Scientific Programming 18 (2010) 127–138
DOI 10.3233/SPR-2010-0308
IOS Press

Experiences using hybrid MPI/OpenMP in the real world: Parallelization of a 3D CFD solver for multi-core node clusters

Gabriele Jost [a,*] and Bob Robins [b]

[a] *Texas Advanced Computing Center, The University of Texas, Austin, TX, USA*
[b] *NorthWest Research Associates, Inc., Bellevue, WA, USA*

Abstract. Today most systems in high-performance computing (HPC) feature a hierarchical hardware design: shared-memory nodes with several multi-core CPUs are connected via a network infrastructure. When parallelizing an application for these architectures it seems natural to employ a hierarchical programming model such as combining MPI and OpenMP. Nevertheless, there is the general lore that pure MPI outperforms the hybrid MPI/OpenMP approach. In this paper, we describe the hybrid MPI/OpenMP parallelization of IR3D (Incompressible Realistic 3-D) code, a full-scale real-world application, which simulates the environmental effects on the evolution of vortices trailing behind control surfaces of underwater vehicles. We discuss performance, scalability and limitations of the pure MPI version of the code on a variety of hardware platforms and show how the hybrid approach can help to overcome certain limitations.

Keywords: Hybrid MPI/OpenMP, CFD solver, scalability, performance

1. Introduction

Most of today's HPC (High Performance Computing) platforms are highly hierarchical and asymmetric. Multiple sockets of multi-core shared-memory compute nodes are coupled via high-speed interconnects. For the scientific software developer, who aims to achieve high scalability, it seems natural to employ a programming model which matches the hierarchy of the underlying hardware platform: shared-memory programming within one node and a message passing-based approach for parallelization across the nodes. The de-facto standard for message passing is MPI (see [11]). The interface is standardized and implementations are available on virtually every distributed memory system. OpenMP (see [13] and [2]) is the currently most common shared-memory programming model. Combining MPI and OpenMP is therefore the path most often taken by the scientific programmer when trying to exploit shared-memory parallelism within one node on multi-core node clusters. While the idea seems good, what happens in practice is that very often a pure MPI implementation will outperform the hybrid MPI/OpenMP code. This situation has led to the general opinion that pure MPI is just more efficient. The purpose of this paper is to describe how we improved the performance of a full-scale CFD application by combining MPI and OpenMP. The rest of the paper is structured as follows. In Section 2 we give a brief description of the PIR3D application, which is the basis of our case study. We describe the physical problem solved and review the MPI-based parallel implementation of the code. In Section 3 we discuss the performance of the pure MPI code, describe our approach to adding OpenMP directives and we compare the performance of the pure MPI and MPI/OpenMPI versions. In Section 4 we discuss related work. In Section 5 we draw our conclusions and provide a set of best practices for successful hybrid parallelization based on our experience.

2. The 3D CFD solver PIR3D

The MPI-based parallel code PIR3D was evolved from the existing sequential code, IR3D, which had been optimized for vector processing. Details about the development of the MPI-based parallel version,

*Corresponding author: Gabriele Jost, Texas Advanced Computing Center, The University of Texas, Austin, TX, USA. Tel.: +1 831 656 3321; E-mail: gjost@tacc.utexas.edu.

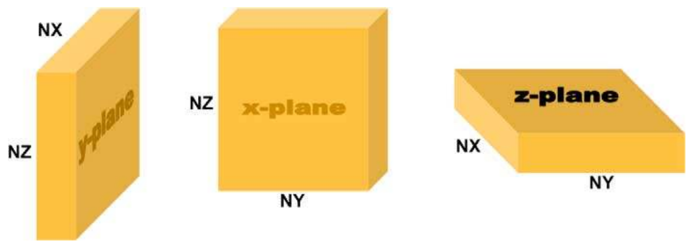

Fig. 1. Data structures used in PIR3D. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0308.)

PIR3D, are described in [15]. In this section we review the material.

2.1. The physical problem and numerical methodology

Control surfaces of underwater vehicles generate trailing vortices and it is often important to understand how the evolution of these vortices is affected by ambient turbulence, density stratification and underwater currents. To study this problem, NorthWest Research Associates, Inc. developed an application that simulates environmental effects on trailing vortices. This code (called IR3D – Incompressible Realistic 3D) solves the Boussinesq flow equations (see [3]) for an incompressible fluid. One or more pairs of trailing vortices are specified as initial conditions for the flow equations and the computed solution of the flow equations provides a simulation of the vortices' evolution. The numerical solution to the flow equations is obtained using an explicit second-order Adams–Bashforth time stepping scheme (see [3]). The horizontal derivatives on the right-hand side of the equations are computed using fast Fourier transforms and vertical derivatives are computed using higher-order compact methods. The incompressibility of the flow is maintained by using a projection method, which requires solution of a Poisson's equation at every time step. We use a Poisson solver specially developed to solve the specific Poisson's equations that arise as the solution proceeds. The solver works by computing 2D FFTs in horizontal-planes, numerically solving the resulting ODEs for Fourier coefficients and then doing Fourier inversion. In order to control the build-up of small scale numerical noise, we do periodic smoothing of the evolving flow. Horizontal smoothing is done using FFT's and vertical smoothing is done with higher-order compact methods. Sub-grid scale viscous dissipation is represented by a model developed by Holzaepfel (see [6]). In addition, IR3D provides diagnostics such as timing information and normalized root-mean-square divergence to check the incompressibility of the flow.

