Experiences in Tuning Performance of Hybrid MPI/OpenMP Applications on Quad-core Systems

Ashay Rane and Dan Stanzione Ph.D.
{ashay.rane, dstanzi}@asu.edu

Fulton High Performance Computing Initiative, Arizona State University

Abstract. The Hybrid method of parallelization (using MPI for inter-node communication and OpenMP for intra-node communication) seems a natural fit for the way most clusters are built today. It is generally expected to help programs run faster due to factors like availability of greater bandwidth for intra-node communication. However, optimizing hybrid applications for maximum speedup is difficult primarily due to inadequate transparency provided by the OpenMP constructs and also due to the dependence of the resulting speedup on the combination in which MPI and OpenMP is used. In this paper we mention some of our experiences in trying to optimize applications built using MPI and OpenMP. More specifically, we talk about the different techniques that could be helpful to other researchers working on hybrid applications. To demonstrate the usefulness of these optimizations, we provide results from optimizing a few typical scientific applications. Using these optimizations, one hybrid code ran up to 34% faster than pure-MPI code.

1 Introduction

MPI (the Message Passing Interface) and OpenMP are the de-facto standards when writing parallel programs. MPI provides an explicit messaging model with no assumption of shared memory, and as such is the standard for use on distributed memory systems. OpenMP provides a threading model, with implicit communication and the assumption of shared memory, and therefore is often used on shared memory systems. Hybrid programs are those that use both MPI and OpenMP – MPI for communication between nodes and OpenMP for communication within a single node. For the last several years, most clusters have been composed of a collection of multi-core nodes, with shared memory at the node level, and distributed memory between nodes. This appears to fit well with the hybrid model. The other, more compelling reason of using both MPI and OpenMP is that communication by means of shared memory is known to support much greater bandwidth as opposed to communication using messages [1]. There have been efforts [2] to make MPI leverage the shared memory architecture on a node. However, from our experience hand-tuning using OpenMP gives greater benefits than relying on the techniques in the MPI distribution alone. Similarly, efforts like those extending OpenMP to clusters [3] failed to give performance equivalent to that of hand-tuned MPI+OpenMP code.

[4] compares the maximum memory-to-CPU bandwidth that can be utilized when using MPI over shared memory in one case and when using Pthreads in the other case. Although it does not compare OpenMP bandwidth, one can get an idea of the potential benefit of using a threading model like OpenMP over

using MPI on shared memory, from the given comparison. The average ratio of the maximum available bandwidth when using threading to that available when using MPI is about 4, with the peak ratio being about 14. Even in the worst case, a Pthreads-based program has twice as much bandwidth available as an MPI-based program. As mentioned in [4] this happens because even when using shared memory, MPI communication involves at least one buffer copy operation which would not be present when threads access a shared segment of memory directly.

However, a straight-forward integration of OpenMP constructs into the MPI program often does not give good speedup results. Using OpenMP to get good speedup values can be difficult due to the inadequate transparency provided by the language constructs. With the programmer specifying just a single statement like "`#pragma omp parallel for`", the compiler and the runtime system together have to find the optimum way of parallelizing the given loop. While this simplicity in specifying the OpenMP construct is good for the programmer, it is difficult to optimize. This simple syntax is complemented by additional library routines to pass hints to the compiler and the runtime system. These include means to set the thread stack size, allowing or disallowing nested parallelism, setting the manner in which workload is distributed among threads, etc [5]. As an aside, one may note that the growing number of such supporting library routines (10 in OpenMP version 1.0, 22 in OpenMP version 2.0 and 31 in OpenMP version 3.0) is an indication of the growing complexity and detail that needs to be added to the OpenMP specification.

Nevertheless, even with the provision of these supporting routines, it becomes necessary to resort to tools and techniques outside of the OpenMP specification. This paper describes the optimization techniques that we learned while working with hybrid applications.

