Scalable MiniMD Design with Hybrid MPI and OpenSHMEM *

Mingzhe Li
The Ohio State University
limin@cse.ohio-state.edu

Jian Lin
The Ohio State University
linjia@cse.ohio-state.edu

Xiaoyi Lu
The Ohio State University
luxi@cse.ohio-state.edu

Khaled Hamidouche
The Ohio State University
hamidouc@cse.ohio-state.edu

Karen Tomko
Ohio Supercomputer Center
ktomko@osc.edu

Dhabaleswar K. (DK) Panda
The Ohio State University
panda@cse.ohio-state.edu

ABSTRACT

The MPI programming model has been widely used for scientific applications. The emergence of Partitioned Global Address Space (PGAS) programming models presents an alternative approach to improve programmability. With the global data view and light-weight communication operations, PGAS has the potential to increase the performance of scientific applications at scale. However, since the PGAS models are emerging, it is unlikely that entire applications will be re-written with them. Instead, unified communication runtimes have paved the way for a new class of hybrid applications that can leverage the benefits of both MPI and PGAS models. In this paper, we re-design an existing MPI based scientific mini-application (MiniMD) with MPI and OpenSHMEM programming models. We propose two alternative designs using MPI and OpenSHMEM programming models and compare performance and scalability of those designs with the original MPI-based implementation. Our performance evaluations using MVAPICH2-X (Unified MPI+PGAS Communication Runtime over InfiniBand) show a 17% reduction in total execution time, compared to existing MPI-based design with 1,024 cores.

1. INTRODUCTION

The Message Passing Interface (MPI) has been the de-facto standard programming model for High Performance Computing (HPC) applications. MPI is successfully used to implement regular, iterative parallel algorithms with well-defined communication behaviors. Data-driven applications often pose challenges associated with load balancing and often exhibit irregular communication patterns. These issues are hard to address with a traditional message-passing programming paradigm.

9*This research is supported in part by National Science Foundation grants #CCF-1213084, #CNS-1347189, #CNS-1419123 and #IIS-1447804. It used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

The Partitioned Global Address Space (PGAS) programming models present an alternative approach compared to message passing and are believed to improve programmability of such applications [2]. However, the PGAS models are still emerging and being standardized, whereas MPI is much more widely adopted. Owing to these reasons, it is unlikely that applications will be re-written solely with the PGAS models in the future. It is more likely that applications will continue to be written with MPI as their primary programming model; but, parts of the applications will be adapted to use a PGAS model, such as Unified Parallel C (UPC) or OpenSHMEM, leading to a class of "hybrid" applications. Indeed, the Exascale roadmap identifies the hybrid model as the 'practical' way of programming exascale systems [2]. Further, this trend has paved the way for unified communication runtimes, such as, MVAPICH2-X [5], that allow applications to leverage the best of both MPI and PGAS models.

Some studies have shown the benefits of porting existing MPI applications to pure OpenSHMEM or Hybrid MPI and OpenSHMEM based design. Pophale et al. [7] presented a port of the MPI based NAS Parallel Benchmarks to a pure OpenSHMEM based version and demonstrates good performance improvements. Jose et al. [4] proposed a hybrid MPI+OpenSHMEM based Graph500. Dinan et al. [1] proposed different hybrid program execution models. Studies [9] and [8] discussed converting MPI applications to PGAS models, but focus only on changing the communication to one-sided semantics. Numrich et al. [6] presented a performance model for the computation and communication in the MPI reference implementation of MiniMD.