2.2. Code implementation and parallelization approach

The data structures used for the calculations are embedded in 3D arrays, each holding $NZ \times NY \times NX$ floating point numbers. The algorithm computes multiple z- and y-derivatives in x-planes, and multiple x-derivatives in y-planes. The Poisson solver utilizes z-planes. The first step in the parallelization process is to decide how to distribute the data among multiple processes. By adopting a one-dimensional domain decomposition that uses slabs to replace the planes in the vector version of the code, we were able to retain most of the original code structure. Thus, every three-dimensional array is distributed so that each MPI process now owns three slabs of the computational domain. The size of these slabs is given by $NZ \times locny \times NX$, $NZ \times NY \times locnx$ and $locnz \times NY \times NX$, where $locn[yxz] = N[YXZ]/nprocs$, $nprocs$ being the number of MPI processes. In Fig. 1 we symbolically represent the data structures used in our domain decomposition. As a default we assume a distribution in the y-direction (using $NZ \times locny \times NX$ slabs) and each MPI process updates one y-slab. This is the configuration in which the initial conditions are defined. Since some routines need to perform updates in x- or z-slabs, it is necessary, within these routines, to redistribute the data among the MPI processes in such a way that data structures are being swapped from y-slabs into x- or z-slabs. This requires global communication among the processes. Listing 1 outlines the structure of the source that examplifies the interaction of computation and computation when data swapping is required.

3. Combining MPI and OpenMP in PIR3D

Initial tests of our MPI-based PIR3D were run on various architectures, each of them a cluster of multi-core nodes. In this section we present timings obtained on a Cray XT5, SGI Altix and a Sun Constellation

```
!In main program:
allocate (vx_y(nz,locny,nx))       ! y-slab array
allocate (vx_x(nz,ny,locnx))       ! x-slab array
!In routine DCALC:
swap  vx_y to vx_x                 ! swap from y-slab to x-slab
swap  dvx2_y to dvx2_x             ! swap from y-slab to x-slab
update dvx2_x using vx_x           ! update x-slab array
swap dvx2_x to  dvx2_y             ! swap back to y-slab
                                   ! prior to exiting DCALC
```

Listing 1. Swapping example.

Cluster. We describe the limitations to scalability of our pure MPI based parallelization approach. Furthermore, we describe our strategy to add OpenMP directives to the MPI code and present timings comparing pure MPI vs MPI/OpenMP.

3.1. *Hardware and system software*

The Cray XT5 we used in our experiments provides 1592 compute nodes. Each node consists of two AMD Opteron Barcelona 2.3 GHz quad-core processors, connected to a 3D torus by HyperTransport links, forming an 8-way shared-memory node. There is 16 GB of shared memory per node. The nodes are connected via the Cray Seastar2 Interconnect systems. A total number of 12,736 cores is available. The MPI implementation is based on MPICH2 [1]. For compiling the code we used the Cray `ftn` compilation script. The `ftn` command invokes the PGI Fortran compiler with the Cray specific runtime environment. We used the compiler options `ftn -fastsse -tp barcelona-64 -r8`. FFTW 3.2.2.1 and Cray xt-libsci 10.4.3 were employed as numerical libraries. The SGI Altix ICE system we used contains 1920 compute nodes (15,360 compute cores). Each compute node contains two 2.8 GHz Intel Xeon 64-bit quad-core Nehalem processors and 24 GB of dedicated memory, forming an 8-way shared-memory node. The nodes are connected to each other in a HyperCube topology DDR 4X InfiniBand network. We used MPI version 1.26 from the SGI Message Passing Toolkit and the Intel compiler version 11.1 with the flags `ifort -O2 -r8` for compiling the code. We used FFTW 3.2.2 and Intel's Math Kernel Library (MKL) as numerical libraries. Results on both systems were obtained by the courtesy of the HPCMO Program, the Engineer Research and Development Center Major Shared Resource Center, Vicksburg, MS, USA. The Sun Constellation Cluster results were obtained on the Ranger system, a high-performance compute resource at the Texas Advanced Computing Center (TACC), The University of Texas at Austin. It comprises a DDR InfiniBand network which connects 3936 nodes, each with four 2.3 GHz AMD Opteron "Barcelona" quad-core chips and 32 GB of memory. This allows for 16-way shared-memory programming within each node. MVAPICH (see [12]) was used for MPI communication. For compilation we used the Intel compiler version 10.1 with the flags `ifort -r8 -O2` and the libraries FFTW 3.2.1 and GotoBlas (see [4]).

All of these systems are clusters of multi-core nodes, connected via a high-speed network. The nodes consist of two or more processors, which are also referred to as sockets. Each socket has multiple computational cores. The sockets within each node can directly access each others' memory, without going through the network, thereby forming a shared-memory system. Data located on different nodes needs to be explicitly communicated via the network. The systems expose a hierarchy of cores, sockets and nodes. They show different characteristics for intra-socket, intra-node and inter-node communication. This has an impact on MPI communication. The effect of communication hierarchies on MPI and application performance is discussed in [5,8,10]. The performance of the application will be affected by how processes are mapped onto processor cores. There are different ways to place MPI processes onto the nodes. For example, when running a code using 4 MPI Processes on Cray XT5, they could all reside within one socket, or be placed on cores on different sockets within one node, or be spread out across cores on different nodes. These possibilities are depicted in Fig. 2.

