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

Glenn Luecke, Olga Weiss, Marina Kraeva, James Coyle, James Hoekstra
High Performance Computing Group
Iowa State University
Ames, Iowa, USA
May 2010

Acknowledgements

Cray for use of their XT5 and their support of this project.

Outline

- Hybrid programming
- MPI Jacobi iteration
- Three hybrid versions of Jacobi iteration
- Jacobi performance results
- MPI and hybrid 3D FFT
- FFT performance results
- Comparison with HPC Challenge 3D FFT
- Conclusions

Hybrid Programming

- MPI designed for distributed memory parallelization
- OpenMP designed for shared memory parallelization
- Current HPC machines are interconnected SMP nodes
- Hope is to use MPI between nodes and OpenMP within nodes and achieve better performance than using MPI among nodes and all cores.
- MPI program obtained by compiling hybrid program without OpenMP support.

Jacobi Iteration

- Used to solve Laplace's equation in a rectangle.
- We use the column blocked distribution described in figure 2.4 of "MPI – The Complete Reference".
- The mpi_sendrecv version is used.
- p = number of MPI processes
- n = 24*1024 = the problem size
- m = n/p
- A(0:n+1,0:m+1), B(1:n,1:m)
- The Jacobi iteration is performed 300 times

MPI Jacobi Iteration

```
call mpi_barrier(comm, ierror)
t = mpi_wtime()
do iter = 1, 300
    do j = 1, m
        do i = 1, n
            B(i,j) = 0.25*(A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1))
        enddo
    enddo
    do j = 1, m
        A(1:n,j) = B(1:n,j)
    enddo
    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
time = mpi_wtime() - t
```

Version 1: hybrid Jacobi Iteration

```
!$omp  parallel shared(A,B,m) private(i,j)
!$omp  do schedule(runtime)
          do j = 1, m
             do i = 1, n
                B(i,j) = 0.25*(A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1))
             enddo
          enddo ! implicit barrier
!$omp  do schedule(runtime)
          do j = 1, m
             A(1:n,j) = B(1:n,j)
          enddo ! implicit barrier
!$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,...)
```

Version 1: hybrid Jacobi Iteration

- mpi calls are outside the parallel region
- a new parallel region is created for each iteration

Version 2: hybrid Jacobi Iteration

```
!$omp  parallel shared(A,B,m) private(i,j)
!$omp  do schedule(runtime)
          . . .
!$omp     do schedule(runtime)
          do j = 1, m
             A(1:n,j) = B(1:n,j)
          enddo
!$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
        enddo
!$omp end parallel
```

Version 2: hybrid Jacobi Iteration

- there is a nowait on the A = B loop
- MPI communication is within a single region
- one parallel region is created for all iterations
- setenv MPICH_MAX_THREAD_SAFETY serialized
- call mpi_init_thread(mpi_thread_serialized, provided, ierror)

Version 3: hybrid Jacobi Iteration

```
!$omp  parallel shared(A,B,m) private(i,j)
!$omp  do schedule(runtime)
          . . .
!$omp     do schedule(runtime)
          do j = 1, m
             A(1:n,j) = B(1:n,j)
          enddo
!$omp     end do nowait
!$omp    sections
          call mpi_sendrecv(B(1,m),n,dp,...,A(1,0), n,dp,...)
!$omp     section
          call mpi_sendrecv(B(1,1), n,dp,...,A(1,m+1),n,dp,...)
!$omp    end sections
        enddo
!$omp end parallel
```

Version 3: hybrid Jacobi Iteration

- Each mpi_sendrecv is executed by a different thread allowing the communication to execute concurrently.
- The following option to link the libmpich_threadm library is needed: -l mpich_threadm.
- call mpi_init_thread(mpi_thread_multiple, provided, ierror)
- setenv MPICH_MAX_THREAD_SAFETY multiple.

ccNUMA nodes on the XT5

- 2 AMD Istanbul 6 core Opteron processors, 6 MB shared L-3 cache, 32 GB of memory/node.
- Hybrid performance problem using hybrid with 1 MPI process per node.
- Hybrid with 1 or more MPI processes per socket will eliminate this problem assuming memory and CPU affinity to cores
- ccNUMA is not a performance problem for pure MPI
- OpenMP allows threads to be scheduled using: static, dynamic and guided with each option having a “chunksize” option.

Compiler and aprun options

- To bind processing elements to CPUs, the “-cc cpu” option was used on the “aprun” command. (CPU affinity)
- To ensure processing elements will allocate only the memory local to its assigned NUMA node, the “-ss” option was also used on the “aprun” command. (memory affinity)
- The PGI “-mp=numa –fast” compiler options were used for hybrid programs and “-fast” was used for pure MPI programs.
- To be able to run the hybrid programs, it was necessary to setenv OMP_STACKSIZE 1G

Table 1: Jacobi timings in seconds with 2 nodes

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: observations

- Hybrid with 1 MPI proc/node: much slower
- Dynamic scheduling: poor performance
- Dynamic with chunksize=1024: good performance
- Static and guided with 2, 4 and 6 MPI proc/node: about the same performance as pure MPI
- Communication only 0.2% of total time
- Note from the MPI algorithm that the communication time is independent of number of nodes, i.e. it doesn't scale.