2 Cluster setup and experimental methodology

The test cluster on which the programs were run was the Saguaro 2 cluster, which comprises of Dual quad-core Intel Xeon processors (E4330 ("Harpertown") cores). Thus each node has eight cores arranged as part of two sockets. Each core has 32 KB of Level 1 instruction cache and 32 KB of Level 1 data cache available to it. Level 2 cache of size 4 MB is shared between a pair of cores. The nodes themselves are connected via Dual data rate Infiniband links with 16 Gbps peak bandwidth. The Intel C/C++ compiler 10.1 was used for compiling the pure-MPI and the hybrid MPI+OpenMP programs. The MPI distribution used for running the different pure-MPI and hybrid programs and for the tests concerning message size was MVAPICH 1.0.1. We preferred using MVAPICH-1 over MVAPICH-2 because most recent multicore optimization has been on MVAPICH-1. MVAPICH-1 is known to support shared memory communication on multi-core processors instead of unnecessarily traversing the network stack.

For each run of a pure-MPI program, we ran one MPI task on each available core, i.e. 8 MPI tasks per node in our case. In contrast, for running hybrid MPI/OpenMP programs, we ran one MPI task per node and used as many OpenMP threads as the number of cores available per node, i.e. 1 MPI task and 8 OpenMP threads for each node.

3 Optimization techniques

3.1 Minimizing OpenMP `parallel` overhead

Scientific codes usually exhibit a "big" outer loop that repeats individual steps of the computation. This loop may represent the time steps of a simulation or continued execution of the body of the loop till the results of the computation converge, etc. In such cases, inserting "`#pragma omp parallel for`" directives for the inner loops doing the computation of a single step is probably the most natural way of parallelizing the code. However, although creation of the threads is performed only once in the program's lifetime, each `parallel for` directive introduces the overhead of waking the threads at the beginning of the `parallel` block. When this is repeated over multiple iterations of the outer loop, this overhead increases too. As mentioned in [6], the Intel OpenMP runtime uses thread pooling to eliminate repeated creation and destruction of threads and thus tries to reduce the overhead. However, even with this provision, we observed that we could reduce the running time of the program by hoisting the `parallel` pragma outside of the outer loop.

To eliminate the repeated overhead mentioned above, we hoisted the `parallel` directive outside the loop. Thus the outer loop is executed by all threads independently. Synchronization between the threads can be done using the OpenMP `single` and `master` directives as illustrated in Figure 1. Also, care needs to be taken to explicitly mark variables as either private to each thread or shared among the threads and what should be their initial value, if any.

```
for (...)
{
        // Initialization
        {
                ...
        }

        // Computation
        #pragma omp parallel for
        for (...)
        {
                ...
        }

        // Communication
        {
                MPI_Send (...);
                MPI_Recv (...);
        }
}
```

```
#pragma omp parallel private (...)
{
        for (...)
        {
                #pragma omp single
                {
                        // Initialization
                        ...
                }

                // Computation
                #pragma omp for
                for (...)
                {
                        ...
                }

                #pragma omp barrier
                #pragma omp master
                {
                        // Communication
                        MPI_Send (...);
                        MPI_Recv (...);
                }

                #pragma omp barrier
        }
}
```

Fig. 1. Minimizing `parallel` overhead

However, in such cases one needs to ensure that the MPI runtime is initialized in a thread-safe manner. For this, this MPI initialization needs to be done using `MPI_Init_thread()` [11] instead of `MPI_Init()`. Thread-safety in MPI is available only as part of the MPI-2 standard.

```
int MPI_Init_thread(int *argc, char *((*argv)[]), int required,
                    int *provided)
```

The value of `required` for the above mentioned case can be:

- `MPI_THREAD_FUNNELED`: Only the master thread in a process will make calls to MPI routines
- `MPI_THREAD_SERIALIZED`: Multiple threads may make calls to MPI routines, but only one at a time
- `MPI_THREAD_MULTIPLE`: Multiple threads may call MPI routines with no restrictions

`provided` returns the highest level of thread support.