3.2. *Timings and scalability of pure MPI PIR3D*

We tested the performance of the parallel code on two sets of input data. One case is of size $NX = 256$, $NY = 512$ and $NZ = 256$ (case 1), the other is other

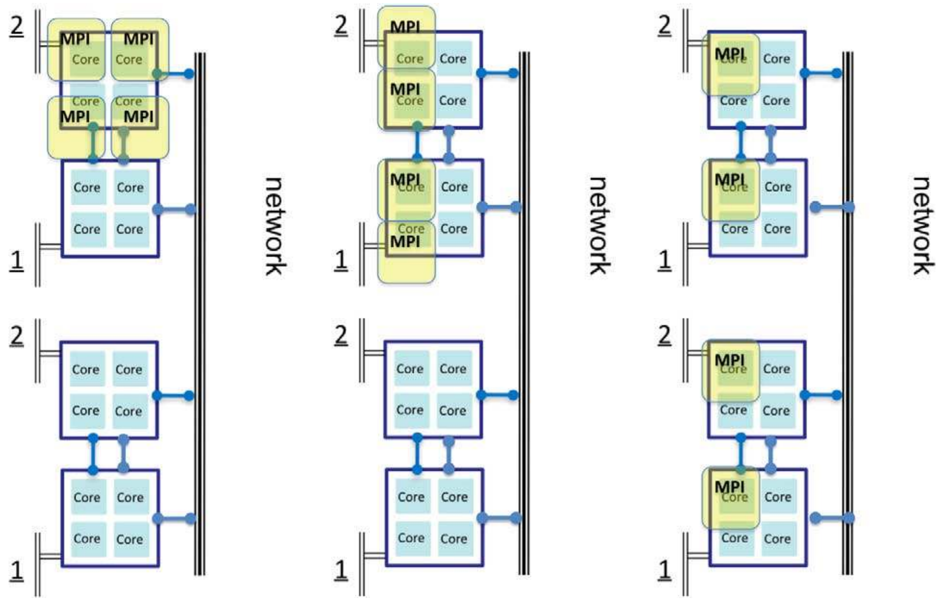

Fig. 2. Different process placement strategies. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0308.)

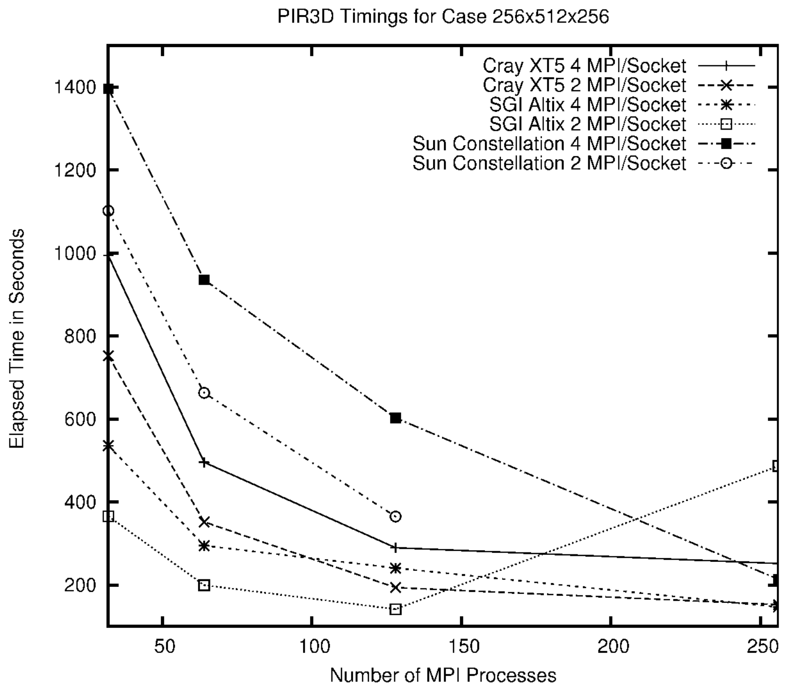

Fig. 3. Test case 1 employing different process placement strategies.

of size $NX = 1024$, $NY = 512$, $NZ = 256$ (case 2). We ran each case for 100 steps employing 2 different process placement strategies by using 4 and 2 cores per socket. The timings for 32, 64, 128 and 256 MPI processes are shown in Figs 3 and 4. Timings on all three platforms indicate a significant performance increase when using only half of the cores per socket for MPI processes. On the one hand, this offers an easy way for the user to reduce the total execution time. The flip side is, that by using only 2 cores per socket,

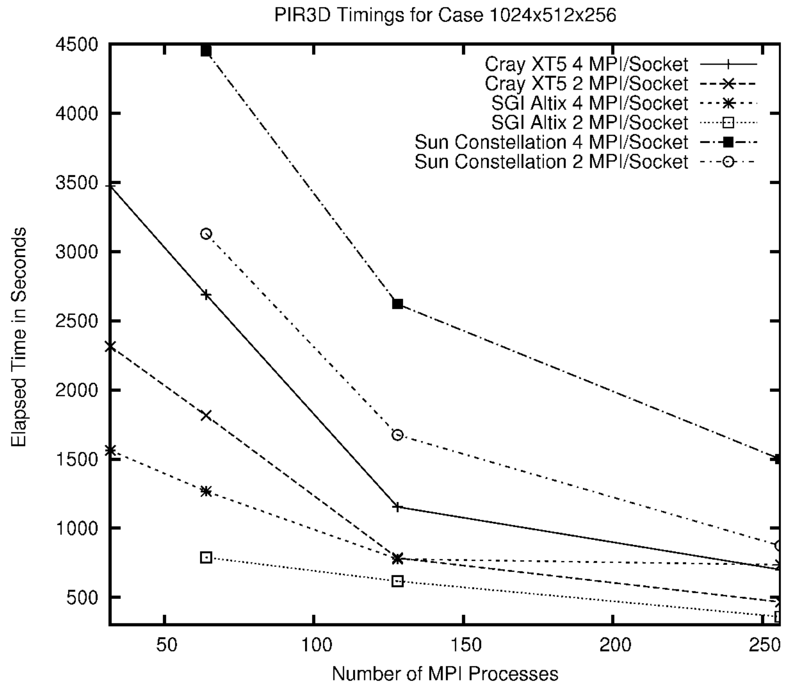

Fig. 4. Test case 2 employing different process placement strategies.

