Performance Analysis of Lattice QCD Application with APGAS Programming Model

Koichi Shirahata
Tokyo Institute of Technology
koichi-s@matsulab.is.titech.ac.jp

Jun Doi Mikio Takeuchi
IBM Research - Tokyo
doichan@jp.ibm.com mtake@jp.ibm.com

Abstract

It is expected that the first exascale supercomputer will be deployed within the next 10 years, but the programming model which allows easy development and high performance is still unknown. APGAS programming model offers a flexible way for wide range of applications to express many patterns of concurrency, communication, and control through the combination of asynchronous operations and a global view of data. However, the comparative performance of APGAS model with existing standard message passing models such as MPI remains unclear. In this work, we give a detailed comparative analysis of APGAS model in X10 with the standard message passing model, by using lattice Quantum Chromodynamics (QCD) as an example, which is one of the most challenging applications for supercomputers. We further analyze the performance of lattice QCD in X10 and apply several optimizations. Our experimental results show that our X10 implementation scales up to 256 places. The results also show that the MPI-based implementation performs 2.26x–2.58x faster than the X10 implementation.

Keywords Exascale Computing, Programming Models, Performance Analysis

1. Introduction

It is expected that the first exascale supercomputer will be deployed within the next 10 years. Partitioned Global Address Space (PGAS) model [4] is a programming model which simplifies parallel programming while exposing data/thread locality to enhance performance. PGAS model provides a global view of distributed memory to programmers. Asynchronous PGAS (APGAS) programming model [5] is a programming model which provides flexible and efficient parallel processing with simple instructions. APGAS model offers a flexible way for wide range of applications to express many patterns of concurrency, communication, and control through the combination of asynchronous operations and a global view of data.

However, the comparative performance of APGAS model with existing standard message passing models such as MPI [9] remains unclear. Although APGAS model provides a good scalability with high productivity, there is a tradeoff between performance and productivity. When compared with standard message passing models, APGAS model has higher programming productivity while less tuning flexibility.

In this work, we give a detailed comparative analysis of APGAS model in X10 and MPI based on lattice Quantum Chromodynamics (QCD) [2], which is an application from theoretical high-energy physics, on parallel computing platforms. Lattice QCD application is one of the most challenging application for supercomputers, since it requires high memory bandwidth, high network bandwidth, and high computational power. We implement lattice QCD in X10 by porting an existing lattice QCD implementation in C++ [3] then parallelize using APGAS programming model. We analyze parallel efficiency of X10 and compare the performance of our lattice QCD implementation in X10 with lattice QCD in MPI. We further analyze the performance of lattice QCD in X10 and apply several optimizations. The results show that our X10 implementation scales up to 256 places. The results also show that MPI implementation performs 2.26x–2.58x faster than the X10 implementation.

Here is a quick summary of contributions of our work:

- We implement lattice QCD application in X10 with several optimizations.
- We give a detailed performance analysis on lattice QCD in X10, including analysis on multi-activities and distributed implementations.
- We give a comparative performance analysis between X10 and MPI. We show a limitation of tuning flexibility in current X10 implementation for overlapping communication.

2. Background

In this section, we explain APGAS programming model with X10 programming language. Then we introduce lattice QCD application, which is used as a benchmark application for large-scale computing environments.

2.1 APGAS Programming Model and X10

Partitioned Global Address Space (PGAS) model [4] is a programming model which virtualizes distributed memory as a global address space where a object can be placed over multiple locations on the distributed memory. There are several PGAS programming languages such as Co-Array Fortran [12] and Unified Parallel C [13].

APGAS (Asynchronous PGAS) programming model [5] is a PGAS model which enables dynamic task creation under programmer control. APGAS programming model mainly consists of two parts: *places* and *activities*. A *place* is simply coherent portion of the address space; a collection of data together with the activities that operate on that data. Places have a important property that they are not required to be single-threaded. That is, multiple activities may be active simultaneously in a single place. An *async* is the denotational mechanism to express activities that perform computation in a place. An *async* is launched at a given place and stays at that place for its lifetime.

X10 [6–8] is a language which implements APGAS programming model. In X10, PGAS memories are called places, where each place is allocated on a process. Programmer controls places by moving to other place using `at` statement. A new activity, which is allocated on the same place, is created dynamically by using `async` statement. There are language specific limitations on how asyncs reference remote data. In X10, the async cannot access locations at remote places. If it desires to operate on remote locations it has access to, it must launch a new async at that place. Asyncs may be used not just to run computations at a remote place but also to specify data-transfers such as array copies from an array at a place p to an array at a place q. `asyncCopy` conducts a place-to-place asynchronous data transfer. Activities in a place can be spawned locally or remotely. To control their execution, `finish` statement is introduced; a synchronization construct that allows a parent computation to wait for the completion of all its children activities. `finish` captures the very powerful notion of distributed termination detection.