The code shown in Figure 1 is a representative of the equivalent hybrid code when `MPI_THREAD_FUNNELED` is the maximum thread support available. Since it is only the master thread that can make calls to MPI routines for the `MPI_THREAD_FUNNELED` case, all MPI sends and receives have to be wrapped into the `omp master` directive. However the initialization code could be executed by any thread and hence is written as part of the `omp single` directive. If the data, and the initialization operations on the data, can be split equally among all the cores, each core could initialize its own data instead.

3.2 Aggregating messages and their effect of running times

For good speedup values, apart from ensuring that the time spent in computation performed by each thread is greater than the thread creation and destruction overhead, it is also important to make sure that there is little overhead in sending an aggregated message (from the work performed by all cores) as opposed to each core sending a message that corresponds to the work performed by itself. Sending one large message is usually more efficient than sending multiple small messages in parallel. Figure 2 compares the transfer time for messages sent in a pure MPI case versus transfer time for messages sent in the aggregated case. The measurements were run using MVAPICH1 with an 8-way MPI per node, with all 8 processes making message passing calls directly in one case, and in the other case all 8 threads aggregating their data to send a single message. It should be noted that the aggregate message size is 8 times that of the individual message size, at any point in the graph. The workload here involved simply sending messages of the desired size to a process on another node.

MPI distributions usually follow two different protocols for `MPI_Send()` depending on the size of the message to be transferred: the Eager send protocol and the Rendezvous send protocol [12]. The Eager send protocol has much less overhead as compared to the Rendezvous protocol. For our cluster setup, this switch from Eager to Rendezvous happens at a message size of 72KB. It can be seen that till the time the switch is made from the Eager protocol to the Rendezvous protocol, sending individual messages is much more efficient than sending aggregated messages. This is because in the window of message sizes from 9KB to 72KB, individual messages are being sent using the Eager send

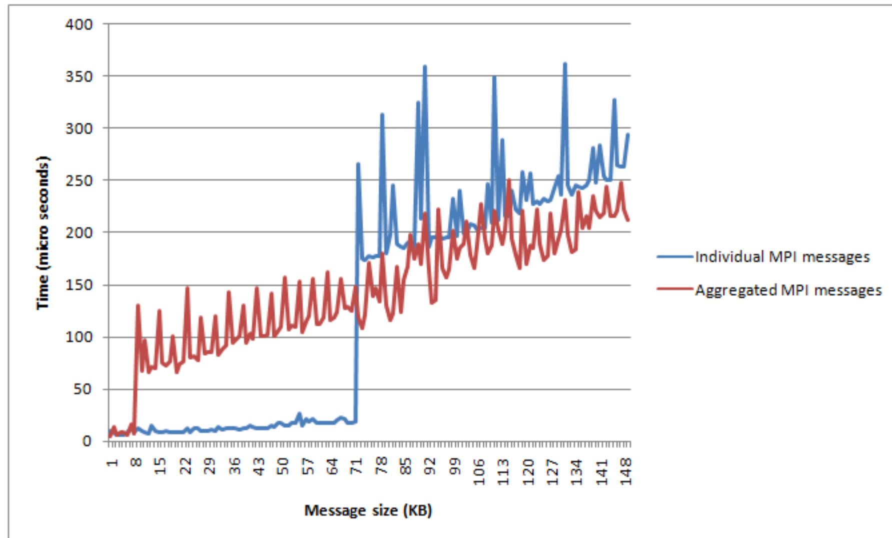

Fig. 2. Individual messages versus aggregated messages

protocol whereas the aggregate messages are sent using the Rendezvous protocol. Beyond 72KB, however, it is more efficient to send a single aggregated message instead of sending multiple per-thread messages.

The above comparison shows us one of the reasons why hybrid applications are likely to perform better than pure-MPI applications. In fact, it also tells us that making each OpenMP thread send MPI messages on its own (`MPI_THREAD_MULTIPLE`) is less likely to be useful as compared to only a single thread sending aggregated MPI messages (`MPI_THREAD_FUNNELED` or `MPI_THREAD_SERIALIZED`).