half of the computational power is left idle. Also, it may increase the actual turnaround time for the user, as the job may have to wait longer to request the large amount of resources. An application-inherent limitation to scalability of our MPI implementation is that the maximum number of MPI processes is limited to the size of the shortest of the 3 dimensions. This is due to the 1D domain decomposition and the need to swap the data in all 3 dimensions. For the input data sets under consideration this restricts the number of MPI processes to 256 in both cases.

3.3. *Adding OpenMP parallelization*

The OpenMP Application Programming Interface (API) is standardized for Fortran and C/C++. It consists of a set of compiler directives, runtime libraries, and environment variables. These facilities allow the user to direct the compiler to generate multi-threaded code for suitable sections of the code, for example parallelizable DO-loops. Threads are being forked at the beginning of parallel regions and joined at the end. The execution proceeds on a single thread outside of parallel regions.

We have noticed that we get better performance when not saturating the sockets with MPI processes. To take advantage of the idle cores within the nodes and the available shared memory we employ OpenMP shared-memory parallelization. This enables an MPI process to use multiple streams of execution during computational intensive phases. We have employed the following strategy for inserting compiler directives into the MPI code:

- Identify most time consuming routines using profiling information.
- Place OpenMP directives on parallelizable time consuming loops.
- Place directives on loops across undistributed dimensions.
- Place the directives so that no MPI calls occur within parallel regions: no thread-safety is required for the MPI library.

An example is `csfftm` which we identified as one of the time consuming routines. The source code is shown in Listing 2. The routine performs multiple one-dimensional ffts, by calls to csfft, which, in turn, calls routines from the FFTW library. Since we are now issuing calls to a runtime library routine, we do have to concern ourselves with the issue of thread-safety. A library routine is thread-safe, if it functions correctly when called simultaneously by multiples threads. The transforms are independent of each other and can be performed in parallel. To achieve this, we inserted the

```
        subroutine csfftm(isign,ny,m,rny,a,m2,f,m1)
        implicit none
        integer isign, n, m, m1, m2, izero
        integer i, ny
        integer omp_get_num_threads
        real work, tabl
        real a(1:m2,1:m)
        complex f(1:m1,1:m)
!$omp parallel if(isign.ne.0)
!$omp do
        do i = 1, m
           CALL csfft (isign,ny,rny,a(1,i),f(1,i))
        end do
!$omp end do
!$omp end parallel
        return
        end
```

Listing 2. Source code for subroutine csfft.

OpenMP `omp parallel do` on the loop over the calls to `csfft`.

FFTW can be used in a multi-threaded environment, but some care must be taken. The `fftw_execute` routines are thread-safe. The FFTW planner routines, however, share data and should only be called by one thread at a time. We decided to have all the plans be created by only one thread. We can achieve this by specifying the OpenMP IF-clause when creating the parallel region. At runtime, the value of `isign` will be checked. During the initialization phase this value is 0 and the call is executed by only one thread. The calls to `fftw_execute`, which take the bulk of the compute, will be executed in parallel. Since the plan is not modified by `fftw_execute`, it is safe to execute the same plan in parallel by multiple threads. We do not have to worry about MPI thread-safety, since all communication occurs outside of parallel regions. Programs that issue calls to the MPI library within parallel regions need initialize MPI by using `MPI_init_thread` instead of `MPI_init and check whether the required level of thread-safety is provided`.

When running multithreaded MPI codes process, thread and memory placement becomes even more critical than for pure MPI codes. It is essential to ensure that the MPI processes actually use multiple cores. If multiple threads end up running on the same core, the performance would actually decrease when increasing the number of threads. In addition, the cores should be located within the same socket. Accessing data on memory modules located on a different socket, even though possible within the same node, usually has higher latency and lower bandwidth. A challenge which we encountered when running our hybrid test cases was, that at this point there is no standard API to control such a placement and we were dependent on vendor provided means to achieve the desired effect. The Cray XT5, mpirun run command (`aprun`) provides means to do so, the SGI Altix has environment variables, e.g., `DSM_CPULIST` and the `dplace` and `omplace` commands, while on the Sun Constellation Cluster we used the Linux `numactl` command. The `numactl` command provides means to control process and memory placement and is available on generic Linux clusters. It may take some effort to achieve the proper placement and the effect on performance can be enormous. Listing 3 shows some examples how to run hybrid codes on different systems.

