# Early Experiments with the OpenMP/MPI Hybrid Programming Model

Ewing Lusk[1]* and Anthony Chan[2]

[1] Mathematics and Computer Science Division
Argonne National Laboratory
[2] ASCI FLASH Center
University of Chicago

**Abstract.** The paper describes some very early experiments on new architectures that support the hybrid programming model. Our results are promising in that OpenMP threads interact with MPI as desired, allowing OpenMP-agnostic tools to be used. We explore three environments: a "typical" Linux cluster, a new large-scale machine from SiCortex, and the new IBM BG/P, which have quite different compilers and runtime systems for both OpenMP and MPI. We look at a few simple, diagnostic programs, and one "application-like" test program. We demonstrate the use of a tool that can examine the detailed sequence of events in a hybrid program and illustrate that a hybrid computation might not always proceed as expected.

## 1 Introduction

Combining shared-memory and distributed-memory programming models is an old idea [21]. One wants to exploit the strengths of both models: the efficiency, memory savings, and ease of programming of the shared-memory model and the scalability of the distributed-memory model. Until recently, the relevant models, languages, and libraries for shared-memory and distributed-memory architectures have evolved separately, with MPI [7] becoming the dominant approach for the distributed-memory, or message-passing, model, and OpenMP [9, 15] emerging as the dominant "high-level" approach for shared memory with threads. We say "high-level" since it is higher level than the POSIX `pthread` specification [8]. We use quotation marks around the expression because OpenMP is not as high level as some other proposed languages that use models with a global view of data [1, 19].

* This work was supported in part by the U.S. Department of Energy Contract #B523820 to the ASC/Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago and in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Departent of Energy, under Contract DE-AC02-06CH11357.

Recently, the hybrid model has begun to attract more attention, for at least two reasons. The first is that it is relatively easy to pick a language/library instantiation of the hybrid model: OpenMP plus MPI. While there may be other approaches, they remain research and development projects, whereas OpenMP compilers and MPI libraries are now solid commercial products, with implementations from multiple vendors. Moreover, we demonstrate in this paper that OpenMP and MPI implementations on significant platforms work together as they should (see Section 2.) The second reason is that scalable parallel computers now appear to encourage this model. The fastest machines now virtually all consist of multi-core nodes connected by a high speed network. The idea of using OpenMP threads to exploit the multiple cores per node (with one multithreaded process per node) while using MPI to communicate among the nodes appears obvious. Yet one can also use an "MPI everywhere" approach on these architectures, and the data on which approach is better is confusing and inconclusive. It appears to be heavily dependent on the hardware, the OpenMP and MPI implementations, and above all on the application and the skill of the application writer.

We do not intend to settle here the question of whether the hybrid approach is good or bad. Postings to assorted discussion lists claim to have proven both positions. Instead we describe three interesting environments in which this question can be studied, and present some preliminary experiments.

Considerable work has gone into studying the hybrid model. Some examples can be found in [12, 18, 20]. What is new in this paper is 1) a discussion of the relationship between the MPI standard and the OpenMP standard, 2) presentation of early results on two brand-new machines that support the hybrid model in an efficient way, and 3) depiction of a particular performance visualization tool looking at hybrid codes.

The rest of the paper is organized as follows. In Section 2 we describe aspects of MPI and OpenMP that pertain to their use with each other. In Section 3 we describe a performance visualization tool and recent enhancements to it that enable it to be used to study the behavior of hybrid programs. Section 4 describes the three environments that we used for our experiments, and early results from our experiments are in Section 5. We present some conclusions and plans for future work in Section 6.

## 2 What the Standards Say

Hybrid programming with two portable APIs would be impossible unless each made certain commitments to the other on how they would behave. In the case of OpenMP, the important commitment is that if a single thread is blocked by an operating system call (such as file or network I/O) then the remaining threads in that process will remain runnable. In our situation, this means that an MPI call that may block, such as `MPI_Recv` or `MPI_Wait`, will only block the calling thread and not the entire process. This is a significant commitment, since it involves

the thread scheduler in the compiler's runtime system and interaction with the operating system.

The commitments made by the MPI standard are more complex. The MPI-2 standard defines four levels of thread safety. These are in the form of what commitments the application makes to the MPI implementation.