Another advantage of aggregating messages is in Strong scaling, i.e. when the problem size is fixed and the number of cores are increased. As we increase the core count, the local data size that each core operates on, goes on decreasing. Thus the messages exchanged also get smaller. However, the message transfer time does not decay till zero. It usually decreases till a specific level and then, due to increasing communication overhead due to the number of participating cores, it increases slightly. Now aggregated messages will always be larger than individual messages. Thus the point when the communication overhead contributes significantly to running time is reached earlier when sending individual messages as opposed to the case when sending aggregated messages. This is visible in the results presented later in this paper.

3.3 CPU affinity

CPU affinity allows us to specify which CPU each thread should run on. Setting the affinity mask is applicable to both pure-MPI as well as in hybrid MPI/OpenMP situations but it gains special importance in hybrid programs due to conflicting settings between the MPI distribution and the OpenMP runtime. MPI distributions often set the affinity mask for the MPI processes to ensure that the processes are confined to individual cores during scheduling.

When using hybrid MPI/OpenMP strategy, the OpenMP threads are created as part of the MPI process. If the affinity for the threads is not set explicitly, they all inherit the affinity mask of the process. This implies that by default, all OpenMP threads are made to run on the same core as the MPI process. Hence it becomes important to set the affinity mask explicitly when using hybrid strategy.

As of today, there are no direct means of specifying the CPU affinity for each thread in the OpenMP standard. Two methods are known, by which, the desired thread affinity can be specified viz., the `KMP_AFFINITY` [7] environment variable for the Intel C/C++ compiler and the `numa_run_on_node()` function in `libnuma` [8] (or the equivalent `numactl` command line utility). It should be noted that the `KMP_AFFINITY` variable is specific to the Intel compiler. This variable essentially permits specifying three things:

- whether threads should be tied down to individual cores or should they float among the available cores
- whether the default affinity mask (of the process) should be respected and
- should consecutive threads be scheduled on neighboring (or close-by) cores (and thus control cache effects)

Another means of setting the thread affinity is by use of the `numa_run_on_node()` function. As seen from the source code [8] of the containing library `libnuma`, this function uses the `sched_set_affinity()` system call to set the affinity mask. The function takes the number of the core that the thread is to be associated with.

3.4 Cache-line alignment and padding

Memory accesses tend to be hundreds of times slower than cache accesses. Thus a drastic performance improvement is obtained when frequently accessed memory items are made suitable to storage in the cache. This applies to the pure-MPI case also, especially when the MPI distribution uses shared memory to communicate between tasks on the same node. This not only implies aligning the start address of the data structures to be on a cache-line boundary but also ensuring that data structures are sufficiently padded so that they occupy complete cache lines. This avoids false-sharing (which is the case when two data structures occupying a single cache line are shared between two different CPUs).

4 Results

In this section, we present results of applying these optimizations to three scientific application kernels. The three kernels are selected from the list of Colella's "dwarfs" in [13], a list of kernels which capture the typical programming constructs in scientific applications. The dwarfs whose results we show here are the dense linear algebra problem, represented by a simple matrix multiplication, the structured grid problem in the form of a photonics simulations code performing a finite difference time domain algorithm in 3D (FDTD-3D) and the N-Body problem, implemented as a 3D gravitation problem. The Photonics simulations code performs nearest neighbor communication in a regular grid. It is important to note that apart from the optimizations mentioned above, the hybrid versions of the programs do not include any other significant changes, i.e. the hybrid programs use the same algorithms used by the pure MPI programs. Trends in

improvement of hybrid code over pure MPI code were similar for different data sizes of the same program and hence the plots have not been included in this paper for the sake of brevity.