2.2 Lattice QCD Application

Lattice QCD [2] is a common technique to simulate a field theory of quantum chromodynamics (QCD) theory of quarks and gluons on 4D lattice consisting of 3D space and 1D time. The quark fields are placed on the sites of 4D lattice and the gauge fields are placed as the links of the lattice sites to represent the effect of the gluons as the transporters of the quark fields. The simulation of the lattice QCD uses a finite difference method to solve the interactions.

Lattice QCD computation mainly consists of Monte-Carlo simulations on 4D lattice. The computation is dominated by solving a system of linear equations of matrix-vector multiplication using iterative methods, such as conjugate gradient (CG) method. The most computation and communication intensive procedure in lattice QCD is solving the following equation for Dirac matrix:

$$M(U)x = b \qquad (1)$$

where M is the discretized Dirac operator which is a sparse matrix whose elements are a function of a background field U, and b and x are the source and solution vectors respectively. Wilson-Dirac operator is used to calculate the physical exchange between 4D lattice sites through the effects of the gluon fields , by multiplying spinor and gauge matrix on 8 neighbors of x, y, z, t dimensions with positive or negative signs. This problem accounts for the majority of operations in lattice QCD.

3. Implementation of Lattice QCD Application in X10

In this section, we explain our implementation of lattice QCD application in X10. We have fully ported an existing open source lattice QCD implementation [3] from C++ [1] to X10. We use a sequential version of lattice QCD in C++ as the baseline implementation. Our implementation is in an object-oriented style, which is composed of objects such as lattice, complex field, communicator, etc. We use `Rail` class, which represents 1D array in X10, for storing 4D arrays of quarks and gluons.

3.1 Parallelization of Lattice QCD in X10

The lattice QCD computation can be parallelized by dividing 4D lattice into partial lattice in each direction and mapping them to each place. When the computation is parallelized, each iteration consists of boundary exchange and bulk computation. The boundary exchanges are required between places after the bulk computation in each direction for each iteration. The boundary exchanges are conducted only between neighbor places in each direction in order to update boundary data for subsequent computation. Computations in each dimension are independent, which means computations in each direction can be shuffled and overlapped among places.

We parallelize the implementation for distributed memory environments. In order to partition 4D lattice into multiple places, the program calculates memory offsets on each place at the initialization time. Boundary exchanges are operated using asynchronous copy functions. We apply several optimizations to our lattice QCD

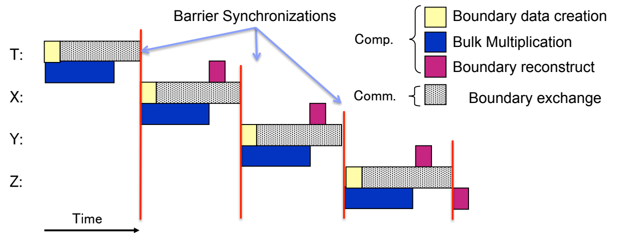

Figure 1. Implementation of overlapping on lattice QCD application in X10. Overlapping computations and communication in the 4 directions.

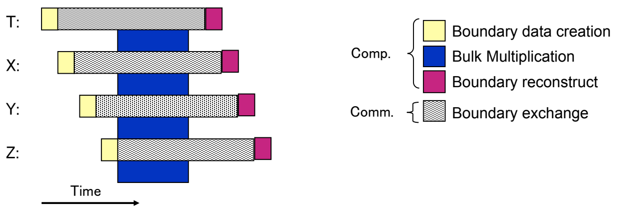

Figure 2. Implementation of overlapping on lattice QCD application in MPI. Overlapping computations and communication by using a pipeline for the independent procedures in the 4 directions.

implementation in X10, including communication optimizations and hybrid parallelization.

3.2 Communication Optimizations of lattice QCD in X10

We apply an overlapping technique between boundary exchanges and bulk computations (Figure 1). Communications are overlapped by using `asyncCopy` method of `Rail` class. `asyncCopy` method creates a new activity and transfers data asynchronously on the activity. While the boundary data is being transferred from the activity, the main activity continues bulk computation. After calling the bulk computation, the main activity waits the completion of the boundary exchange by using barrier synchronization.