In this paper, we first discuss the existing MPI MiniMD implementation in detail. Then we analyze the dominant communication functions. Based on our analysis, we re-design the existing MiniMD application with hybrid MPI and OpenSHMEM programming models. Performance evaluations using MVAPICH2-X [5] show a reduction in total execution time by up to 17%, compared to the existing MPI design at 1,024 cores. Our scalability analysis reveals that our hybrid MPI plus OpenSHMEM based design demonstrates good scaling (both weak and strong). To summarize, the following contributions are made in this paper:

1. Identify dominant MPI communication routines in existing

MPI based MiniMD implementation

2. Propose two alternative hybrid MPI and OpenSHMEM based designs

3. Present in-depth performance evaluation among different designs

2. OVERVIEW OF MINIMD AND COMMUNICATION CHARACTERISTICS

MiniMD is a Molecular Dynamics (MD) mini-application in the Mantevo [3] mini-application suite. It implements the Lennard–Jones interaction calculations and corresponding communication as in the LAMMPS software. MiniMD computes atom movement over 3D space and uses a spatially decomposed parallel MD method, where each process owns a subset of the simulation box. The primary work loop performs the following steps:

1. Depending on the atom locations, migrate the atoms to different ranks in every 20th iteration
2. Neighboring ranks exchange position information of atoms in boundary regions
3. Compute forces based on both local atoms and those in boundary regions from neighboring ranks
4. Neighboring ranks exchange force information of atoms in boundary regions
5. Update velocities and positions of local atoms

MiniMD has a stencil communication pattern which employs point-to-point message passing with irregular data during steps 2 and 4. A more complete description of the algorithm can be found in [6]. For this study, we focus on porting the MPI routines used in steps 2 and 4 to OpenSHMEM one-sided semantics.

In order to re-design MiniMD with MPI and OpenSHMEM, we need to find out which are the dominant MPI routines used in MiniMD. Table 1 lists all the MPI routines used in the reference MPI MiniMD and their purposes. Our profiling results show that the MPI point-to-point routines take up to 40%–90% out of the total communication time for input sizes ranging from 64*64*64 to 128*256*256. The MPI collective routines for data distribution at the beginning of the program and data collection at the end of the program take a very small portion of total communication time. Similarly, the MPI cartesian topology setup routines do not take much time either. Based on our analysis, our design focuses on porting point-to-point message passing semantics to OpenSHMEM one-sided semantics. With this approach, our proposed design can take advantage of the MPI topology routines that are not available for OpenSHMEM yet and also use the light-weight OpenSHMEM one-sided semantics.

3. PROPOSED DESIGNS

In the following two subsections, we introduce two design alternatives to re-design the point-to-point communication in the existing MPI-based version. We keep the same molecular dynamics (MD) algorithm in both proposed designs. Thus, the communication pattern and volume of data transferred in our proposed hybrid versions are the same as in the existing MPI-based version. Also, MPI rank and OpenSHMEM rank of the hybrid program are kept the same.

Table 1: Major MPI Routines Used in Existing MiniMD

MPI Routine	Purpose
MPI_Cart_create MPI_Cart_get MPI_Cart_shift	Topology setup for spatial decomposition
MPI_Send MPI_Irecv MPI_Wait	Transfer atoms between neighbor processes and exchange position and force information of atoms
MPI_Allreduce MPI_Bcast	Setup input data and collect information to generate output report

3.1 Hybrid-Barrier Design

To port message passing semantics to one-sided semantics, we need to take care of both communication and synchronization. Since all communication patterns are kept the same as the original version, our Hybrid-Barrier design aims to replace MPI_Send/MPI_Irecv with corresponding OpenSHMEM one-sided functions. In our case, we replace MPI_Send with shmem_putmem in the origin process. In order to ensure completeness and synchronization between the origin and target process, we add a shmem_barrier before and after each shmem_putmem. Our Hybrid-Barrier design is similar to the one proposed in [7]. We keep the same send buffer and receive buffer used in the original MPI based design. And, we allocate this receive buffer in OpenSHMEM heap memory. In this way, the origin processes can directly access this memory region without interrupting the target process.

3.2 Hybrid-Advanced Design

From Section 3.1, we see that directly porting MPI semantics to corresponding OpenSHMEM semantics will add some overhead as a result of the global barrier. In order to re-design a high performance and scalable MiniMD, there are several things to be taken care of: What is the size of the receive buffer and how can it be managed? How can the origin process know the target address? How to ensure coordination between multiple sender processes and coordination between the origin and the target processes?