`MPI_THREAD_SINGLE` There is only one thread in the application.
`MPI_THREAD_FUNNELED` There is only one thread that makes MPI calls.
`MPI_THREAD_SERIALIZED` Multiple threads make MPI calls, but only one at a time.
`MPI_THREAD_MULTIPLE` Any thread may make MPI calls at any time.

An application can find out at run time which level is supported by the MPI library it is linked with by means of the `MPI_Init_thread` function call.

These levels correspond to the use of MPI in an OpenMP program in the following way:

`MPI_THREAD_SINGLE` There is no OpenMP multithreading in the program.
`MPI_THREAD_FUNNELED` All of the MPI calls are made by the master thread. This will happen if all MPI calls are outside OpenMP parallel regions or are in master regions. A thread can determine if it is the master thread with the `MPI_Is_thread_main` call. More precisely, it determines whether it is the same thread that called `MPI_Init` or `MPI_Init_thread`.
`MPI_THREAD_SERIALIZED` The MPI calls are made by only one thread at a time. This can be enforced in OpenMP by a construction like:

```
#pragma omp parallel
...
#pragma omp single
{
    ...MPI calls allowed here...
}
```

(as long as nested parallelism is not used.)
`MPI_THREAD_MULTIPLE` MPI calls can be made anywhere, by any thread.

All MPI implementations of course support `MPI_THREAD_SINGLE`. The nature of typical MPI implementations is such that they probably also support `MPI_THREAD_FUNNELED`, even if they don't admit it by returning this value from `MPI_Init_thread`, presuming they use a thread-safe malloc and other system calls, likely in any OpenMP application. Usually when people refer to an MPI implementation as "thread safe" they mean at the level of `MPI_THREAD_MULTIPLE`. It is worth noting that OpenMP encourages a style of programming that only requires `MPI_THREAD_FUNNELED`, so hybrid programming does not necessarily require a fully "thread safe" MPI.

## 3 Visualizing the Behavior of Hybrid Programs

Over the years we have found it surprisingly difficult to predict the behavior of complex MPI programs, and have developed a number of tools to assist in the process of understanding them as an important step in tuning them for performance. Understanding complex hybrid programs will be even more difficult. We describe here an extension to an existing tool to the hybrid case.

### 3.1 Jumpshot

Jumpshot [5, 22] is a parallel program visualization program that we have long used to examine the detailed behavior of MPI programs. It provides a "Gantt chart" view of time lines of parallel processes, with colored rectangles to indicate the state of a process over a particular time interval. It also uses arrows from one line to another to indicate messages. While not scalable to very large numbers of processes (say, greater than 512), Jumpshot's panning and zooming capabilities, coupled with its summary views allowing a wide range of time scales to be viewed, have made it a valuable tool for studying the detailed behavior of parallel programs.

Jumpshot displays data from SLOG2 [14] files, which are written in an efficient way during the course of a parallel program execution. The library for logging events is provided by the MPE package [4] distributed with MPICH2 [6]. Both automatic logging of MPI calls via the MPI profiling interface and user-defined states and events are provided.

### 3.2 Jumpshot and Threads

Recently MPE, SLOG2 and Jumpshot have been extended to allow visualization of multi-threaded, and hence hybrid, programs. We made the assumption that the threads calling the MPI function or the MPE logging routines directly were POSIX `pthreads`, so that we could use the `pthread` library for the mutexes required as multiple threads wrote to the same memory buffer containing the logging records. MPE and SLOG2 were modified to include thread ID's in the log records.

Jumpshot needed to be augmented so that separate time lines for separate threads would be shown. Controls were added to the Jumpshot display to allow threads to be 1) collapsed into their parent processes, 2) grouped with their parent processes, or 3) grouped into separate communicators. (A common application structure in a program requiring `MPI_THREAD_MULTIPLE` is for separate threads in a process to use separate communicators.) The rest of the figures in this paper are screenshots of Jumpshot viewing SLOG2 files created by MPE logging of hybrid programs.

