Optimizing the Advanced Accelerator Simulation Framework Synergia Using OpenMP

Hongzhang Shan[1], Erich Strohmaier[1], James Amundson[2], and Eric G. Stern[2]

[1] Future Technology Group
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720
hshan, estrohmaier@lbl.gov
[2] Fermi National Accelerator Laboratory
Batavia, IL60510
amundson, egstern@fnal.gov

Abstract. Synergia is an advanced accelerator framework widely used by accelerator community. However, its performance suffers significantly from the high communication requirement. In this paper, we address this issue by replacing the flat MPI programming model with the hybrid OpenMP+MPI programming model. We describe in detail how the code has been parallelized in OpenMP and what the challenges are. The improved hybrid code can perform over 1.7 times better than the original program for a benchmark problem.

1 Introduction

Synergia [1] is an open source framework developed for accelerator community to simulate beam dynamics with fully three dimensional space-charge capabilities and a higher order optics implementation. It can be used to predict the motion of high energy particles in a beam (bunched or continuous) in 6D phase spaces. The electric and maganetic fields are expressed on a 3D rectangular grid, and, at any given time, both longitudinal and transverse motions are treated consistently. It is designed to be run efficiently on parallel computers. The ultimate goal is to run the accelerator simulations on the leadership large-scale computing platforms. However, due to the tremendous difficulties in the optimization of tightly coupled 6D simulations, the current goal is able to run the codes on medium size clusters.

The most difficult challenge is the high communication requirement. Synergia uses Particle-In-Cell (PIC) [6] method to simulate the beam dynamics. The interactions between the particles and the fields cause a large amount of data to be commuted globally. Figure 1 shows the scaling behavior of the benchmark program used by synergia. With the increase of the number of MPI processes, the communication time (MPI time) increases steadily. When the number of processes reaches 256, over 70% of the running time has been spent for MPI communication and the total running time no longer goes lower. In this paper, we will discuss in detail why using OpenMP can improve the performance of our applications and how exactly we do it. In addition, we will also discuss the choices and problems we face during our optimizing process.

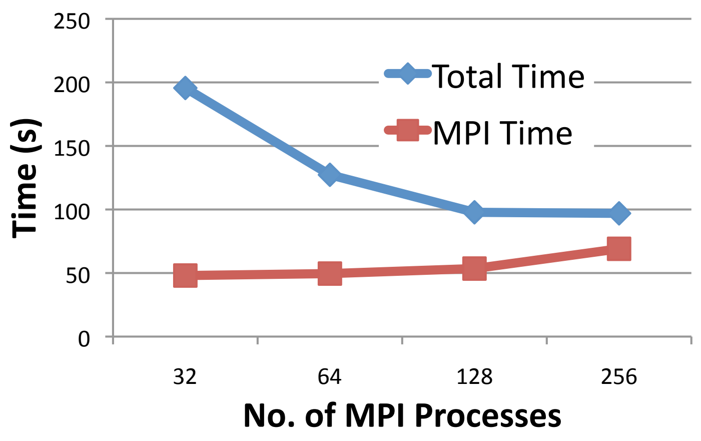

Fig. 1. The scalability of synergia on the Cray XE6.

Using hybrid MPI+OpenMP programming model also fits more naturally with the trend of the development of the computer architectures as multicore or manycore technology has started to dominate the current high-performance computing (HPC) platforms. MPI is used for internode communication while OpenMP is used for inside node computation and communication. Compared with flat MPI programming model, using OpenMP also helps to reduce the amount of memory needed by applications due to its global shared address space. This advantage will become increasingly important as the amount of memory per core will be reduced on future petascale or exascale platforms.

The rest of the paper is organized as follows. First, Section 2 describes the algorithms and communication patterns of the benchmark application for synergia. Section 3 describes the experimental platform. The detailed optimization process using OpenMP is discussed in Section 4. Related work is discussed in Section 5. Finally, we summarize our conclusions and future work in Section 6.

2 Benchmark Application

Synergia is a multi component, multi language framework. It provides a straightforward user interface through Python classes. The benchmark application is contained in the subdirectory cxx_test in the file cxx_example.cc. It is build as part of the normal synergia build process.