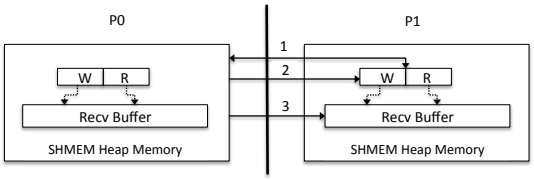

Figure 1: Hybrid-Advanced Design Overview

Figure 1 shows overview of our Hybrid-Advanced design. To start, we replace MPI_Send in the sender process with shmem_putmem as in our first design. In the initialization phase, every process uses shmalloc to allocate a heap memory region as receive buffer. Each process maintains two indices for this buffer. One is the read index and the other is the write index, both of them initially point to the beginning of the process's local receive buffer. Both of them will be reset when they reach the end of the receive buffer. Write index marks the point where the origin process can start to write data. Read index marks the point where the local process starts to read from the receive buffer. When one process needs to write data to a target process, it issues atomic shmem_longlong_fadd to fetch both the read and write indices. When the origin process gets the target process write index and read index, it needs to check whether there is enough space for new data. Based on the return values, the

origin process proceeds with one of the three options: 1) If the new data can fit into the target process receive buffer, it issues atomic shm_longlong_fadd (data_size) to the target process to update this write index and get a target address where it will write the data. After this atomic operation, the origin process can start to write data into target process memory. 2) If the write index reaches the end of the receive buffer, the origin process needs to reset this write index by issuing atomic shmem_longlong_cswap. When this resetting is finished, the origin process can follow the same step as step 1 to write data to the target process. 3) If the write index does not reach the end of the receive buffer and read index minus write index is smaller than data size, it means that there is not enough space for the new data in the receive buffer at this point. The origin process has to wait until more receive buffer space is available before sending. Regarding the target process, it is not actively involved in the data transfer. So it only polls on local buffer to wait for incoming data. With this circular buffer management design, we can avoid a high memory footprint. An alternative buffer management scheme is double buffering. However, it will increase memory footprint compared to our proposed circular buffer design. So we did not pursue that design. Furthermore, we can avoid the global barrier bottleneck.

4. PERFORMANCE EVALUATION

4.1 Experimental Setup

Cluster A: This cluster (TACC Stampede [10]) is equipped with compute nodes with Intel Sandybridge series of processors using Xeon dual eight-core sockets, operating at 2.70 GHz with 32 GB RAM. Each node is equipped with MT4099 FDR ConnectX HCAs (56 Gbps data rate) with PCI-Ex Gen3 interfaces. The operating system used is CentOS release 6.3.

Cluster B: This cluster consists of 144 compute nodes with Intel Westmere series of processors using Xeon Dual quad-core processor nodes operating at 2.67 GHz with 12 GB RAM. Each node is equipped with MT26428 QDR ConnectX HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces. The operating system used is Red Hat Enterprise Linux Server release 6.3 (Santiago), with kernel version 2.6.32-71.el6 and OpenFabrics version 1.5.3-3.

We used the MPI reference implementation from MiniMD v1.0 as our base code for these experiments and MVAPICH2-X 2.0 stack.