The first step in determining whether Jumpshot could be used with OpenMP was to determine whether the threads created by OpenMP compilers really were POSIX pthreads, which MPE, SLOG2, and Jumpshot had already been modified to handle. Fortunately, in all three of the environments described here, a simple

diagnostic program, in which OpenMP threads created by `#pragma omp parallel` used the POSIX interface to request their `pthread` id's, was able to prove that this was the case, and that thread id's were reused in multiple parallel regions. This meant that no additional work was needed to adapt the MPE-SLOG2-Jumpshot pipeline to OpenMP. While most OpenMP implementations "park" threads like this, the situation could change for some emerging architectures that support fast thread creation.

Figure 1 is a Jumpshot view of a program with simultaneously communicating threads that demonstrates that MPICH2 is thread safe at the level of `MPI_THREAD_MULTIPLE`.

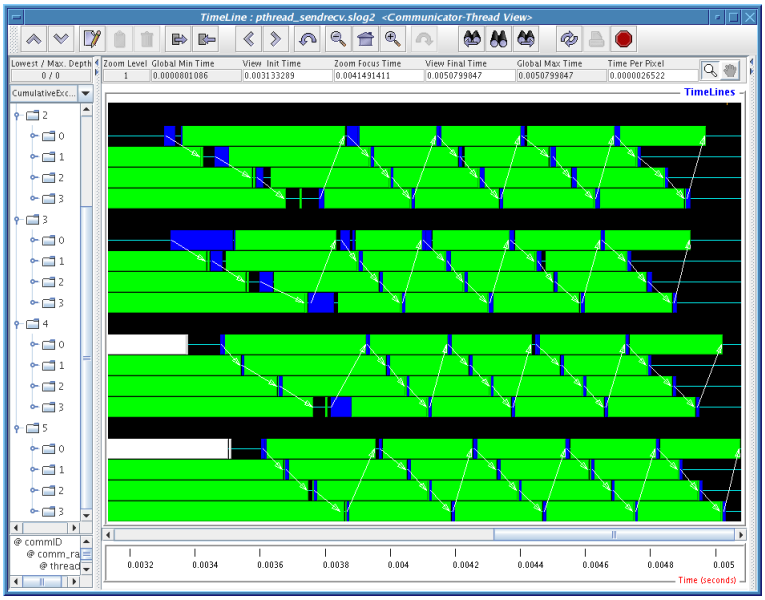



**Fig. 1.** Jumpshot's Communicator-Thread view of an MPI/pthreads program that forms a send and receive ring in the subcommunicator created from one single thread of each process.

Other tools, such as Paraver [13, 17], Vampir [11], and TAU [3] are also available for visualizing hybrid programs. We focus here on the use of Jumpshot because of its ability to show extreme detail and its wide availability, in particularly on the platforms presented here. Jumpshot is included as an optional viewer in recent releases of TAU.

## 4 The Hybrid Environments

The three machine/compiler environments that we tested were different in both hardware and software. All are quite new. The first is a "standard" Linux cluster

based on AMD dual dual-core nodes. The latter two are examples of low-power, highly scalable computers. Our intention was not to explicitly compare the performance of these machines, but rather to demonstrate the viability of the hybrid programming model consisting of OpenMP and MPI on them, and to demonstrate the Jumpshot tool in these environments.

The details on the three platforms are as follows:

**Linux cluster** Each node is dual Opteron dual-core 2.8Ghz, i.e. four 2.8 Ghz cores. The Intel 9.1 fortran compiler, `ifort`, is used with MPICH2-1.0.6p1 nemesis channel, which uses shared memory for intranode MPI communication and provides `MPI_THREAD_MULTIPLE` support. This is a typical Linux cluster. Ours has multiple networks, but the experiments were done on Gigabit Ethernet.

**IBM BG/P** In IBM's BlueGene/P system [2], each compute node consists of four PowerPC 850 Mhz cores. IBM's XLF 11.1 fortran cross-compiler, `bgxlf_r`, is used with BlueGene MPI version V1R1M2 which provides `MPI_THREAD_MULTIPLE` support. The BlueGene system has a high-performance 3-D torus network for point-to-point communication.

**SiCortex SC5832** The SiCortex SC5832[10] consists of 972 six-way SMP compute nodes, i.e six MIPS 500 Mhz cores per node. The Pathscale 3.0.99 fortran cross-compiler, `scpathf95`, is used with the SiCortex MPI implementation, which provides `MPI_THREAD_FUNNELED` support. The SiCortex machine has a high-performance Kautz network.