Table 2: 1 MPI proc/socket
(results nearly identical for 2 MPI proc/socket)

Nodes	MPI	static			guided		
		Hybrid 1	Hybrid 2	Hybrid 3	Hybrid 1	Hybrid 2	Hybrid 3
2 nodes	149.07	148.57	147.70	148.31	148.13	147.99	148.54
4 nodes	74.96	75.07	74.25	75.18	74.65	74.23	75.22
8 nodes	37.65	37.42	37.01	37.34	37.61	37.61	37.76
16 nodes	18.94	18.88	18.68	18.76	19.16	19.09	19.18
32 nodes	9.37	9.58	9.65	9.51	9.85	9.87	9.73
64 nodes	4.91	4.98	4.95	4.91	5.20	5.15	5.05

Table 2: Jacobi with 2 - 64 nodes

- Hybrid and pure MPI performed about the same
- Good scaling
- Notice that hybrid with 2 MPI proc/node has 6 times less communication = > rectangular Jacobi = > table 3
- Table 3 shows hybrid gives about 16% improvement in time when about 25% of time spent in communication.

Table 3: rectangular Jacobi, 2 nodes, 1 MPI proc/socket

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

3D FFT

- Developed from Cooley-Tukey algorithm
- Distributed FFT for problem size of n*n*n with calls to 1D FFT from AMD's Core Math Library double complex routine.
- There are 8 loops: loops 1,3 and 6 perform n*m 1D FFT's
- Loops 2, 4, 5, 7 and 8 all transpose/rearrange data in x and y
- There are two calls to mpi_alltoall (when p divides n evenly)
- Each loop was timed and independently optimized via loop reordering, OpenMP scheduling and loop collapsing
- Recall m = n/p where p = number of MPI processes
- n = 12*128, largest problem for 4 nodes scaling to 64 nodes

Loop 1: m*n 1D FFT's

```
!$omp  parallel shared(x, y, m) private(i, j, k, comm)
!$omp  do schedule(runtime)
          do k = 0, m-1
               call zfft1m(-1, n, n, x(k*n*n), comm, info)
          enddo
```

- Loop collapsing did not help performance

loop 2: transposing data

```
!$omp  do schedule(guided) collapse(2)
          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
```

loop 4: transposing data

```
!$omp  do schedule(dynamic)
       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
```

MPI_ALLTOALL

```
!$omp  single
       call mpi_alltoall(x, n*m*m, mpi_double_complex, y, &
          n*m*m, mpi_double_complex, mpi_comm_world, ierror)
!$omp  end single ! implicit barrier
```

- Note: message length = n*m*m = (n*n*n)/(p*p)
- Hybrid with 2 MPI processes/node has 6 times fewer calls to mpi_alltoall with messages that are 36 times larger.

Loop 8: rearranging data

```
!$omp  do schedule(dynamic) collapse(3)
          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
```

Table 4: FFT timings with 4 nodes

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: observations

- 1 MPI proc/node would not run (2 GB message limit)
- 2 and 4 MPI proc/node performed about 13% better than MPI
- 6 MPI proc/node did not perform well
- Dynamic, static and guided scheduling performed nearly the same
- 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. However, the total amount of data sent is the same.

Table 5: FFT with 1 (2) MPI processes/socket

Nodes	MPI	Hybrid with 1 MPI proc/socket (2 MPI proc/socket)		
		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: observations

- Hybrid is faster than pure MPI for 4 (13% faster), 32 (19% faster) and 64 (25% faster) nodes
- 2 MPI proc/socket is faster than 1 MPI proc/socket for 4, 8 and 16 nodes and slower than 4 MPI proc/socket for 32 and 64 nodes
- All scaled scaled well.
- Time spent in mpi_alltoall ranged from about 60% to 70% of the total time.

Table 6: HPCC 3D FFT comparison

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)

Comparison with HPC Challenge 3D FFT

- NPCC 3D FFT blocks for cache (D. Takashi, pzfft3d.f)
- Our FFT does not block for cache
- Uses default thread scheduling (likely, static)
- No loop collapsing (OpenMP 3.0 is new)
- Pure MPI program was obtained by compiling hybrid program without OpenMP support
- Our hybrid performed nearly the same even though our FFT did not block for cache
- Pure MPI HPCC FFT performed 8-12% faster

Conclusions

- Use 1 or 2 MPI processes per socket
- Square Jacobi – hybrid and pure performed nearly the same
- Rectangular Jacobi – hybrid faster
- FFT: hybrid is faster than pure for 4 (13% faster), 32 (19% faster) and 64 (25% faster) nodes
- Hybrid FFT performed as well as the HPC Challenge FFT even though there is no blocking for cache
- Pure MPI HPCC FFT also blocked for cache and performed 8-12% faster