4.2 Performance

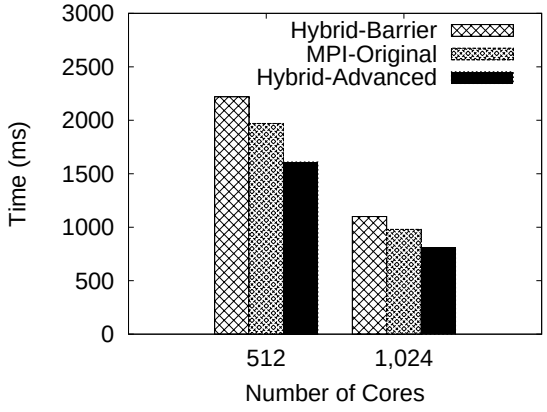

Figure 2: Performance Results

Figure 2 presents the performance results of those two proposed designs and compare them with the existing MPI-based version. These experiments were executed on Cluster B with 512 and 1,024 cores. In our experiments, we run one MPI process per core. From the figure, we can see that the Hybrid-Advanced version outperforms each of the other two versions. With 1,024 cores, the time taken by the existing MPI version was 0.98 seconds, whereas our proposed Hybrid-Advanced version took just 0.81 seconds. This is about a 17% reduction in execution time. The time taken by the Hybrid-Barrier version was 1.10 seconds, which is about 12% worse than the existing MPI based version. The performance degradation in this Hybrid-Barrier version comes from the global barrier used to ensure completeness of each communication operation and synchronization between processes. There are three reasons for we obtain performance benefits in our proposed design: 1) Reducing the communication and synchronization time by using lighter-weight one-sided routines vs. MPI two-sided. 2) Avoiding the busy polling of MPI_Wait used in existing MPI based version by resorting to light-weight local polling for the target process. 3) Achieving a better computation/communication overlap due to the one-sided semantics. Figure 3 shows the execution time breakdowns in computation and communication for two runs with 512 and 1,024 cores, respectively. We can see that our proposed Hybrid-Advanced version helps reduce the communication time compared with existing MPI based version.

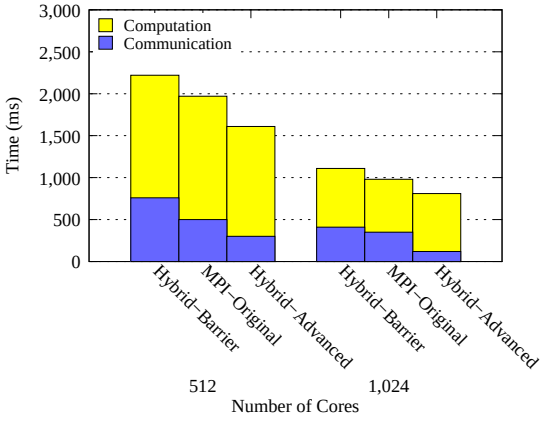

Figure 3: Execution Time Breakdown

4.3 Scalability

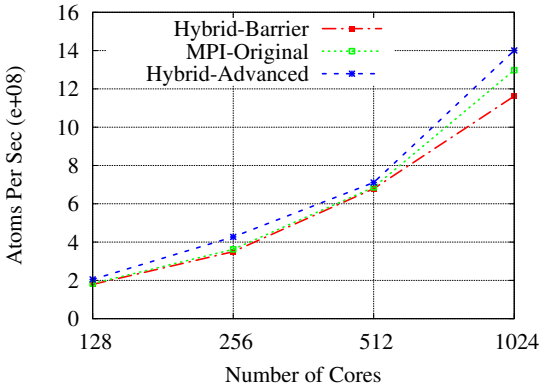

Figure 4: Strong Scaling (Atoms Per Sec)

In this section, we present strong and weak scalability evaluation

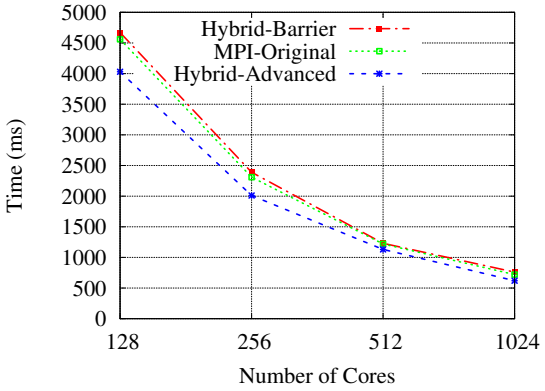

Figure 5: Strong Scaling (Time)

results. These experiments were executed on Cluster A. We report the performance (number of atoms transferred per second) and total execution Time. Figure 4 and Figure 5 depict the strong scaling results. In these experiments, we kept a constant problem size (128 * 128 * 128) and varied the scale of the system from 128 cores to 1,024 cores. From the Graph, we can see that the existing MPI version and our proposed versions scale well from 256 cores to 1,024 cores. The performance results indicate that the Hybrid-Advanced design exhibits good strong scalability. For our Hybrid-Barrier design which uses shmem_barrier there is a large degradation at 1,024 cores. This is expected due to the global barrier.