Timings for the hybrid MPI/OpenMP code for test case 1 on the Cray XT5 are shown in Figs 5 and 6. The behavior was similar on the Sun Constellation Cluster. The horizontal axis denotes the number MPI processes and the number of OpenMP threads employed. The vertical axis is the time in seconds. Figure 5 shows timings for case 1. The columns are ordered according to the total number of cores with different MPI process and OpenMP thread combinations. Figure 6 shows timings for case 2, using only 1 MPI process per socket and increasing the number of OpenMP threads from 1 to 4. This figure shows the speed-up obtained through

```
Place 4 processes employing 2 threads each on 8 cores
Cray XT5:
aprun -n 4 -N 2 -d 2 ./a.out

SGI Altix:
mpirun -np 4 omplace -nt 2 ./a.out

Generic Linux Cluster assuming 8 cores per node:

procplace.sh:
MPI Proc 0:
numactl --physcpubind=0,1
MPI Proc 1:
numactl --physcpubind=2,3
MPI Proc 2:
numactl --physcpubind=4,5
MPI Proc 4:
numactl --physcpubind=6,7

mpirun procplace.sh -np 4 ./a.out
```

Listing 3. Examples how to achieve proper placement for hybrid runs.

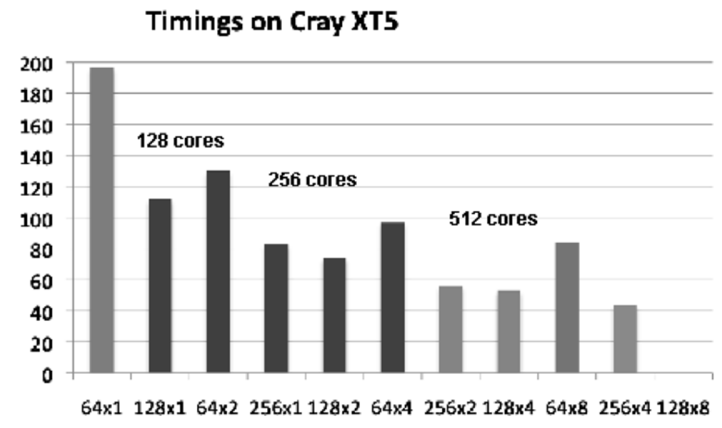

Fig. 5. Test case 1 employing different MPI process and OpenMP thread combinations.

OpenMP parallelization. Depending on hardware platform and test case, we do observe that MPI/OpenMP can outperform pure MPI on the same number of cores. An example is test case 1, running on 256 cores, where 128 processes using 2 threads achieve better performance than 256 single threaded processes. We collected performance statistics on the Cray XT5 using the CrayPat performance analysis tool to explain this behavior. We profiled the following scenarios:

- 128×1 running on 128 cores;
- 128×1 running on 256 cores;
- 128×2 running on 256 cores;
- 256×1 running on 256 cores.

Comparing the profiles in Fig. 7 explains the effect why we observe the significant increase in execution time when placing MPI on all cores within a socket. The figure shows profiles for running 128 MPI processes using 128 and 256 cores. Intuitively we would expect communication to be fastest, when saturating all core with MPI processes, as inter-node communication can take advantage of the available shared memory without using the network. From the profiles

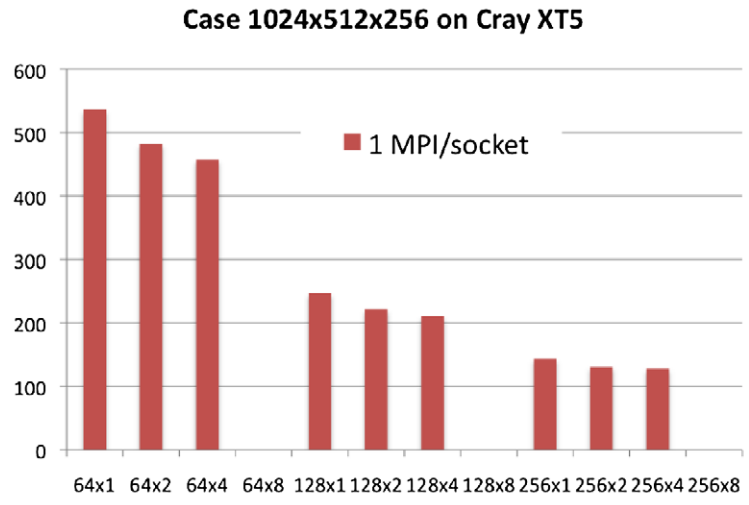

Fig. 6. Test case 2 employing 1, 2 and 4 threads per MPI process. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0308.)

```
 Time %  |         Time   |Group
|-------------------------------------
|  49.7%  |   86.118545  |  USER
||------------------------------------
||  14.8%  |  25.658211  |       main
||   5.2%  |   9.052784  |pcalc_mpi_
||   4.7%  |   8.231422  |dcalc_
||   3.5%  |   6.125497  |csfft_
||   2.8%  |   4.771806  |rvcalc_
||   2.5%  |   4.380827  |scfft_
||=====================================
|  49.0%  |   84.993020  |    MPI
||-------------------------------------
||  24.1%  |  41.848550  |  mpi_waitall_
||  10.6%  |  18.375160  |  mpi_ibsend_
||   7.1%  |  12.314227  |  mpi_bsend_
||   4.8%  |   8.331485  |  mpi_send_
||   2.1%  |   3.722090  |  mpi_irecv_
```