All the Fortran compilers mentioned above provide OpenMP support for Fortran. In addition, the companion C compilers provide OpenMP support for C.

## 5 Experiments

We did two sorts of experiments: first, a basic exploration of how things work, and then two of the NAS parallel benchmarks, in order to investigate programs amenable to the hybrid model for improving performance. The BG/P and SiCortex machines that we used are so new that they are still being tuned by the vendors, so we focused on behavior rather than scalability of performance. Even on the large machines we used a small number of nodes.

### 5.1 Basic tests

We wrote a simple hybrid Fortran program and instrumented it with MPE. The core of the program looks like this:

```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(ii, jj, sum, ierr)
!$OMP DO
      do ii = 1, imax
         ierr = MPE_Log_event( blkA_startevt, 0, '' )
         sum = 0.0
```

```
         call random_number( frands )
         do jj = 1, jmax
            sum = sum + frands(jj) * jj
         enddo
         ierr = MPE_Log_event( blkA_finalevt, 0, '' )
      enddo
!$OMP END DO nowait
!$OMP END PARALLEL
```

This loop is repeated three times, to check consistency of thread ids, with three different event ids in the `MPI_Log_event` calls. The MPI calls are outside the parallel regions, so `MPI_THREAD_FUNNELED` is sufficient. Also, no MPI calls are made in the `MPE_Log_event` calls, so while pthread locks are used there, `MPI_THREAD_MULTIPLE` is not needed.

### 5.2 NAS Benchmarks

To initiate our study of hybrid programs, we choose a family of parallel benchmarks that have been written to take advantage of the hybrid style, the NAS parallel multi-zone benchmarks, NPB-MZ-MPI, version 3.1 [16], as the base code for our experiments. Two application benchmarks in NPB-MZ, namely BT and SP, were compiled with the respective OpenMP Fortran compiler on the AMD Linux cluster, the IBM Blue Gene P, and the SiCortex SC5832. We ran the codes in two different sizes (W and B) and different "modes" i.e., 16 MPI processes on four multi-core nodes and with four processes on four nodes, with each process having four threads (and with six each on the SiCortex). Note that the structure of NPB-MZ-MPI is such that it does not require `MPI_THREAD_MULTIPLE`.

From the outset it was clear that a tool capable of showing considerable detail would be useful in understanding what these programs were actually doing. For example, Figure 2 shows a short interval of execution on the Linux cluster. It is clear that although process 1 has 4 threads (actually 6, in this case) active at one time or another, something is preventing complete 4-way parallelism. (We would expect the red and green states to be stacked 4 deep, instead of offset in pairs the way they are.) This turned out to be a consequence of the way we had set the environment variables `OMP_NUM_THREADS` and `NPB_MAX_THREADS`, which had the side effect of deactivating the thread load-balancing algorithm in BT. Without Jumpshot we might not have realized that something was wrong. Note that we can see that MPI communication is being done only by the first thread in each process. The "extra" threads that appear here are created by the load-balancing code in the BT benchmark, which overrides the `OMP_NUM_THREADS` environment variable set by the user. In later runs we controlled this with `NPB_MAX_THREADS`. Jumpshot alerted us to this anomaly. Results of our experiments are shown in Table 1.

The benchmarks were compiled with maximum optimization level known on each platform, i.e. -O3 with `ifort` on AMD Linux cluster, -O5 with `bgxlf_r` on BG/P and -O3 with `scpathf95` on SiCortex. All these experiments are performed on 4 nodes on each chosen platform with either one process per core