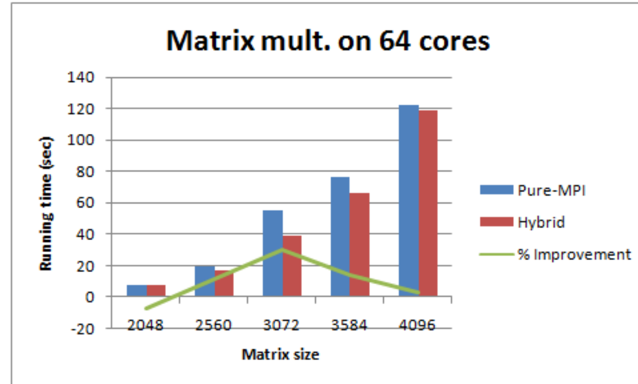

Fig. 3. Matrix multiplication on 64 processors

For the dense linear algebra program, we optimized the hybrid MPI/OpenMP version of Cannon's algorithm [14]. Due to its block-oriented nature, the algorithm is susceptible to available cache size. This is visible in the plot presented in Figure 3. For the matrix size of 3072x3072, the hybrid program performs about 30% better in comparison with the pure-MPI program. This illustrates the importance of choosing the correct block size. For block sizes larger than the available amount of cache, the matrices to be multiplied may be recursively broken down into pieces which ultimately fit into the cache, as described in [15].

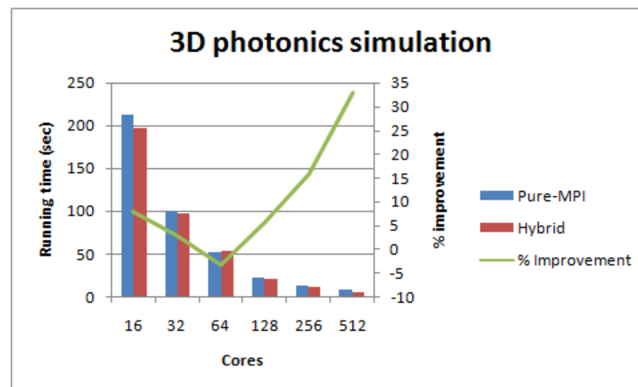

Fig. 4. 3D Finite difference time domain code operating over constant data size of 320x320x320 elements

The next application that we show here is the 3-dimensional Finite Difference Time Domain for a constant problem size of 320x320x320 elements. This

application is a more computation-intensive application and hence exhibits characteristics that are somewhat reverse of those found for the N-Body problem (which is a communication intensive application). Each processor is assigned a part of the cube and collaborates with its neighbors in the three dimensions. The optimization results for this application are shown in Figure 4. One can observe that the improvement curve suddenly begins to rise after falling till 64 cores. To understand the cause, we studied the time spent by the code in the computation and communication phases. This was done using the TAU performance profiling system [16]. It was observed that beyond 64 cores, the time spent in communication increased significantly in the pure-MPI case whereas it continued to decrease in the hybrid case. The hybrid program sends messages in an aggregated fashion and contributes lesser to message transfer overhead as opposed to the pure-MPI program, as described in section 3.2.

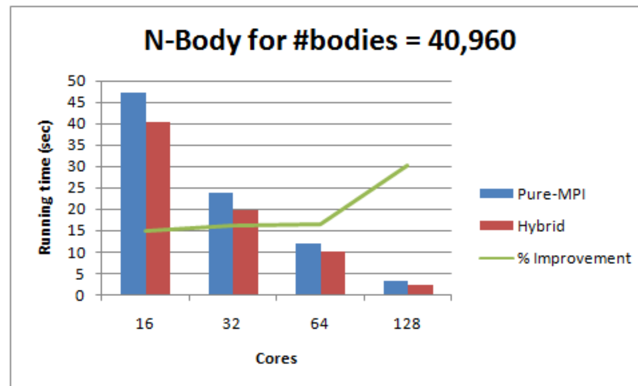

Fig. 5. N-Body program running time for number of bodies

The N-Body program is known to be a communication intensive application. Due to this, message latencies and overheads significantly affect the running time of the program. As shown in Figure 5, the code shows a constant improvement of about 16% till 64 cores, beyond which the hybrid code performs better in comparison with the pure-MPI code. This behavior is similar to that exhibited by the Photonics simulation code shown in Figure 4.

To illustrate the effect of the above mentioned optimizations, we took the 3D Photonics simulation code as a representative application and measured the effect of the optimizations on the running time. The percentage improvements of the hybrid code over the same code with the specific optimization removed, are shown in Figure 6. Here were are increasing the number of cores for a fixed problem size of 320x320x320 elements. Since the benefits of caching and message transfer time vary according to the problem size, they have been excluded. When the affinity of the threads is not set explicitly, all threads were being mapped to the same core and hence this had the maximum effect on the running time.