(a)

```
   Time %  |  Time   |Group
|-------------------------------
|   59.8%  |   76.092252  |USER
||------------------------------
||   18.7%  |  23.850167  |     main
||    6.1%  |   7.788667  | pcalc_mpi_
||    5.8%  |   7.438354  | dcalc_
||    4.0%  |   5.155557  | csfft_
||    3.2%  |   4.110129  | rvcalc_
||    2.8%  |   3.553428  | scfft_
||    2.3%  |   2.967025  | swapxy_
||    2.1%  |   2.666762  | drcalc_
||==================================
|   39.2%  |   49.854492  |  |MPI
||----------------------------------
||   16.7%  |  21.253903  | mpi_waitall_
||    8.9%  |  11.281670  | mpi_ibsend_
||    6.3%  |   7.959985  | mpi_bsend_
||    4.5%  |   5.719580  | mpi_send_
|    2.5%  |   3.226080  | mpi_irecv_
```

(b)

Fig. 7. CrayPat performance profile for case 1 on Cray XT5. (a) Performance profile using 128 MPI on 128 cores. (b) Performance profile using 128 MPI on 256 cores.

we learn that the computation time, included in the USER time, decreases from 86 to 76 s when using only 2 cores per socket. This can be explained by resource contention for on socket memory access. The major time decrease, however, is due to a decrease in time spent in MPI routines, which drops from 85 to 50 s, mostly due to reduced time spent in the MPI_Waitall routine. The MPI_Waitall results from the all-to-all type communication, which is necessary for the data redistribution as discussed in Section 2. In Figs 8 and 9 we show the bandwidth of MPI_Alltoall for 16 and 64 MPI processes. The measurements were taken on two of our test platforms. We notice that for both platforms spacing out the MPI processes is advantageous,

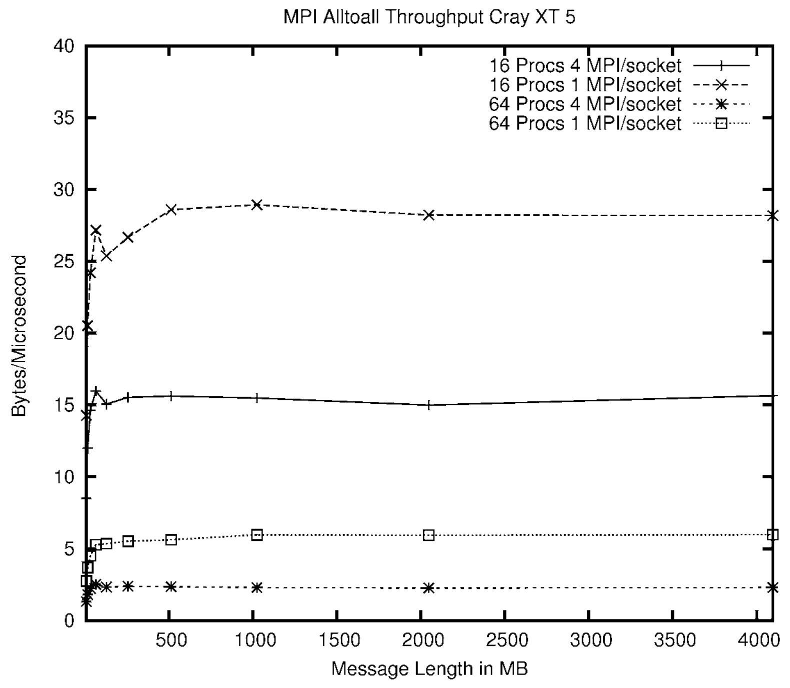

Fig. 8. Comparing 1 MPI vs 4 MPI procs per socket.

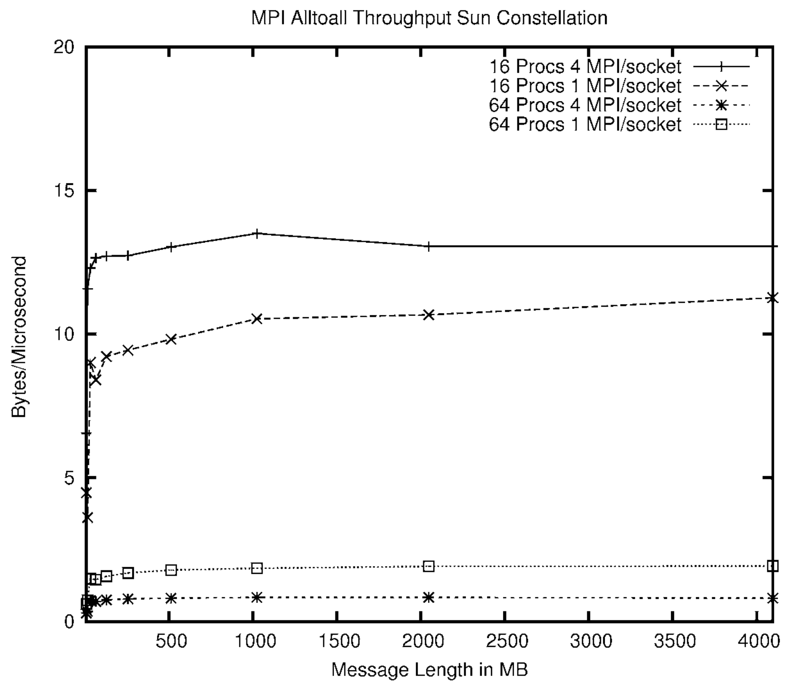

Fig. 9. Comparing 1 MPI vs 4 MPI procs per socket.