The code simulates the beam dynamics with full 3D space-charge effects using particle-in-cell algorithm [6]. The charged particles interacting via magnetic and electric fields and invokes many performance critical components. The field is

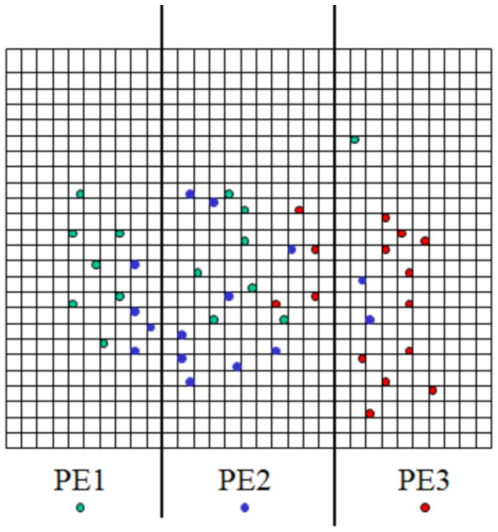

Fig. 2. A schematic plot of the particle-field decomposition method, source from [6].

modeled by a 3D computational grid and partitioned uniformly among all the processes. Similarly, the charged particles are also evenly partitioned among the processes. However, the particles belong to a process may scatter over the whole field. Figure 2 shows a schematic plot of the particle-field decomposition among three processors. Following is the list of major steps performed by the benchmark:

1. **Get Local Charge Density (RHO):** The local particles are deposited onto the computational grid to obtain the charge density distribution. Due to the global distribution of the particles, the whole comptational grid is needed. (local)
2. **Get Global Charge Density (RHO):** The local charge density is summed up by calling MPI_Allreduce. (global)
3. **Get PHI2:** Solving the poison equation using the FFT-based Green function method. The FFT is performed using FFTW [3] software package. The transpose communication is implemented using pairwise MPI_Sendrecv. (Local + Global)
4. **Get PHI:** Communicate with neighbors to get data for the boundary of the subdomain. (Global)
5. **Sort:** Sort the local particles based on position in Z direction. This is a periodic function to improve the data locality. (Local)
6. **Get Local En:** Prepare local field data for the following global communication. There are three field components needed to be prepared. (Local)
7. **Get Global En:** All processes gather the whole computational field data (electric potential) using MPI_Allgather. (Global)

8. **Apply Kick:** The electric fields are interpolated onto individual particles and the particles are advanced using the self-consistent electromagnetic field and the external maps. (Local)

There are three major communication operations. The first is the MPI_Allreduce operation in *Get Global Charge Density* phase to sum up all local charge density. The second is the MPI_Sendrecv operation in *Get PHI2* phase for the matrix transpose in the solution of the Poisson equation using the FFT-based method. The third is the MPI_Allgather operation in the process of gathering the electric potential from the subdomain of each processor after the solution of the Poisson equation. The total communication volume is proportional to the number of computational grid points. Furthermore, the communication volumes of the first and the third operations are also proportional to the number of MPI processes. Therefore, using OpenMP can potentially reduce the total communication volume and improve application performance.

3 Platforms

Our work has been performed on a Cray XE6 platform, called Hopper, which is located at NERSC and consists of 6,384 dual-socket nodes each with 32GB DDR3 1333-MHz memory. The peak Gflops rate is 8.4 Gflops/core and 201.6 Gflops/node. Each socket within a node contains an AMD "Magny-Cours" processor at 2.1 GHz with 12 cores. Each Magny-Cours package is itself a MCM (Multi-Chip Module) containing two hex-core dies connected via hyper-transport. (See Fig. 3.) Each die has its own memory controller that is connected to two 4-GB DIMMS. This means each node can effectively be viewed as having four chips and there are large potential performance penalties for crossing the NUMA domains. Each core has its won L1 and L2 caches, with 64KB and 512KB respectively. One 6-MB L3 cache shared between 6 cores on the Magny-Cours processor. Every pair of nodes is connected via hypertransport to a Cray Gemini network chip, which collectively form a 17x8x24 3-D torus.