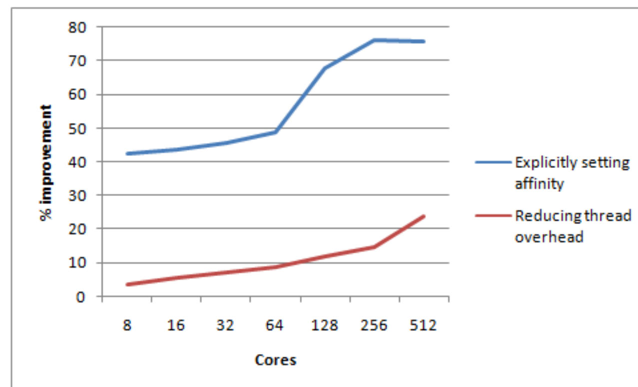

Fig. 6. Comparing the effect of optimization techniques

5 Conclusion and future work

The paper presented a series of optimization techniques that we found to be most useful when optimizing hybrid MPI/OpenMP applications for performance. It is important to realize that conversion of a pure-MPI code to an optimized hybrid MPI/OpenMP code may not be possible by inserting OpenMP `#pragma`s alone. One may even have to restructure the code to effectively leverage the inter-processor bandwidth via shared memory. The hybrid codes for n-body problem and for matrix multiplication were parallelized without using any `parallel for` constructs of OpenMP. This emphasizes the need to look at the underlying arrangement of cores, their interconnections, cache organization and the effect of messages on the running time of the program. As an extension of this work, we plan to develop criteria of issues that affect performance of an application. We plan to use these to build a tool that can suggest changes to be made to programs so that they would be tuned for optimal performance.

References

[1] F Cappello, D Etiemble, "MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks", Supercomputing ACM/IEEE 2000 Conference

[2] Jin, H. W., et. al, "LiMIC: support for high-performance MPI intra-node communication on Linux cluster", International Conference on Parallel Processing, 2005

[3] Hoeflinger, J.P.: Extending OpenMP to Clusters (2006)

[4] `https://computing.llnl.gov/tutorials/pthreads/`

[5] OpenMP Version 3.0 specifications, OpenMP Forum: `http://www.openmp.org/mp-documents/spec30.pdf`

[6] Xinmin Tian et. al., "Compiler and Runtime Support for Running OpenMP Programs on Pentium- and Itanium-Architectures", Proceedings of the 17th International Symposium on Parallel and Distributed Processing 2003, Page: 130.1

[7] Intel `KMP_AFFINITY` documentation: `http://www.intel.com/software/products/compilers/docs/fmac/doc_files/source/extfile/optaps_for/common/optaps_openmp_thread_affinity.htm`

[8] SGI `libnuma` project page: `http://oss.sgi.com/projects/libnuma/`

[9] `sched_setaffinity()` man page: `http://linux.die.net/man/2/sched_setaffinity`

[10] Ranger cluster at Texas Advanced Computing Center: http://www.tacc.utexas.edu/resources/hpcsystems/
[11] MPI_Init_thread() documentation: http://www.mpi-forum.org/docs/mpi-20-html/node165.htm
[12] http://www.mhpcc.edu/training/workshop2/mpi_performance/MAIN.html#Protocols
[13] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, K. Yelik, "The Landscape of Parallel Computing Research: A View from Berkeley" Technical report, EECS Department, University of California at Berkeley, December, 2006.
[14] L. E. Cannon, "A cellular computer to implement the Kalman Filter Algorithm", Ph.D. dissertation, Montana State University, 1969
[15] Michael J. Quinn, "Parallel Programming in C with MPI and OpenMP", McGraw Hill, ISBN 0-07-282256-2
[16] S. Shende and A. D. Malony, "The TAU Parallel Performance System", International Journal of High Performance Computing Applications, Volume 20 Number 2 Summer 2006. Pages 287-331