whenever network access is involved. On the Sun Constellation Cluster, with 16 cores per node, this is the case for the 64 process run. On the Cray XT5, with only 8 cores per node, the 16 as well as the 64 process run both benefit from not saturating the nodes with processes. From this we conclude that the cause is con-

```
Overall User Time
 Time % |        Time  |Group
 100.0% | 121.517044 | 1782821.4 |Total
|-----------------------------------------
|  57.3% |  70.877576 | 1194165.0 |USER
||----------------------------------------
||   0.5% |  70.716171 | 1194165.0 |pe.68
|||---------------------------------------
3||   0.4% |  70.716171 | 1194165.0 |thread.0
3||   0.1% |  10.409677 | 1191148.0 |thread.1
|||---------------------------------------


CSFFT  User Time
|||---------------------------------------
||   2.5% |   3.039335 |  658944.0 |csfft_
|||----------------------------------------
3||   0.0% |   3.116081 |  658944.0 |pe.68
||||---------------------------------------
4|||   0.0% |   3.049541 |  658944.0 |thread.0
4|||   0.0% |   2.859632 |  658944.0 |thread.1
||||---------------------------------------
```

Listing 4. Profile for 128 processes on 256 cores 2 threads per process. The listing shows process 68 as and example.

tention when multiple MPI processes all require access to the network link of the socket. Reducing the number of MPI processes per node mitigates the contention for network access, as fewer processes compete with each other.

Load-balancing across the threads, when using 2 threads per MPI process on 256 cores is displayed in Listing 4. We show the load-balance for one of the 128 MPI processes. Only 10 s of the 70 s total user time is actually executed by multiple threads. This is due to the fact that we have parallelized only a small number of loops. The execution time for both test cases, however, is spread across many subroutines, which we have not considered for OpenMP parallelization at this point. If we look at `csfft` as an example for a routine executed by multiple threads, we see that the time decreases from 5.1 for single threaded execution to 3.0 on 2 threads. In order to achieve similar speed-up for the whole application, more fine grained-parallelism needs to be exploited.

In Listing 5, we show the performance profile for a run with 256 MPI processes on 256 cores. This run suffers from high MPI overhead due to the already discussed contention for network access.

We also looked into the memory requirements of the 128 and 256 process executions for test case 1. From the Linux `size` command applied to the executables for the 128 (xpir3d.128) and the 256 (xpir3d.256) process runs:

```
xpir3d.128 40.8 MB total size
xpir3d.256  36.7 MB total size
```

per MPI process. This does not include the memory for the distributed data, which is allocated at runtime. The per process requirements for the distributed data are around 54 and 27 MB for the 128 and 256, respectively. The per process memory requirements are somewhat higher for the 128 execution than for the 256 execution, but certainly not by a factor of 2. The reason is that the program, requires many 2D local work arrays, which are replicated on all MPI processes. We monitored the size of the resident data for both executions using the Linux `top` command. It shows 228 MB per process and 184 MB per process for 128 and 256 runs, respectively. The MPI library itself allocates space for communication buffers. All of this yields significantly higher memory requirements the more MPI processes are involved. For test case 2 the memory requirements restricted us to place only 1 MPI process per socket. In this, we gained an extra 10–14% speedup by using multiple OpenMP threads. Timings are shown in Fig 6.

```
Time %  |        Time   |Group
|-----------------------------------
|  61.0%  |  85.577149  |MPI
||------------------------------
||   23.3%  |  32.714120  |mpi_waitall_
||   17.2%  |  24.199623  |mpi_ibsend_
||    9.5%  |  13.276114  |mpi_bsend_
||    7.6%  |  10.708347  |mpi_send_
||    3.1%  |   4.406558  |mpi_irecv_
||----------------------------------------
|  37.2%  |  52.132345  | USER
||----------------------------------------
||   14.9%  |  20.943498  |   main
||    3.6%  |   5.090790  |   pcalc_mpi_
||    2.9%  |   4.088041  |   dcalc_
||    2.1%  |   2.941339  |   csfft_
```

Listing 5. Profile for 256 processes on 256 cores single thread.

4. Related work

The pure MPI implementation of the PIR3D and its performance limitations are the topic of [15]. However, this paper does not discuss the hybrid implementation of the code. There are many papers discussing hybrid parallelism under various aspects. We can only name a few: a very useful set of hybrid benchmark codes from the field of CFD are the multi-zone versions of the NAS Parallel Benchmarks NPB-MZ. They are described in [16] and their performance characteristics are discussed in [7]. Potentials and challenges of hybrid MPI/OpenMP on multi-core node clusters are presented in [5] and [14]. The authors conclude that hybrid parallelism should only be employed if scalability can no be obtained with pure MPI parallelism and that the benefit of hybrid parallelism depends on the particular application. The work presented in this paper complements the previous work in that it discusses a full-scale real-world application and provides detailed performance analysis statistics. Opportunities for exploiting OpenMP task parallelism in MPI programs to overlap communication and computation are discussed in [9]. In the current work we use non-blocking MPI communication for the data swapping. Overlapping communication and computation is not feasible for the application.