We also apply another communication optimization by put-wise data transfer operation, since put-wise communication conducts one-sided communication while get-wise communication conducts two-sided communication. One-sided communication has less latency since it does not require the sender to wait for acknowledgement from the receiver. In put-wise communication, the main activity moves to the place where data source is located by `at` statement, then copy the data to destination place by `asyncCopy` method. On the other hand, In get-wise communication, the main activity moves to the destination place where data to be copied, then copy the data from the source place by `asyncCopy` method.

In our current implementation, communication may not be fully overlapped. This is mainly due to the limitation of communication patterns in X10. Barrier synchronization requires all the places to synchronize at the same point of computation, which results in the requirement of barrier synchronizations in each direction. On the other hand, in our lattice QCD implementation in MPI, instead of barrier synchronization, process-to-process synchronization is used in each direction which enables to overlap communications in multiple directions using simple asynchronous send and receive operations (Figure 2).

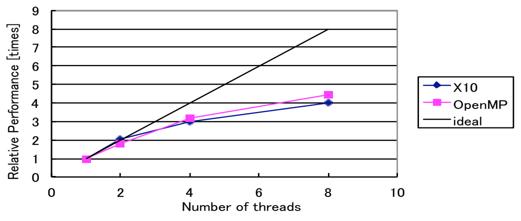

Figure 3. Comparative strong scaling of multi-threaded lattice QCD in X10 and OpenMP on problem size of (x, y, z, t) = (16, 16, 16, 32).

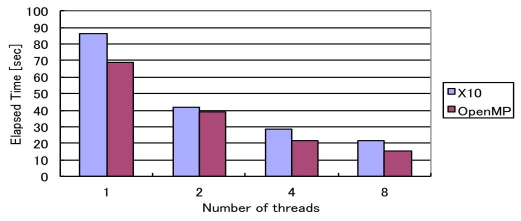

Figure 4. Elapsed time of multi-threaded lattice QCD in X10 and OpenMP on problem size of (x, y, z, t) = (16, 16, 16, 32).

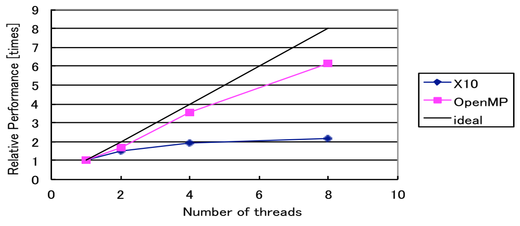

Figure 5. Comparative strong scaling of multi-threaded lattice QCD in X10 and OpenMP on problem size of (x, y, z, t) = (8, 8, 8, 16).

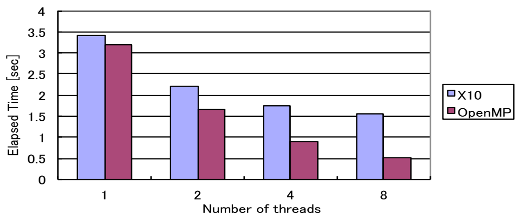

Figure 6. Elapsed time of multi-threaded lattice QCD in X10 and OpenMP on problem size of (x, y, z, t) = (8, 8, 8, 16).

3.3 Hybrid Parallelization of Lattice QCD in X10

We apply hybrid parallelization on multiple places and activities into our lattice QCD in X10. The objective of our hybrid parallelization is to find the optimal balance between the number of activities and the number of places on each node as well as to avoid the overhead of using more than the optimal number of activities or places.

There are two parallelization strategies for places.

1. The main place activates places for each computational part which can be parallelized. Each parallelized part is synchronized by `finish` statement.
2. The main place activates places only once at the beginning of computation. Barrier synchronization is used for synchronization among places. `finish` statement is used only once.

We adopt the strategy 2 in order to reduce the number of synchronizations. We observe calling `finish` for each part of computation as the strategy 1. causes increase of synchronization overheads.

There are two parallelization strategies for activities as well.

1. The main activity invokes activities for each computational part which can be parallelized. Each parallelized part is synchronized by `finish` statement.
2. The main activity invokes activities only once at the beginning of computation. Clock-based synchronization is used for synchronization among activities. `finish` statement is used only once.

We adopt the strategy 1 since we observe `finish` performs better scalability compared with clock-based synchronization.

4. Performance Evaluation

We conducted performance evaluation of our lattice QCD in X10. The objective of the experiments is to understand parallel efficiency of our lattice QCD in X10. We also investigated comparative performance with lattice QCD in MPI. We measured the elapsed time of iterations of CG method until convergence. Typically CG method takes 300 to 500 iterations. We compared the performance of lattice QCD in native X10, which is compiled to C++, with the baseline implementation in C++ and MPI.