The compilation of synergia is through an automatic build system based in GNU Autotools [8]. The compiling software packages we used include the GNU compiler gcc (SUSE Linux) 4.3.4, Python 2.6, and CMAKE version 2.8.2.

4 Improving the Performance Using OpenMP

In this section, we will focus on improving the benchmark performance using OpenMP. The computational grid size is set as 64, 64, and 256 in X, Y, and Z direction, respectively. There are 10 particles per cell and total about 10 Million particles. Total 256 time steps have been simulated.

4.1 Parallelizing the Loops

The first step is straightforward. Finding those loops which have no data dependence across iterations and using "omp parallel for" pragma to parallelize the

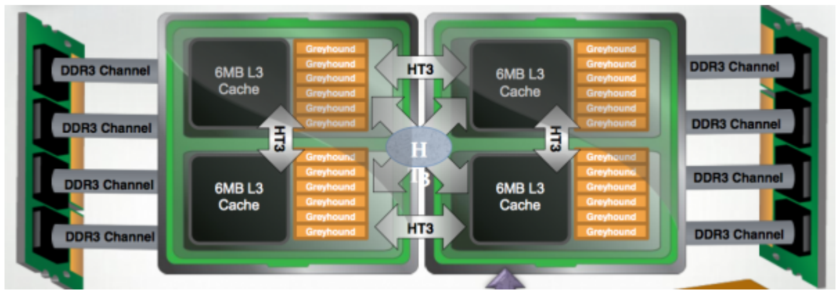

Fig. 3. The node architecture of Hopper.

work, including those loops to perform reductions at the end. One thing needs to pay spacial attention is the data placement. Since first touch policy is used on Hopper, we intentionally touch the data immediately after memory allocation so that the data and the OpenMP thread that will work on it will have the same core affinity. Otherwise, accessing data across the NUMA domains inside a node will cause a large performance penalty.

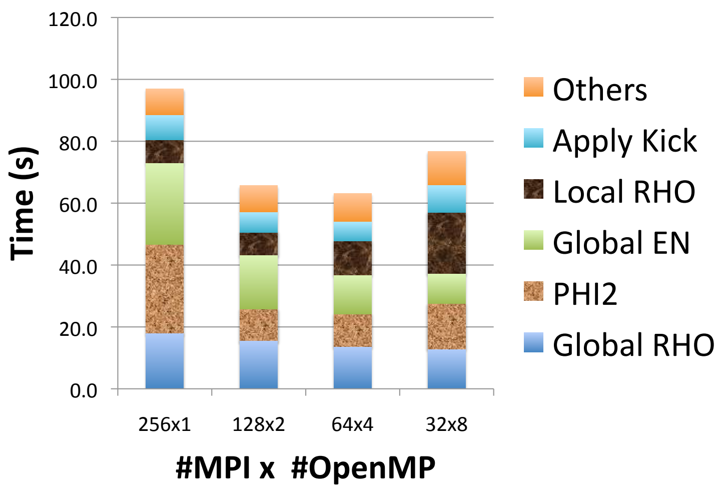

Fig. 4. The time breakdowns for different number of OpenMP threads .

Figure 4 displays the time breakdowns for different number of OpenMP threads per MPI process when total 256 cores are used. The top five time-consuming phases are shown (from bottom to up). They are for *Get Global Charge Density (Global RHO), Get PHI2 (PHI2), Get Global EN (Global EN), Get Local Charge Density (Local RHO), and Apply Kick (Apply Kick)*. The remaining time is counted as *Others*. The bottom three phases are dominated by communication and can be treated as communication time. The other three can be roughly teated as local computation time. For the flat MPI implementation (#OpenMP=1), the time spent on these three phases is around 73 seconds. It drops to 43 seconds when two OpenMP threads per MPI process are used and drops further to 37 seconds when four OpenMP threads per MPI process are used. For *Global RHO* and *Global EN*, the better performance is mainly due to the reduced amount of total communication volume as the number of MPI processes goes down. For *PHI2*, the communication is dominated by the matrix transpose needed by the FFT operation. The communication volume is constant. The transpose time drops significantly when switching from flat MPI to using two OpenMP threads per MPI process. This is probably because of the larger message sizes. However, further increasing #OpenMP does not improve the performance. Instead when #OpenMP=8, the time for *PHI2* goes up, causing the total communication time going up accordingly.

