS
          do k = 0, m-1
             call ZFFT1M(-1, n, n, x(k*n*n), comm, info)
          enddo
!$omp  do schedule(guided) collapse(2) ! LOOP 7: transpose
          do j = 0, m-1
             do k = 0, n-1
                do i = 0, n-1
                   y(i + j*n + k*m*n) = x(k + i*n + j*n*n)
                enddo
             enddo
          enddo
!$omp  single
       call mpi_alltoall(y, n*m*m, mpi_double_complex, x, &
       n*m*m, mpi_double_complex, mpi_comm_world, ierror)
!$omp  end single ! implicit barrier
!$omp  do schedule(dynamic) collapse(3) ! LOOP 8:
                                          !  rearrange data
          do k = 0, m-1
             do ip = 0, p-1
                do j = 0, m-1
                   do i = 0, n-1
                      y(i + j*n + ip*n*m + k*n*n) = &
                                   x(i + j*n + (ip*m+k)*m*n)
                    enddo
                enddo
             enddo
          enddo
!$omp   end parallel
 time = mpi_wtime() – t
 call mpi_barrier(comm, ierror)
 call mpi_reduce(time, max_time, 1, dp, mpi_max, 0, comm,
ierror)
```

5. Performance Results for the 3D FFT

All runs and timings were made as was done for the Jacobi iteration, i.e. the minimum of 10 trials is reported. As was done for the Jacobi iteration, the pure MPI program was obtained by compiling the hybrid program without OpenMP support. Since each node has 12 cores, the problem size, n, should be a multiple of 12 so that all cores will have the same amount of computation. So we could compare hybrid and pure MPI performance using 4, 8, 16, 32 and 64 nodes, we chose the largest problem that could be run on 4 nodes of the Cray XT5 we used, and also could be run on 64 nodes. n = 12*128 satisfied these conditions.

Notice that when using the hybrid FFT with 2 MPI processes per node instead of 12 in the pure MPI FFT, there are 6 times fewer processes calling mpi_alltoall, but the message size is 36 times greater. From this it follows that the total amount of data sent is the same. The effects of this are presented in tables 4, 5 and 6.

Table 4 gives performance comparisons in seconds for 4 nodes using 2, 4 and 6 MPI processes per node when using dynamic, static or guided scheduling for those loops where the scheduling is not specified. Notice that using 4 MPI processes/node (2 per socket) gives better performance than 2 MPI processes/node (1 per socket). Also notice that the data for 1 MPI process/node is missing. This is because the message size in this case is over 2 GB and the MPI on the XT5 currently has a limit of 2 GB for the message size. We were able to run with 1 MPI process/node when using 8 or more nodes. Times were much worse when compared with 2 or 4 MPI processes/node.

MPI time = 65.61			
hybrid	**dynamic**	**static**	**guided**
2 MPI proc/node	58.98	58.19	58.18
4 MPI proc/node	57.85	56.94	57.71
6 MPI proc/node	69.46	69.61	69.08

Table 4. 3D FFT timings in seconds for 4 nodes.

Table 5 presents times with 1 and 2 MPI processes/socket (2 and 4 MPI processes/node) using 4 nodes to 64 nodes so scalability can be evaluated for both the pure MPI and hybrid 3D FFTs. Notice the hybrid performs faster than pure MPI for 4 (13% faster), 32 (19% faster) and 64 (25% faster) nodes and the performance is about the same for 16 nodes. The performance using 2 MPI processes/socket is faster than 1 MPI process/socket for 4, 8 and 16 nodes and slower for 32 and 64 nodes. Both the pure MPI and hybrid programs scaled well. The time spent in the two calls to

mpi_alltoall for 4, 8, 16, 32 and 64 nodes ranged from about 60% to 70% of the total time.