We use two computing environments. One is IBM BladeCenter HS23 [15] for evaluation of the performance of multi-threaded lattice QCD on single node, and the other is IBM Power 775 [16] for evaluation of the performance of distributed lattice QCD on multiple nodes. We used X10 2.4.0 build with `-Doptimize=true -DNO_CHECKS=true`. IBM BladeCenter HS23 is configured as follows. We use HS23 7875-C5J model, which has 2 sockets of Xeon E5-2680 CPU (2.70 GHz, 8 cores, L1, L2, L3 cache sizes are 32KB, 256KB, 20MB, SMT enabled), 32 GB of DDR3 RAM. g++ 4.4.6 is used for C++ compiler. MPICH2 [10] 1.2.1 is used for MPI. Compiler options for native X10 are `-x10rt mpi -O -NO_CHECKS` and for MPI C++ are `-O2 -finline-functions -fopenmp`. We use IBM Power 775 up to 13 compute nodes for scalability study, which has Power 7 CPU (3.84 GHz, 32 cores, SMT enabled), 128 GB of memory. xlC_r 12.1 is used for C++ compiler. Parallel Active Messaging Interfaces (PAMI) is used for message passing. Compiler options for native X10 are `-x10rt pami -O -NO_CHECKS` and for PAMI C++ are `-O3 -qsmp=omp`.

4.1 Performance of Multi-threaded Lattice QCD

We conducted performance experiments on a single compute node for measuring the effect of multi-threading. We compared the performance of multi-threaded lattice QCD in X10 with lattice QCD parallelized in OpenMP [11].

First we evaluated strong scaling of the multi-thread parallelization using up to 8 threads using lattice QCD in X10 and in C++ with OpenMP. Multi-activities are invoked for multi-threading in X10. Note that one thread is assigned on one activity and only one place is used. The result of strong scaling based on performance on 1 thread of each implementation on problem size of (x, y, z, t) = (16, 16, 16, 32) is shown in Figure 3 and elapsed time on each number of threads is shown in Figure 4. The result in Figure 3 exhibits lattice QCD in X10 performs 4.01 times speedup on 8 threads compared

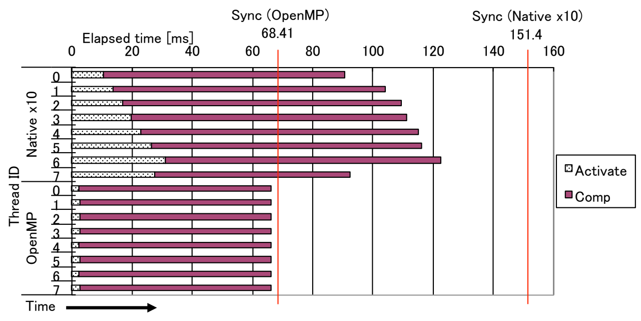

Figure 7. Performance breakdown of multi-threaded lattice QCD in X10 and OpenMP. x-axis shows the elapsed time of each phase, where white bars show thread invocation time, red bars show computation time, and vertical red lines show the time when thread synchronizations are finished. y-axis shows thread indices for each implementation.

with on 1 thread. We see that the results of lattice QCD in X10 exhibit comparative scalability compared with OpenMP. The result in Figure 4 exhibits that lattice QCD in X10 performs 71.7% of lattice QCD in C++ with OpenMP on 8 threads.

We then evaluated the multi-thread parallelization on different problem size. We used (x, t, z, t) = (8, 8, 8, 16), which is 16 times smaller than the previous experiment. The results of strong scaling and elapse time are shown in Figures 5 and 6, respectively. The result in Figure 5 exhibits that lattice QCD in X10 on 8 thread performs 2.18 times speed up from on 1 thread, which is smaller than the case on (x, y, z, t) = (16, 16, 16, 32). Figure 6 exhibits that lattice QCD in X10 performs 33.4% of lattice QCD in C++ with OpenMP on 8 threads. We breakdown the performance in X10 on 8 threads. Figure 7 shows each phase of consecutive elapsed time of 460 CG steps in x-plus way computation including thread activation time, computation time, and threads synchronization time. The result exhibits thread activation overhead accounts for 20.5%, and synchronization overhead accounts for 19.2% of total execution time each other. The result also exhibits computation time is 36.3% slower compared with in OpenMP.

4.2 Performance of Hybrid Multi-threaded Lattice QCD

We compared the performance of hybrid multi-threaded performance of lattice QCD in X10 with the baseline implementation in C++ parallelized by OpenMP and MPI. We parallelized using both multi-activities and multi-places in X10 on a single node. We vary the number of processes and threads such that the number of processes times the number of threads is constant to be 16 or 32 (as shown in Figures 8 and 9). We used problem size of (x, y, z, t) = (16, 16, 16, 32).