For *Local RHO*, it's responsible for depositing the local particles onto a local auxiliary grid and involves no MPI communication. The time goes down slightly when #OpenMP=2 and then goes up when higher number of OpenMP threads are used. Due to the data dependence across iterations, this phase can not be easily parallelized using OpenMP "prallel for" pragma. It is performed by only one OpenMP thread now. When the number of MPI processes goes down, the number of local particles belong to a MPI process becomes larger, leading to higher depositing time. The time does not go up when #OpenMP=2 is due to less memory contention. For *Apply Kick*, its time is mainly related with the number of particles assigned to each OpenMP thread. When the loop is perfectly parallelized with OpenMP, the number of particles per OpenMP thread remains constant. Therefore, the time should be constant. The small variation is caused by the memory performance.

Overall, the best performance is obtained when four OpenMP threads per MPI process are used. The total running time has been reduced over 50%. In the next two sections, we will investigate the performance of phase *PHI2* and phase *Local RHO*.

4.2 Using OpenMP for FFTW

FFTW [3] is used in synergia to perform the FFT to solve the Poisson equations in phase *Get PHI2*. The computational domain used for FFT is a doubled domain padded on the boundary. The actual size is 130, 120, and 512 for X, Y, and Z direction, respectively. By default, only MPI processes are involved. We changed the initialization process for FFTW3 and enabled OpenMP so that all OpenMP threads can participate in the FFT process. The results using one OpenMP thread and using more OpenMP threads are shown in Fig. 5. Using all OpenMP threads for FFTW helps the performance. But the improvement is slightly for 2 and 4 OpenMP thread cases and only become explicit when 8 OpenMP threads are used. In that case, more than 20% of FFTW time has been saved.

To understand whether the performance could be improved further, we isolate the code related with FFTW and develop an independent micro benchmark.

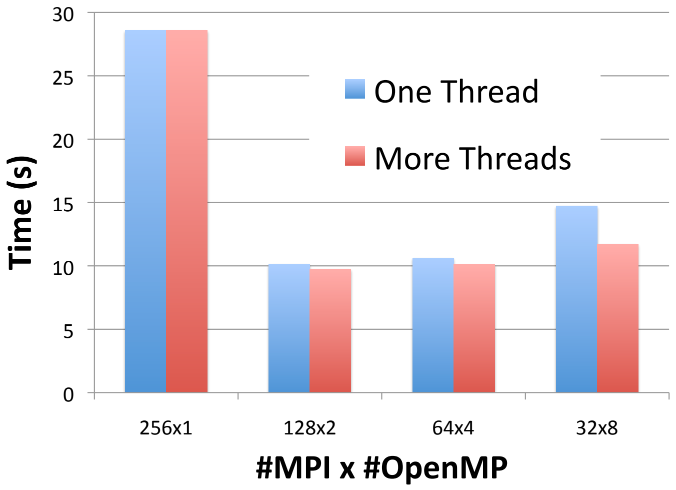

Fig. 5. The times for FFTW when one OpenMP thread and more threads are used.

We find that the micro benchmark results match those of synergia very well, indicating further improvement should be dependent on the progress of FFTW. The best result is obtained when #OpenMP=2.

4.3 Parallelizing Deposit

During the stage of computing local charge density, the particles are deposited onto a computational grid to obtain the charge density distribution. Due to the spatial distribution the particles, the grid size should cover the whole field instead of only the subdomain assigned to a process. As we mentioned earlier, due to the data dependence, this section can not be easily parallelized using "pragma omp parallel for".

Naive Approach The naive approach is to allocate an auxiliary grid for each OpenMP thread so that each thread can directly deposits its particles onto it. The particles assigned to a MPI process will be evenly partitioned among all the OpenMP threads spawned by the process. After the deposition, the charge density stored on the auxiliary grid will be reduced together by a sum operation. There are a lot of algorithms to perform the reduction operation at the end. In this study, we examined three implementations: Critical, Slicing, and BinaryTree.