Nodes	MPI	hybrid with 2 MPI proc/node (4 MPI proc/node)		
		dynamic	static	guided
4	65.61	58.98 (57.85)	58.19 (56.94)	58.18 (57.71)
8	31.95	35.14 (34.06)	35.09 (33.94)	34.63 (34.14)
16	16.93	16.94 (15.42)	16.73 (15.29)	16.78 (15.44)
32	8.99	7.28 (8.07)	7.26 (7.96)	7.28 (8.06)
64	5.28	3.95 (4.53)	3.92 (5.51)	3.92 (4.51)

Table 5. 3D FFT with 2 (and 4) MPI process/node for hybrid.

6. Comparison with the HPC Challenge 3D FFT Benchmark

In this section we compare the performance of the 3D FFT presented in section 4 with the 3D FFT from the HPC Challenge (HPCC) Benchmarks (pzfft3d.f) written by Daisuke Takahashi at the University of Tsukuba in Japan [10, 11]. This program includes blocking for cache to increase performance. The experiments reported in [10] show that the blocking for cache nearly doubled the performance compared with the performance of a corresponding FFTW routine on a cluster in Japan. Reported performance results in [10] comparing the hybrid and pure MPI programs showed that sometimes the hybrid outperformed the pure MPI program and sometimes the pure MPI program outperformed the hybrid program. Timings were performed as was done throughout this paper, i.e. reported times are the minimum of 10 trials.

Several changes to the HPCC FFT program were made so it could run for n = 12*128. Since this program blocks for cache, we set the cache size to 512 KB, the size of the L2 cache on the Cray XT5 (there was no improvement in performance when the cache size was set to 256 KB). The HPCC hybrid FFT does not specify any scheduling so default scheduling is done (likely static).

Table 6 shows timing comparisons between the FFT presented in this paper and the HPCC FFT using 1 MPI process/socket and n = 12*128. For both FFT's, the pure MPI program was obtained from the hybrid program by compiling without OpenMP support. This guarantees that the MPI program uses the same algorithm as the hybrid program. The pure MPI HPCC FFT performs from 5% to 12% faster than the MPI FFT presented in this paper and both the hybrid FFT's perform nearly the same. The hybrid FFT's performed better than the pure MPI FFT's for 32 and 64 nodes and performed worse for 4, 8, and 16 nodes. Notice that the hybrid and pure MPI FFT's presented in the paper both performed well compared with the hybrid and pure HPCC FFT's even though there was no blocking for cache. The good performance of the hybrid 3D FFT presented in this paper is likely due to the careful selection of loop ordering, thread scheduling and loop collapsing that was not done for the HPCC FFT.

Nodes	HPCC MPI	FFT MPI (speedup)	HPCC hybrid	FFT hybrid (speedup)
4	57.57	65.61 (0.88)	61.87	58.18 (1.06)
8	29.65	31.95 (0.93)	35.86	34.63 (1.04)
16	15.47	16.93 (0.92)	16.87	16.73 (1.01)
32	8.43	8.99 (0.94)	7.31	7.26 (1.01)
64	4.71	5.28 (0.89)	3.96	3.92 (1.01)

Table 6. Comparing the performance of the HPCC FFTs with the FFT presented in section 4 using 2 MPI process/node

7. Conclusions

The purpose of this study was to compare the performance of a pure MPI program with the performance of a hybrid program for the Jacobi iteration and for a 3D FFT on the Cray XT5. We started with an efficient MPI parallelization of the Jacobi iteration and a 3D FFT and then developed an efficient OpenMP version of the MPI program. Compiling the hybrid program without OpenMP support produced the original MPI program. For the Jacobi iteration, three hybrid versions are presented and their performance compared with the pure MPI program. When using the square Jacobi, all three of the hybrid programs perform nearly the same as the pure MPI for 2, 4, 8, 16, 32 and 64 nodes using 1 and 2 MPI process per socket. When using a rectangular region for 2 nodes, hybrid version 3 out-performed the pure MPI program by about 17%. The pure MPI and all of the hybrid programs scaled well for the square Jacobi iteration.