The results exhibit 2 threads per node exhibits the best performance in X10 in both cases. The results indicate that increasing the number of processes scales better than increasing the number of threads. The result also indicates that hybrid parallelization performs better than using only multiple processes or multiple threads (16 processes with 2 threads performs the best in the experiments).

In the case of using C++ and MPI, The result exhibits using 4 processes and 4 threads performs the best. The result also exhibits significant performance degradation when the number of processes times the number of threads is 32 compared with the case of 16. A possible reason for the degradation is assigning more number of processes and threads than the number of physical CPU cores causes resource contention among processes or threads.

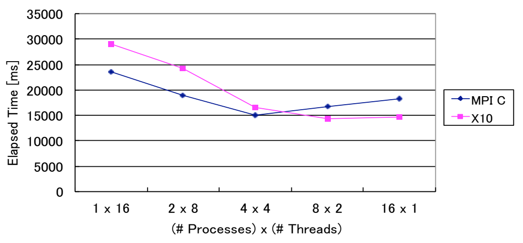

Figure 8. Elapsed time of hybrid multi-threaded lattice QCD in X10 and MPI + OpenMP, where the number of processes times the number of threads is equal to 16.

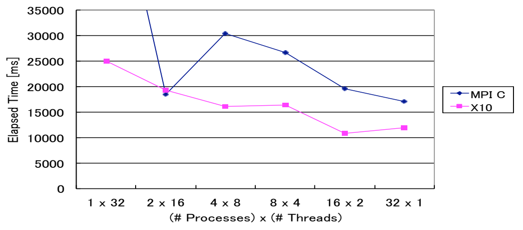

Figure 9. Elapsed time of hybrid multi-threaded lattice QCD in X10 and MPI + OpenMP, where the number of processes times the number of threads is equal to 32.

4.3 Performance of Distributed Lattice QCD

We evaluated scalability of the distributed implementation of lattice QCD in X10 on multiple compute nodes using IBM Power 775. We measured strong scaling and weak scaling using up to 256 places on up to 13 nodes, which means 19 or 20 places are located per node. We used problem size of (x, y, z, t) = (32, 32, 32, 64) for strong scaling and (x, y, z, t) = (16, 16, 16, 32) per place for weak scaling. We also compared the performance of lattice QCD in X10 with the performance in MPI. Note that we did not apply hybrid parallelization using multi-activities on one place in the experiments.

First we show the result of strong scaling performance of lattice QCD in X10 compared with in C++ and MPI in Figure 10 as well as elapsed time in Figure 11, where x-axis shows the number of processes and y-axis shows relative performance based on the performance on 1 process for each implementation. We see that 102.8 times speedup using 256 places compared with using 1 place. The result also exhibits lattice QCD in MPI performs better scalability compared with in X10. The performance of lattice QCD in MPI on 256 places performs 2.58 times faster than in X10.

In order to understand the effect of communication optimization using put-wise data transfer operations, we compared the performance of lattice QCD using put-wise operations with using get-wise operations. We also measured the performance using get-wise operations with overlapping as much as possible with barrier synchronization, shown in Figure 12. Overlapping multiple communications using `asynchCopy` in 4 directions and one barrier synchronization. The result of the comparison are shown in Figures 13 and 14. As we explained in section 3.2, using put-wise operations shows better scalability than using get-wise operations, even though overlapping is more optimized when using get-wise operations. The reason is that underlying implementation of put-wise communication uses one-sided communication while get-wise communication uses two-sided communication in MPI. Note that

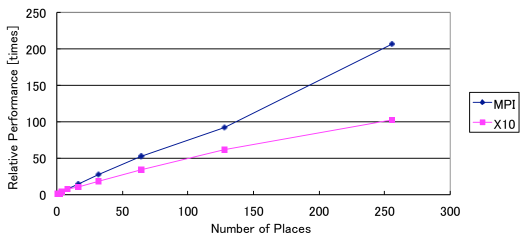

Figure 10. Comparative strong scaling of distributed lattice QCD in X10 and MPI on problem size of (x, y, z, t) = (32, 32, 32, 64).