- **Critical** Critical depends on "pragma omp critical" statement to perform the reduction. As long as an OpenMP thread finishes its particle deposition, it starts to compete for the critical section to add its particle contributions to the final field.

- **Slicing** In Slicing, each OpenMP thread is responsible for a fixed slice of the final field and fetches the data from all other OpenMP threads to perform the reduction.
- **BinaryTree** The reduction among the OpenMP threads will be carried out according to a binary tree structure from bottom to up. At the bottom, the reduction will be done in pairs. One thread of a pair will be responsible to perform the reduction. In the next step, the participated number of OpenMP threads will be reduced to half, only including those threads which performed the reduction operation in the last step. This process will be repeated $\lceil \log_2 n \rceil$ times (n is the number of OpenMP threads spawned by the same MPI process).

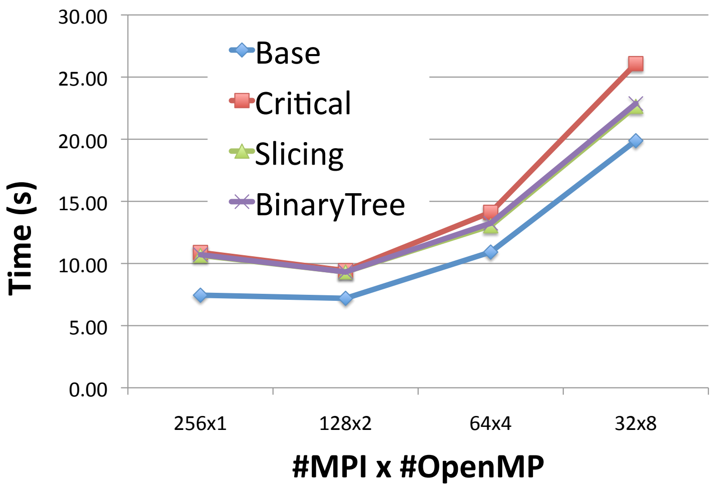

Fig. 6. The times for computing local charge density for different algorithms.

The timing results for phase *Get Local Charge Density* for different algorithms are shown in Fig. 6. The *Base* times are those measured in Section 4.1. None of the new algorithms performs better than the *Base*. The advantage of using more OpenMP threads is overshadowed by the overhead to access extra memory and perform the reduction operation. We also tried to use the reduction operation supported by OpenMP itself. However, we did not see better performance results either.

Lock Approach Another strategy is to use omp_locks instead of allocating extra amount of memory. The whole field domain will be partitioned along Z direction among all the OpenMP threads spawned by the same MPI process. Each OpenMP thread will be only responsible to compute the charge density for its assigned subdomain. All the particles assigned to a MPI process, no matter which OpenMP thread they are assigned to, as long as they fall into the same subdomain, will be deposited onto the field by the same thread which owns the subdomain. However, the particles will not only affect the charge density of its own position. They will also affect the charge density of its neighbor positions. Therefore, for boundary positions omp_locks are needed to assure result correctness. Different locks will be allocated for different positions to maximize concurrncy. The number of locks allocated is proportional to the number of OpenMP threads.

The remaining question is how each thread will find those particles for which it should be responsible. The thread can not afford going through all the particles to fulfill this purpose. One way is to allow the MPI process to sort the particles first based on their positions in Z direction and then we partition the particles among the OpenMP threads. However, the sorting turns out to be very expensive. Even worse, the time goes up when more OpenMP threads per MPI process are used. Therefore, it can only be done periodically to improve data locality.