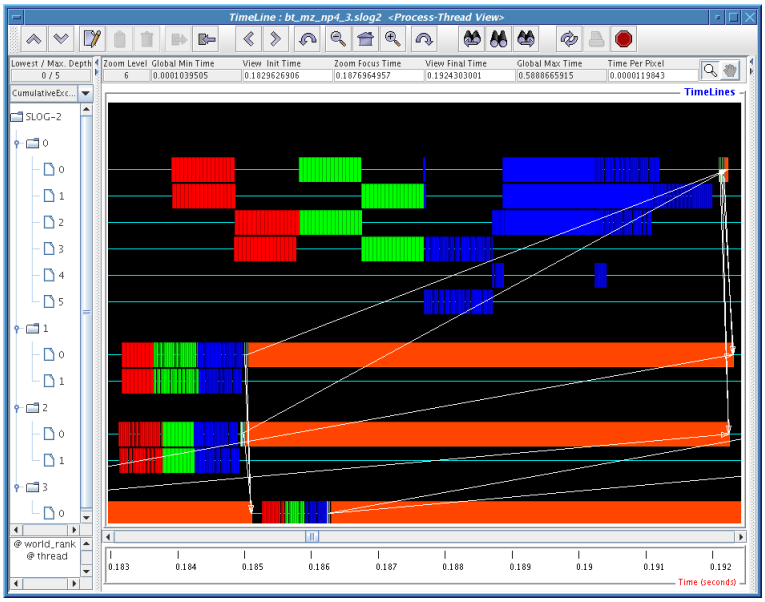



**Fig. 2.** Blocked Parallelism captured by Jumpshot

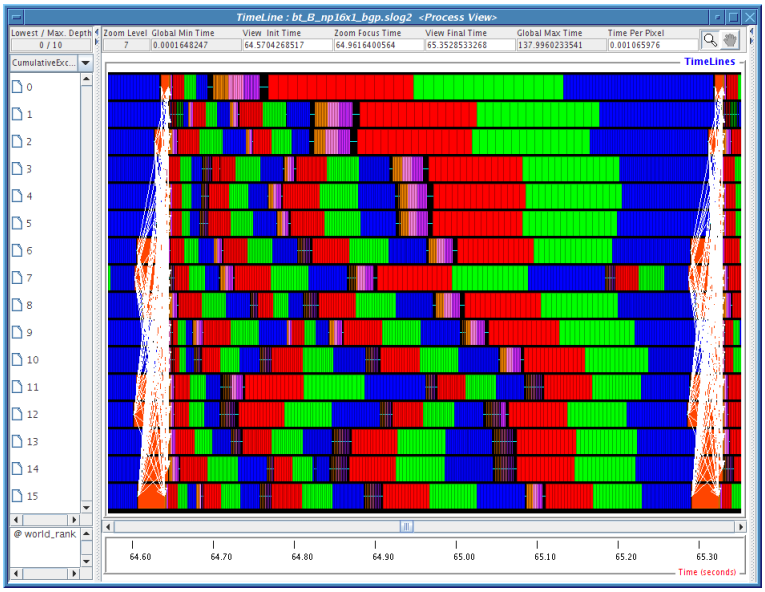
or one thread per core. For instance 16x1 refers to 16 processes running with 1 thread per process on 4 nodes, and 4x4 refers to 4 processes running with 4 threads per process on 4 nodes.

Here are a few general observations.

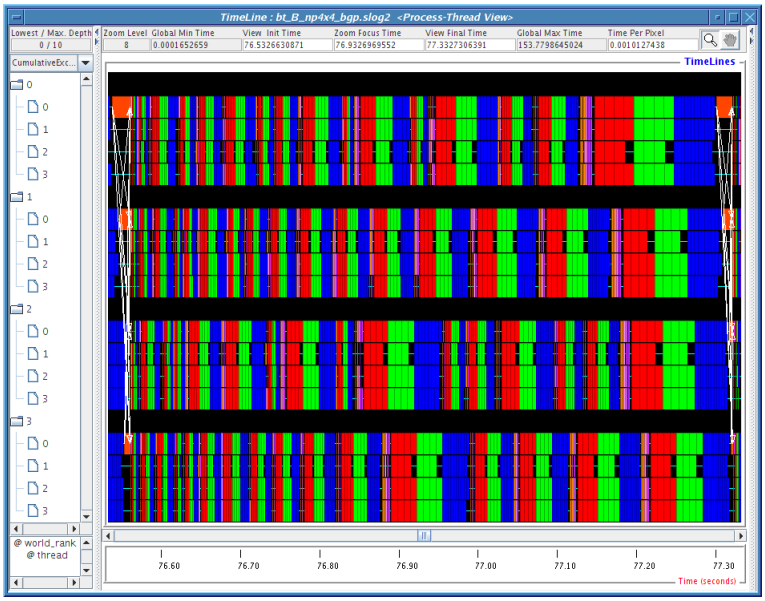
- The hybrid approach provides higher performance on small (size W) version of BT on all three of these machines, as message-passing time dominated.
- For SP, even at size W, "MPI everywhere" was better.
- On the size B problems, the "MPI everywhere" model was better than the hybrid approach.
- On the Sicortex only, we also ran 6 processes or threads per node, on 4 nodes, since each has 6 cores. The overall time dropped, showing the effect of applying more CPUs, but the machine still preferred the MPI everywhere model to the hybrid model.