5. Conclusions

In this paper, we described how we employed hybrid MPI/OpenMP programming to increase the performance of a full-scale CFD application which was originally parallelized using MPI. We tested the new hybrid code on different compute systems and identified common benefits and challenges when combining the two programming models. Benefits include the decrease in communication overhead by lowering the number of MPI processes, which, in turn, mitigates contention for network access among processes running on the same node. We also noted decrease in memory requirements by lowering the amount of replicated data and nearly linear speed-up of the parallel regions. Challenges we experienced are that the performance depends very much on process, thread and memory placement. The fact that there is currently no standard API to control process and thread placement requires user familiarity with the means of process placement for the particular compute system. Adding OpenMP directives also raises issues regarding the use of run time libraries. Numerical libraries called from within parallel regions need to be thread-safe, which means that the calls to these routines can be issued by multiple threads at the same time without causing side effects. Furthermore, in case the numerical libraries are multithreaded themselves, care has to be taken with regard to how this interacts with the OpenMP parallelization. Within an OpenMP parallel region, for example, the library routine itself should execute in single thread mode. In order to benefit from hybrid MPI/OpenMP parallelism the application needs to have sufficient fine grained parallelism in order to benefit from OpenMP. At this point we have only begun exploiting OpenMP for our application and focused on a very small subset of parallelizable loops. The fact, that we were able to, with very little effort, outperform the pure MPI code, makes us optimistic about this approach.

We will conclude with a set of best practices for exploiting hybrid parallelism, based on our experience with PIR3D. If the application shows good scalability just using MPI we would not recommend adding OpenMP directives unless there are idle cores available. Hybrid parallelization should be considered if the scalability of MPI parallelism is limited, for example by the limited amount of coarse-grained parallelism or inefficient load-balance at MPI level. Another reason is the available of idle cores, which occurred for PIR3D because of high memory requirements. We then recommend obtaining a profile for a relevant set of input data and to investigate whether the most time consuming routines contain a sufficient amount of parallelizable, computational intensive loops. Possible overlap of computation and communication is also a reason to consider hybrid parallelism. One should have an understanding of the hierarchies within the underlying compute system and the means of how to place processes, threads and memory during execution. This placement impacts the performance significantly. The scalability of the OpenMP parallelization within a single node should be ensured. When calling runtime library routines within parallel regions, these routines need to be thread-safe and should run on only a single thread. In summary we conclude that hybrid parallelization is a way to increase the performance when the application exposes a hierarchy of coarse-grained and fine-grained parallelism. It is important that this hierarchy is mapped correctly onto the hierarchy within the underlying hardware platform.

Acknowledgements

This work was funded by the Office of Naval Research (ONR), the National Aeronautics and Space Administration (NASA) and NorthWest Research Associates, Inc. (NWRA). Support was also provided by the DoD High Performance Computing Modernization Program (HPCMP), User Productivity Enhancement, Technology Transfer and Training (PETTT) program. We do thank the Engineer Research and Development Center Major Shared Resource Center, Vicksburg, MS, USA for providing computional resources and support on the Cray XT5 and SGI Altix systems. Our gratitude also goes to the Texas Advanced Computing Center (TACC) of the University of Texas at Austin which supported us by providing compute time and support for our experiments on the Sun Constellation Cluster.

References

[1] Argonne National Laboratories, MPICH2, 2010, available at: http://www.mcs.anl.gov/research/projects/mpich2/.

[2] B. Chapman, G. Jost and R. van der Pas, *Using OpenMP*, MIT Press, Cambridge, MA, 2007.

[3] J. Ferzinger and M. Peric, *Computational Methods for Fluid Dynamics*, Springer, New York, NY, 2002.

[4] K. Goto and R.A. van de Geijn, Anatomy of high-performance matrix multiplication, *ACM Transactions on Mathematical Software* **34**(3) (2008), Article 12.

[5] G. Hager, G. Jost and R. Rabenseifner, Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes, in: *Proceedings of CUG 09*, Atlanta, GA, USA, May 4–7, 2009.

[6] F. Holzaepfel, Adjustment of subgrid-scale parameterizations to strong streamline curvature, *AIAA Journal* **42** (2004), 1369–1377.

[7] H. Jin and R.F. Van Der Wijngaart, Performance characteristics of the multi-zone NAS parallel benchmarks, *Journal of Parallel and Distributed Computing* **66** (2006), 674–685, Special Issue: 18th International Parallel and Distributed Processing Symposium.

[8] L. Koesterke and K.F. Milfeld, How does an asymmetric node architecture affect applications? A simple method to gauge the effects of NUMA on load-balancing, in: *2009 TeraGrid Conference*, Arlington, VA, USA, 2009.

[9] A. Koniges, R. Preissl, J. Kim, D. Eder, A. Fisher, N. Masters, V. Mlaker, S. Ethier, W. Wang, M. Head-Gordon and N. Wichmann, Application acceleration on current and future cray platforms, in: *Proceedings of CUG 2010*, Edingburgh, 24–27 May, 2010.

[10] K.F. Milfeld, L. Koesterke and K.W. Schulz, Parallel communications and NUMA control on Teragrid's New Sun Constellation system, in: *2008 TeraGrid Conference*, NV, USA, 2008.

[11] MPI Forum, MPI-2.2 documents, 2010, available at: http://www.mpi-forum.org/docs/docs.htm.

[12] Ohio State University, 2010, MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE, available at: http://mvapich.cse.ohio-state.edu/.

[13] OpenMP Architexture Review Board, 2010, The OpenMP API specification for parallel programming, available at: http://openmp.org/wp/.

[14] R. Rabenseifner, G. Hager and G. Jost, Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes, in: D.E. Baz, F. Spies and T. Gross, eds, in: *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2009*, Weimar, Germany, 18–20 Febuary, 2009, IEEE Computer Society, pp. 427–436.

[15] R. Robins and G. Jost, Parallelization of a vector-optimized 3-D flow solver for multi-core node clusters, in: *Proceedings of UGC10*, June 2010.

[16] R.F. Van Der Wijngaart and H. Jin, NAS parallel benchmarks, multi-zone versions, NAS Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA, USA, 2003.