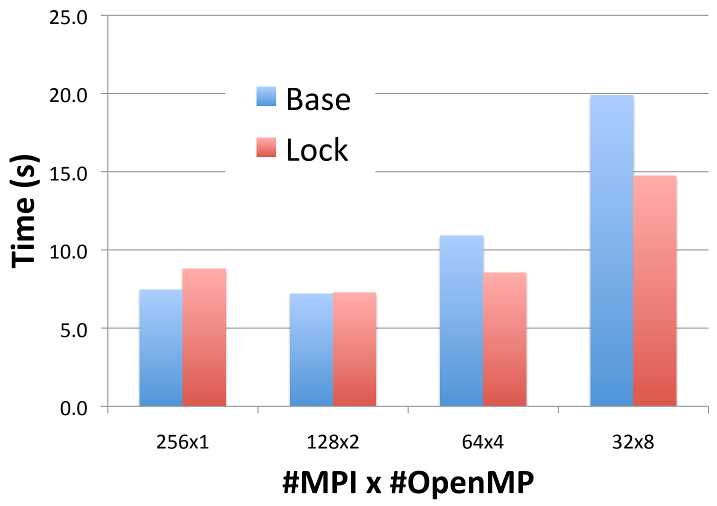

Fig. 7. The times for computing local charge density using omp_lock.

Instead, we partition the particles first among the OpenMP threads and allow each OpenMP thread to perform a sort on its own particles. We add a function called subsort in the synergia source code for this purpose.. The results is that all the particles assigned to a MPI process are now divided into n (n = #OpenMP) sorted sections. For each section, an OpenMP thread can use binary search to find the first particle it should work on and move left and right to get other particles as those particles it will work on should be continuous. This process will be repeated for every section.

Figure 7 shows the new results using mop_lock. When one OpenMP thread is used, the time becomes slightly higher due to the extra sorting work. When two threads are used, it is similar to the *Base* case. However, when four or eight threads are used, the performance becomes better. Over 30% of the time has been saved for eight-thread run.

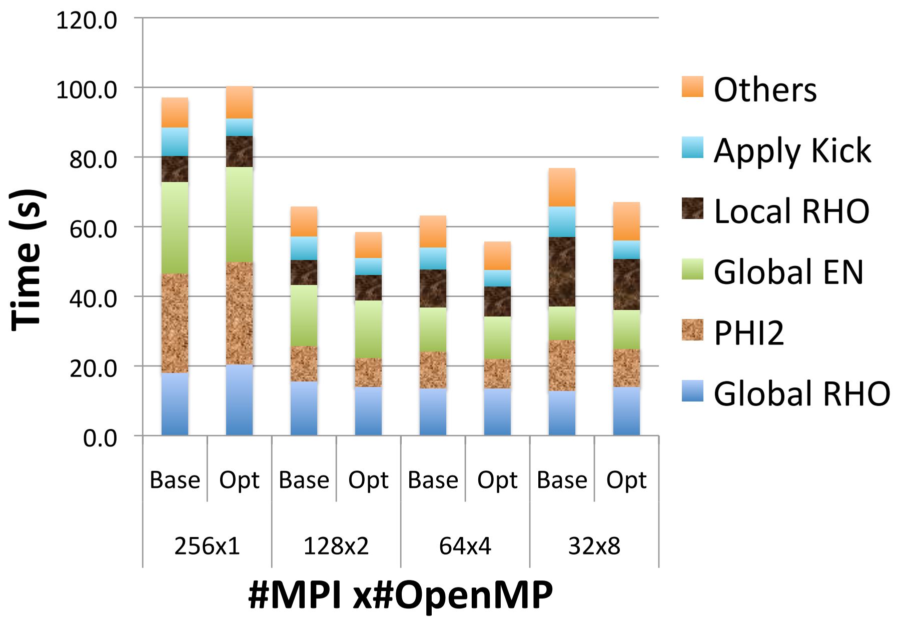

Fig. 8. The time breakdowns before and after optimization.

The final time breakdowns are shown in Fig. 8 when both optimizations for FFTW and charge deposition have been applied (labeled as OPT and comparing with the Base). The best performance is obtained when four OpenMP threads per MPI process are used. Compared with flat MPI results, the performance become more than 1.7 times better when 256 core are used. Using hybrid OpenMP+MPI programming model has significantly improve the performance. Another advantage of using OpenMP is the memory usage. Substantial amount of memory could be saved due to the shared address space supported by OpenMP. Table 1 shows the memory footprints when different number of OpenMP threads are used. As the growth in memory capacity is not keeping track with the growth in the number of cores on the future architectures, memory considerations are becoming much more important.