Figure 3 shows the difference on BG/P between processes and threads. Subfigures a) and b) show similar time intervals.

Figure 4 shows the effects of running various numbers of threads in processes on the SiCortex nodes, including more threads than cores, on our basic threading test program (not NPB). As the number of threads increases over the number of physical cores, the speed improvement due to parallelization of OpenMP threads does not seem to be diminishing yet. The reason could be because the SiCortex machine has yet to be fully optimized, so that extra CPU cycles are available for extra work, letting it appear to have more physical cores than there are.




(a) np16x1, ViewDuration=0.7825s

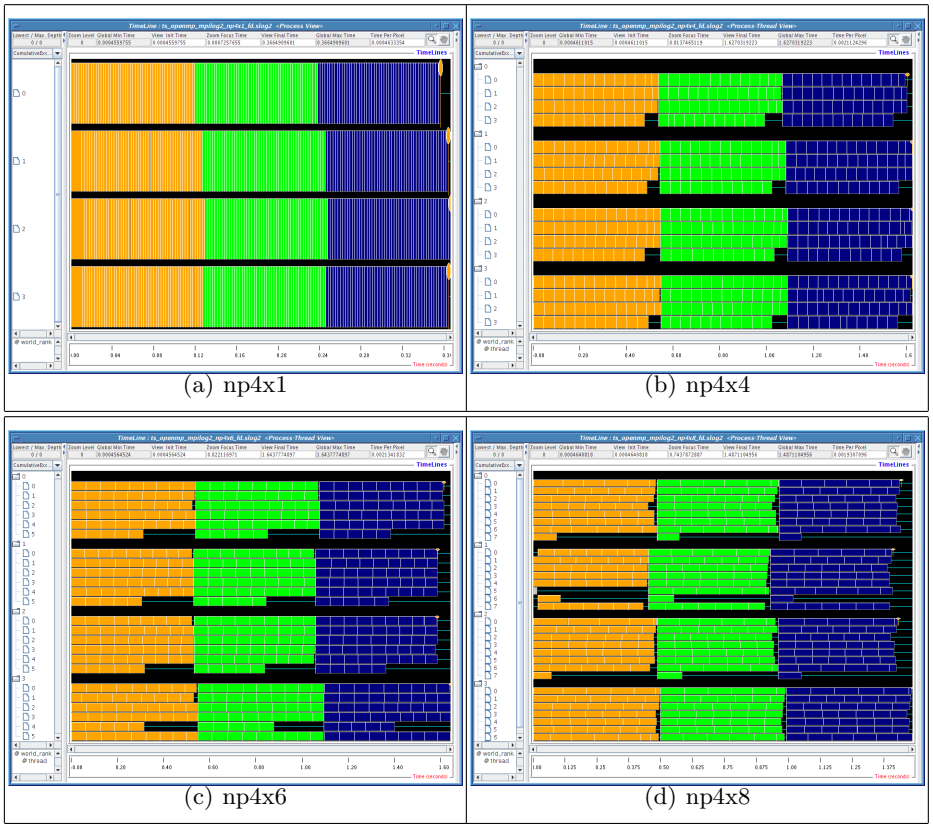



(b) np4x4, ViewDuration=0.8000s

**Fig. 3.** Jumpshot pictures of BT class B running on 4 BG/P nodes with either 1 process or 1 thread per core. (a) np16x1, ViewDuration=0.7825s. (b) np4x4, ViewDuration=0.8000s.

| | AMD cluster | BG/P | SiCortex |
|---|---|---|---|
| bt-mz.W.16x1 | 1.84 | 9.46 | 20.60 |
| bt-mz.W.4x4 | 0.82 | 3.74 | 11.26 |
| sp-mz.W.16x1 | 0.42 | 1.79 | 3.72 |
| sp-mz.W.4x4 | 0.78 | 3.00 | 7.98 |
| bt-mz.B.16x1 | 24.87 | 113.31 | 257.67 |
| bt-mz.B.4x4 | 27.96 | 124.60 | 399.23 |
| sp-mz.B.16x1 | 21.19 | 70.69 | 165.82 |
| sp-mz.B.4x4 | 24.03 | 81.47 | 246.76 |
| bt-mz.B.24x1 | | | 241.85 |
| bt-mz.B.4x6 | | | 337.86 |
| sp-mz.B.24x1 | | | 127.28 |
| sp-mz.B.4x6 | | | 211.78 |