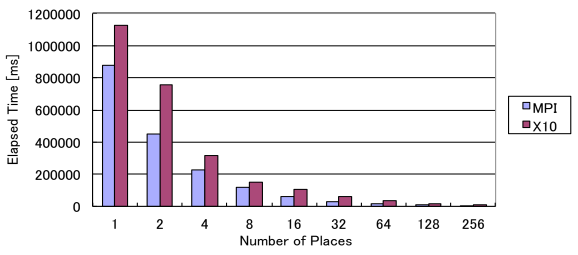

Figure 11. Elapsed time of distributed lattice QCD in X10 and MPI on problem size of (x, y, z, t) = (32, 32, 32, 64).

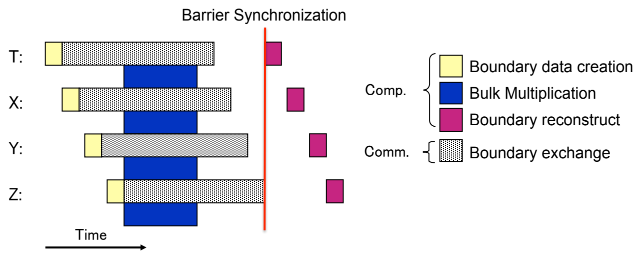

Figure 12. Implementation of overlapping on lattice QCD application in X10 using get-wise operations.

overlapping multiple communications should be possible with put-wise operations as with get-wise operations, however, we did not get correct results in IBM Power 775 with PAMI. We will further investigate the reason for this phenomenon.

We also evaluated weak scaling performance with MPI. The result of weak scaling performance is shown in Figures 15 and 16, where x-axis shows the number of processes and y-axis shows relative performance based on the performance on 1 process for each implementation. The result exhibits the lattice QCD in X10 performs 97.5 times faster on 256 places from 1 place. The result also indicates the lattice QCD in MPI performs 2.26 times faster on 256 places, which means the lattice QCD in MPI scales better than in X10. A possible reason for the performance degradation when using the number of places is that the lattice QCD in MPI has larger time region of communication overlapping than in X10. We plan to further investigate for clarifying the reason and improving the performance.

4.4 Discussion

From the experiments we conducted, we revealed several important issues on X10. As for multi-threading, although X10 exhibits com-

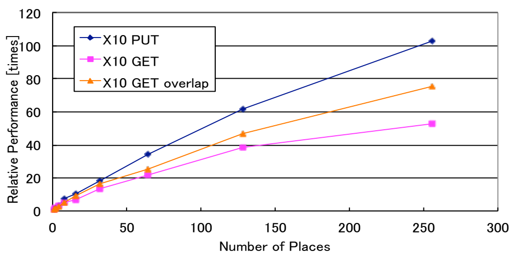

Figure 13. Comparative strong scaling of distributed lattice QCD in X10 among different communication strategies on problem size of (x, y, z, t) = (32, 32, 32, 64).

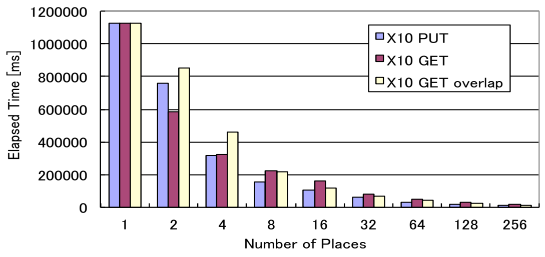

Figure 14. Elapsed time of distributed lattice QCD in X10 among different communication strategies on problem size of (x, y, z, t) = (32, 32, 32, 64).

parable multi-thread scalability with OpenMP, X10 exhibits significant overhead of thread activation as well as threads synchronization, as shown in Figure 7. As for hybrid parallelization, using up to 2 threads per node and increasing there number of places performs the best in our environment. We see that the increasing the number of places often scales better than increasing the number of threads.

As for scalability on multiple nodes, our current lattice QCD implementation in X10 scales but not as much as in MPI. A possible reason is limited communication overlapping in X10 due to the lack of tuning flexibility which is related to APGAS programming model in X10, as shown in Figures 1 and 2. In our lattice QCD in X10, barrier synchronization is used for synchronizing boundary data exchanges and bulk computations. The reason for using barrier synchronization is that confirmation of boundary data exchanges are required for next computation (boundary reconstruct). On the other hand, in our lattice QCD in MPI, process-to-process synchronization is conducted for neighbor processes in each direction independently, by using standard asynchronous communication functions such as `MPI_Isend`, `MPI_Irecv` and `MPI_Wait`. This process-to-process synchronization in MPI makes more use of overlapping compared with the barrier synchronization in X10. However, there is still rooms for improvements. For example, place-to-place synchronization in X10 would improve scalability by more overlapping communication. In our current implementation of lattice QCD in X10, all the places are synchronized at the same time by barrier synchronization in each computational part. Using `uncountedCopy` may enable place-to-place synchronization.