Table 1. The memory footprints for different MPI x OpenMP configurations (GB).

#MPI x #OpenMP	256x1	128x2	64x4	32x8
Memory (GB)	12.06	6.50	3.80	2.23

However, as we noted from Fig. 8, using eight OpenMP threads per MPI process will substantially increase the local computation time compared with cases using two or four OpenMP threads per MPI processes. The current architecture trend is to use more and more cores on a node. The number of cores will reach several hundreds or even a thousand in a few years. Using OpenMP to scale complex applications like synergia to a full node scale is extremely challenging. Some tools need to be developed to automatically optimize data placement and thread affinity. If not impossible, it will be very challenging for developers to perform such kind of task for complex applications as the number of OpenMP regions and related number of variables become very large. To improve the data locality for the NUMA architecture inside a node, some optimization techniques developed in the last decade for MPI may need to be applied to OpenMP also.

5 Related Work

Using OpenMP or hybrid MPI+OpenMP to improve the performance has been studied by many researchers. To name a few, Nakajima [5] described how to use a three-level hybrid programing model (vectorization, OpenMP, and MPI) to program efficiently on Earth Simulator. Shan et al. [7] discussed the advantage of using hybrid MPI+OpenMP programming model for NAS parallel applications. Kaushik et al. [4] investigated the performance of implicit PDF simulations for hybrid MPI+OpenMP programming model on a multicore architecture. Brunst and Mohr [2] introduced a tool to analyze the performance for hybrid OpenMP and MPI programs. The main difference from our work is that we focus on specific application synergia and on a new architecture, Cray XE6.

6 Summary and Conclusions

In this paper, we describe in detail how to use OpenMP to improve the performance for synergia. Using two or four OpenMP threads per MPI process, the performance could be improved significantly. In the best case, the performance has become over 1.7 times better when 256 cores are used. However, using more OpenMP threads per MPI process can not improve the performance further. Instead, the improvement starts to become less as memory contention becomes more severe. To address this challenge, we are currently working on a tool that can optimize the data placement and dynamically schedule the OpenMP threads inside a node to improve data locality. We are also planning on changing the workload partition method for synergia. Currently, it only partitions the grid along Z direction, which limits the scalability of the code.

References

1. J. Amundson, P. Spentzouris, J.Qiang, and R. Ryne. Synergia: An accelerator modeling tool with 3-d space charge. In *J. Comp. Phys. vol. 211, 229*, 2006.
2. H. Brunst and B. Mohr. Performance analysis of large-scale openmp and hybrid mpi/openmp applications with vampirng. In *IWOMP'05/IWOMP'06 Proceedings of the 2005 and 2006 international conference on OpenMP shared memory parallel programming*, 2005.
3. M. Frigo and S. G. Johnsoni. The design and implementation of fftw3. In *Proceedings of the IEEE 93 (2), 216231*, 2005.
4. D. Kaushik, D. Keyes, S. Balay, and B. Smith. Hybrid programming model for implicit pde simulations on multicore architectures. In *IWOMP'11 Proceedings of the 7th international conference on OpenMP in the Petascale era*, 2011.
5. K. Nakajima. Three-level hybrid vs. flat MPI on the Earth Simulator: parallel iterative solvers for finite-element method. In *Applied Numerical Mathematics, Volume 54 Issue 2*, July 2005.
6. J. Qiang and X. Li. Particle-field decomposition and domain decomposition in parallel particle-in-cell beam dynamics simulation. In *Computer Physics Communications, 181, 2024*, 2010.
7. H. Shan, F. Blagojevic, S. J. Min, P. Hargrove, H. Jin, K. Fuerlinger, A. Koniges, and N. J. Wright. A programming model performance study using the nas parallel benchmarks. In *Scientific Programming-Exploring Languages for Expressing Medium to Massive On-Chip Parallelism, Vol. 18, Issue 3-4*, August 2010.
8. G. Vaughan, B. Elliston, T. Tromey, and I. Taylor. GNU autoconf, automake and libtool. In *Pearson Education*, 2000.