For the 3D FFT, we used a standard algorithm for the MPI program and then developed an efficient OpenMP parallelization by carefully selecting the ordering of loops, loop scheduling and loop collapsing. The hybrid program gave the best performance using 4, 32 and 64

nodes but did not perform as well as the pure MPI program for 8 and 16 nodes. Sometimes using 2 MPI process/socket gave a little better performance than using 1. The performance of our FFT was then compared with the performance of the cache blocking, 3D FFT from the HPC Challenge Benchmarks. The FFT presented here did not block for cache yet achieved nearly the same performance as the HPC Challenge FFT.

Experiments showed that for both Jacobi iteration and for the 3D FFT, it was most efficient to use all 12 cores on the compute nodes for the pure MPI program. Our experiments also showed that using 1 MPI process per node on a CC-NUMA compute node does not give good performance. Best performance for the hybrid programs considered in this paper were achieved using either 1 or 2 MPI processes per socket.

Acknowledgments

We would like to thank Cray Inc. for providing time on their "koi" XT5 for this study.

About the Authors

All authors are members of Iowa State University's (ISU's) High Performance Computing (HPC) Group that has had contracts continuously with Cray since 1992. Glenn Luecke is Director of ISU's HPC Group and Professor of Mathematics. He can be reached at Durham Center, Iowa State University, Ames, Iowa 50011, USA, phone: 515-294-6659, E-Mail: grl@iastate.edu. Olga Weiss is a student member of ISU's HPC Group. She can be reached at 213 Atanasoff, Iowa State University, Ames, Iowa 50011, USA, phone: 515-509-3368, E-Mail: olga@iastate.edu. Marina Kraeva (kraeva@iastate.edu), James Coyle (jjc@iastate.edu) and Jim Hoekstra (hoekstra@iastate.edu) are PhD and senior members of ISU's HPC Group. They can be reached at Durham Center, Iowa State University, Ames, Iowa 50011, USA.

References

1. J. Dongarra, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg and M. Snir, *MPI – The Complete Reference Volumes 1 and 2,* The MIT Press, 1998.
2. B. Chapman, G. Jost and R. van der Pas, *Using OpenMP,* The MIT Press, 2007.
3. R. Rabenseifner, G. Hager, G. Jost, R. Keller, *Hybrid MPI and OpenMP parallel programming*, Proceedings of PVM/MPI 2006, Lecture Notes in Computer Science, Vol. 4192, 2006.
4. G. Hager, G. Jost and R. Rabenseifner, *Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes,* Cray User's Group 2009.
5. M. Stuermer, G. Wellein, G. Hager, H. Koestler, U. Ruede, *Challenges and potentials of emerging multicore architectures*, High Performance Computing in Science and Engineering, Garching/Munich 2007, 551–566, Springer, 2009.
6. R. Rabenseifner, G. Hager, G. Jost, *Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes,* Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009), Weimar, Germany, Feb. 16–18, 2009, pp. 427–236, Computer Society Press 2009.
7. D. Mallon, G. Taboada, T. Carlos, J Tourino, B. Fraguela, A. Gomez, R. Doallo and J. Mourino, *Performance evaluation of MPI, UPC and OpenMP on multicore architectures*, Proceedings of EuroPVM/MPI 2009, Lecture Notes in Computer Science, Vol. 5759, 2009.
8. Cray XT5 System Highlights, http://www.cray.com/Assets/PDF/products/xt/CrayXT5Brochure.pdf
9. J.W. Cooley, J.W.Tukey, *An algorithm for the machine calculation of complex Fourier series*, Mathematics of Computation, Vol.19, No. 90, pp. 297-301, 1965.
10. D. Takahashi, *A Hybrid MPI/OpenMP Implementation of a Parallel 3-D FFT on SMP Clusters*, Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics (PPAM 2005), Lecture Notes in Computer Science, No. 3911, Springer Verlag 2006.
11. HPC Challenge Benchmarks: http://icl.cs.utk.edu/hpcc/