5. Related Work

There is work on high performance large-scale lattice QCD implementations. Doi et al. work on a peta-scale lattice QCD implementation [17] on Blue Gene/Q supercomputer [14]. The implementa-

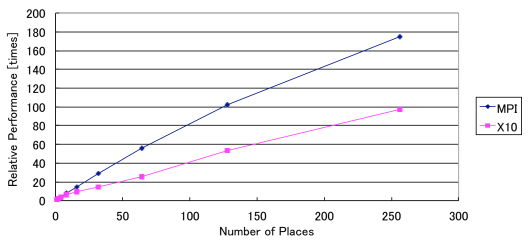

Figure 15. Comparative weak scaling of distributed lattice QCD in X10 and MPI, where problem size is (x, y, z, t) = (16, 16, 16, 32) for each process.

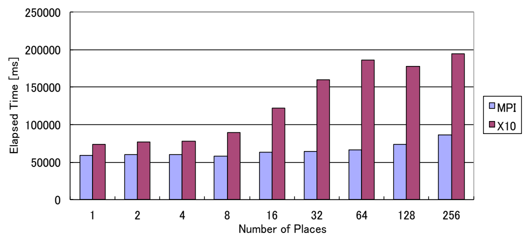

Figure 16. Elapsed time of distributed lattice QCD in X10 and MPI on problem size of (x, y, z, t) = (16, 16, 16, 32) for each process.

tion fully optimizes overlapping communication by computation. The implementation also applies node-mapping optimizations for fully utilizing network topology on Blue Gene/Q. In our work, we compare the scalability of our lattice QCD implementation in X10 with in MPI. Our implementation in MPI also applies overlapping technique in the same way as this work. Shan et al. work on computing lattice QCD using multiple GPUs [18].

There is work on performance comparison of PGAS programming model with standard message passing model [19]. The authors compare the performance of Co-Array Fortran (CAF) [12] with standard message passing on micro benchmarks.

There is also work about hybrid programming model of PGAS and MPI [20]. The authors introduced hybrid programming of Unified Parallel C (UPC) and MPI, which allows MPI programmers incremental access of a greater amount of memory by aggregating the memory of several nodes into a global address space.

6. Conclusion

We gave a detailed comparative analysis of APGAS model in X10 and the standard message passing model with the lattice QCD application. We implemented lattice QCD in X10 using APGAS programming model with several optimizations. We analyzed parallel efficiency of X10 and compared the performance of lattice QCD in X10 with the baseline implementation in C++ and MPI. The results show that our X10 implementation scales up to 256 places. The results also indicate the tradeoff between performance and productivity through the comparison of X10 with MPI.

In future work, we will further investigate the performance bottleneck of our lattice QCD in X10. We will investigate communication overlapping efficiency in X10 and also consider to use place-to-place synchronization rather than barrier synchronization. We will also plan to conduct performance experiments on other computing environments, such as TSUBAME2.5 [21] and Blue Gene/Q.

Acknowledgments

This work was partially supported by JSPS KAKENHI Grant Number 26011503.

References

[1] B. Stroustrup. *The C++ Programming Language, Fourth Edition*. Addison-Wesley Publishing Company. 2013.

[2] R. Gupta. Introduction to Lattice QCD, 1997. Lectures given at the LXVIII Les Houches Summer School ”Probing the Standard Model of Particle Interactions.

[3] S. Ueda, S. Aoki, T. Aoyama, K. Kanaya, H. Matsufuru, S. Motoki, Y. Namekawa, H. Nemura, Y. Taniguchi, and N. Ukita. Bridge++: an object-oriented C++ code for lattice simulations. In *31st International Symposium on Lattice Field Theory - LATTICE 2013*, 2013.

[4] PGAS – Partitioned Global Address Space Languages. `http://www.pgas.org/`.

[5] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. The Asynchronous Partitioned Global Address Space Model. In *The First Workshop on Advances in Message Passing (co-located with PLDI 2010)*, 2010.

[6] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification Version 2.4, February 2014. URL `http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA ’05)*, 2005.

[8] X10: Performance and Productivity at Scale. `http://x10-lang.org/`

[9] MPI: A Message-Passing Interface Standard, Version 2.2. `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`

[10] MPICH — High-Performance Portable MPI. `http://www.mpich.org/`

[11] OpenMP Application Program Interface, Version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`

[12] Co-Array Fortran `http://www.co-array.org/`

[13] Berkeley UPC - Unified Parallel C `http://upc.lbl.gov/`

[14] IBM Blue Gene/Q, `http://www.ibm.com/systems/technicalcomputing/solutions/bluegene/`

[15] IBM BladeCenter HS23. `http://www.ibm.com/systems/bladecenter/hardware/servers/hs23/`.