**Table 1.** NPB-MZ benchmark results are shown in seconds. The row labels are written in the form of <benchmark name>.<class name> .<process count> x <thread count>. Where benchmark name is either bt-mz or sp-mz, and class name is either W or B.



(a) np4x1 (b) np4x4

(c) np4x6 (d) np4x8

**Fig. 4.** Jumpshot pictures of the basic fortran program with `OMP_NUM_THREADS`=1, 4, 6, and 8 on 4 SiCortex nodes.

## 6 Conclusions and Future Work

Our principal conclusion is that the hybrid programming model represented by the OpenMP and MPI standards is now available on the newest entries in the list of scalable high-performance computers as well as on traditional clusters. Jumpshot is no doubt only one of the tools available for inspecting the detailed behavior of hybrid programs, but so far few are both portable and freely available. Thus the pieces are in place, even on some of the largest and newest computers in the high performance computing complex, for application developers to create applications using this approach.

We have already begun extending the work presented here to other benchmarks at larger scale and to begin developing useful benchmarks specialized for the hybrid approach. Lack of space has limited us to the preliminary experiments presented here, but these show a promising beginning to a more thorough study.

## References

1. http://crd.lbl.gov/~parry/hpcs_resources.html.
2. http://www-03.ibm.com/servers/deepcomputing/bluegene.html.
3. http://www.cs.uorigon.edu/research/tau.
4. http://www.mcs.anl.gov/perfvis/download/index.htm#MPE.
5. http://www.mcs.anl.gov/perfvis/software/viewers/index.htm#Jumpshot-4.
6. http://www.mcs.anl.gov/research/projects/mpich2.
7. http://www.mpi-forum.org/docs/docs.html.
8. http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html.
9. http://www.openmp.org/blog/specifications.
10. http://www.sicortex.com/products/sc5832.
11. http://www.vampir.eu.
12. Freack Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks. In *Proceedings of SuperComputing 2000*. IEEE Computer Society, 2000.
13. Jordi Caubet, Judit Gemenez, Jes Labarta, Luiz DeRose, and Jeffrey Vetter. A dynamic tracing mechanism for performance analysis of (openmp applications. In *Proceedings of WOMPAT'01*, 2001.
14. Anthony Chan, Ewing Lusk, and William Gropp. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. 2008, in press.
15. Barbara Chapman, Gabrielle Jost, and Ruud van der Pas. *Using OpenMP*. MIT Press, 2007.
16. R. F. Van der Wijingaart and H. Jin. the NAS parallel benchmarks, multi-zone versions. NAS Technical Report NAS-03-010, NASA Ames Research Center, 2003. http://www.nas.nasa.gov/Resources/Software/npb.html.
17. F. Freitag, J. Caubet, and J. Labarta. A trace-scaling agent for parallel application tracing. In *IEEE International Converence on Tools with Artivficial Intelligence (ICTAI)*, pages 494–499, 2002.
18. Gabriele Jost, Haoquiang Jin, Dieter an Mey, and Ferhat F. Hatay. Comparing the openmp, mpi and hybrid programming paradigms on an SMP cluster. In *Proceedings of EWOMP'03*, 2003. http://www.nas.nasa.gov/News/Techreports/2003/PDF/nas-030019.pdf.

19. E. Lusk and K. Yelick. Languages for high-productivity computing: the DARPA HPCS language project. *Parallel Processing Letters*, 17(1):89–102, 2001.
20. Rolf Rabenseifner. Hybrid parallel programming on HPC platforms. In *Proceedings of EWOMP'03*, 2003. http://www.compunity.org/events/ewomp03/omptalks/Tuesday/Session7/T01.pdf.
21. Thomas Sterling, Paul Messina, and Paul H. Smith. *Enabing Technologies for Petaflops Computing*. MIT Press, 1995.
22. Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.