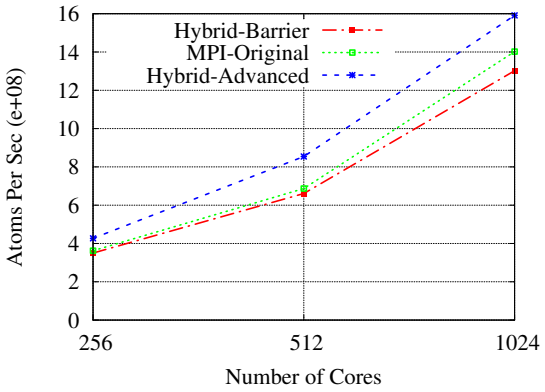

Figure 6: Weak Scaling (Atoms Per Sec)

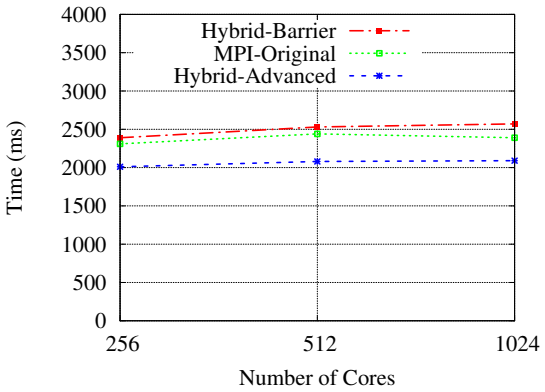

Figure 7: Weak Scaling (Time)

Figure 6 and Figure 7 show the Weak scaling results. In these experiments, we increased the number of processes used and the input size at the same rate. The input size per core was kept constant based on a 128 * 128 * 128 grid for 256 cores. We can see that the Hybrid-Advanced version also performs better in the weak scaling tests than all other implementations.

5. CONCLUSION AND FUTURE WORK

In this paper, we have presented an analysis of existing MPI based MiniMD and identified its dominant communication routines. Based on these observations, we proposed two alternative designs of MiniMD using MPI and OpenSHMEM models. Performance evaluations using MVAPICH2-X show a 17% reduction in total execution time compared to the existing MPI based design at 1,024 cores. Scalability analysis shows that the proposed Hybrid-Advanced design can achieve both good strong and weak scalability. In the future, we plan to continue working along these directions. We are interested in evaluating our design at a larger scale with different hardware architectures. Last but not least, we would like to redesign other real world MPI applications using MPI and OpenSHMEM and demonstrate the benefits.

References

[1] J. Dinan, D. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable Work Stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[2] Dongarra, Jack and Beckman, Pete and Moore, Terry and Aerts, Patrick et al. The International Exascale Software Project Roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, Feb. 2011.

[3] M. A. Heroux, D. W. Dorfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-Applications. Technical Report SAND2009-5574, 2009.

[4] J. Jose, K. T. Sreeram Potluri, and D. K. Panda. Designing Scalable Graph500 Benchmark with Hybrid MPI+OpenSHMEM Programming Models. In *International Supercomputing Conference (ISC)*, 2013.

[5] MVAPICH2-X: Unified MPI+PGAS Communication Runtime over OpenFabrics/Gen2 for Exascale Systems. http://mvapich.cse.ohio-state.edu/.

[6] R. W. Numrich and M. A. Heroux. A Performance Model with a Fixed Point for a Molecular Dynamics Kernel. *Computer Science - Research and Development*, 23(3-4):195–201, 2009.

[7] S. Pophale, H. Jin, S. Poole, and J. Kuehn. OpenSHMEM Performance and Potential: A NPB Experimental Study. In *Proceedings of the 1st Conference on OpenSHMEM Workshop*, Oct 2013.

[8] R. Preissl, J. Shalf, N. Wichmann, B. Long, and S. Ethier. Advanced Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms. In *Conference on Partitioned Global Address Space Programming Models (PGAS)*, 2011.

[9] H. Shan, F. Blagojević, 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. *Sci. Program.*, 18(3-4):153–167, Aug. 2010.

[10] TACC Stampede Cluster. http://www.xsede.org/resources/overview.