[16] IBM Power 775. `http://www.ibm.com/systems/power/hardware/775/`.

[17] J. Doi. Peta-scale Lattice Quantum Chromodynamics on a Blue Gene/Q Supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[18] R. Babich, M. A. Clark, B. Joó, G. Shi, R. C. Brower, and S. Gottlieb. Scaling Lattice QCD Beyond 100 GPUs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[19] H. Shan, N. J. Wright, J. Shalf, K. Yelick, M. Wagner, and N. Wichmann. A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS Languages and Comparison with MPI. In *Proceedings of the Second International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, 2011.

[20] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur. Hybrid Parallel Programming with MPI and Unified Parallel C. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, 2010.

[21] S. Matsuoka. The TSUBAME2.5 Evolution. *Tsubame e-Science Journal*, 10:2 – 8, 2013. `http://www.gsic.titech.ac.jp/en/TSUBAME_ESJ`.

A. Source Code of Lattice QCD in X10

Source code of our lattice QCD code in X10 as well as in MPI + OpenMP can be obtained from the following URL: `http://sourceforge.net/p/x10/code/HEAD/tree/applications/trunk/LatticeQCD/`.

B. Pseudo Code of Lattice QCD in X10

B.1 Lattice QCD with Put-wise data transfer in X10

Algorithm 1 Dirac Operator with Put-wise Data Transfer in X10.

```
class WilsonDslash {
    def Dopr_Put(v: WilsonVectorField, u: SU3MatrixField, w:
WilsonVectorField, cks :Double, bx : LatticeComm) {
        finish { // compute in T plus direction
            val iTP = bx.neighbors()(bx.TP);
            at (Place(iTP)) async { // move to source place
                MakeTPBnd(bx,w); // boundary data creation
                bx.Put(bx.TP); // send the boundary data
            }
             v.Copy(w);
            MultTP(v,u,w,-cks); // bulk multiplication
        }
        finish { // compute in T minus direction
            val iTM = bx.neighbors()(bx.TM);
            at (Place(iTM)) async {
                MakeTMBnd(bx,w,u);
                bx.Put(bx.TM);
            }
            MultTM(v,u,w,-cks);
            SetTPBnd(bx,v,u,-cks); // boundary reconstruct
        }
        // compute in the other 3 directions (X, Y, Z)
         ...
    }
}
```

Algorithm 2 Communicator for Put-wise Data Transfer in X10.

```
class LatticeComm {
    val bufSend = new Rail[HalfWilsonVectorField](8);
    val refBufs: PlaceLocalHandle[Rail[GlobalRail[Double]]];

    def Put(dir: Long)
    {
        val size = bufSend(dir).size;
        Rail.asyncCopy[Double](bufSend(dir).v(),0,
                                        refBufs()(dir),0,size);
    }
}
```

B.2 Lattice QCD with Get-wise overlapping data transfer in X10

Algorithm 3 Dirac Operator with Get-wise Overlapping Data Transfer in X10.

```
class WilsonDslash {
    def Dopr_Get(v: WilsonVectorField, u: SU3MatrixField, w:
WilsonVectorField, cks: Double, bx: LatticeComm)
    {
        finish {
            MakeTPBnd(bx,w); // boundary data creation
            bx.Send(bx.TP);
            MakeTMBnd(bx,w,u);
            bx.Send(bx.TM);
            // compute in the other 3 directions (X, Y, Z)
            ...
            v.Copy(w);
            // bulk multiplication
            MultTP(v,u,w,-cks);
            MultTM(v,u,w,-cks);
            ...
        }
        // boundary reconstruct
        SetTPBnd(bx,v,u,-cks);
        SetTMBnd(bx,v,-cks);
        ...
    }
}
```

Algorithm 4 Communicator for Get-wise Overlapping Data Transfer in X10.

```
class LatticeComm {
    val bufSend = new Rail[HalfWilsonVectorField](8);
    val bufRecv = new Rail[HalfWilsonVectorField](8);

     def Send(dir : Long) {
        val bufRef = GlobalRail(bufSend(dir).v());
         finish {
            at(Place(iDest)) async {
            val size = bufRecv(dir).size;
             finish{
                Rail.asyncCopy[Double](bufRef,0,
                                        bufRecv(dir).v(),0,size);
            }
        }
    }
}

```