PROCEEDINGS OF SCIENCE

Lattice QCD code Bridge++ on multi-thread and many core accelerators

S. Ueda[*a], **S. Aoki**[b], **T. Aoyama**[c], **K. Kanaya**[d], **H. Matsufuru**[ef], **S. Motoki**[e], **Y. Namekawa**[g], **H. Nemura**[g], **Y. Taniguchi**[d] **and N. Ukita**[g]

[a] *Theory Center, IPNS, High Energy Accelerator Research Organization (KEK), Tsukuba 305-0810, Japan*
[b] *Yukawa Institute for Theoretical Physics, Kyoto University, Kyoto 606-8502, Japan*
[c] *Kobayashi-Maskawa Institute for the Origin of Particles and the Universe (KMI), Nagoya University, Nagoya 464-8602, Japan*
[d] *Graduate School of Pure and Applied Sciences, University of Tsukuba, Tsukuba 305-8571, Japan*
[e] *Computing Research Center, High Energy Accelerator Research Organization (KEK), Tsukuba 305-0801, Japan*
[f] *Graduate University for Advanced Studies (Sokendai), Tsukuba 305-0801, Japan*
[g] *Center for Computational Sciences, University of Tsukuba, Tsukuba 305-8577, Japan*
E-mail: sueda@post.kek.jp

We are developing an object oriented code set "Bridge++" for simulation of lattice gauge theories. It aims to be an extensible, readable, and portable workbench while keeping sufficiently high performance in actual productive runs. This paper describes the status of two extensions. One is multi-threading with OpenMP in the recently released version 1.2. The other is a design to use arithmetic accelerator devices such as GPGPUs, which is still under developing version. Feasibility test is performed with OpenCL.

The 32nd International Symposium on Lattice Field Theory,
23-28 June, 2014
Columbia University New York, NY

*Speaker.

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike Licence.

PoS(LATTICE2014)036

1. Introduction

Lattice simulations are extensively applied not only to precision studies of QCD with controlled systematic errors, but also to nonperturbative analyses of models beyond the standard model. This is largely owing to rapid increase of computer power. On the other hand, algorithms and programming techniques have become more and more involved to fully make use of powerful architecture of massively parallel clusters and arithmetic accelerators. To catch-up the rapid development of the research frontier, it is desired to prepare a general code-set applicable in a wide range of lattice simulations keeping high performance, based on a uniform design policy.

We have been developing a general-purpose code set named Bridge++ since 2009 [1] which has the followings features:

- Readability: the code structure is transparent so as to be understandable even for beginners.
- Extensibility: the code is easy to be modified for testing new ideas.
- Portability: the code runs not only on laptop PC but also on supercomputers.
- High-performance: the code has a high performance enough for productive researches.

The code is written in C++ so as to adopt virtues of the object-oriented programming. The first public version was released in July 2012. It was parallelized by MPI for cluster systems with distributed memory while multi-threading was not supported yet. For recent massively parallel clusters, hybrid parallelization for multinode and multicore is mandatory to achieve a high performance. In the version 1.2 released in September 2014, we started to support the hybrid parallelization employing OpenMP. As for the arithmetic accelerators such as GPGPUs and coprocessors, we have been developing a system to use them and testing feasibility of OpenCL [2].

This paper is a status report of these issues. Section 2 describes our approach to the multi-thread parallelization. In Section 3, we explain our implementation to control the arithmetic accelerators using OpenCL. Section 4 is devoted to summary.

2. Multi-thread

Recent parallel cluster systems consist of computing nodes, each of which possesses several processor cores that share the memory on the node. For the internode parallelization, MPI (Message Passing Interface) is commonly used, while some systems may provide dedicated communication libraries that are more efficient. In Bridge++, the internode communication is implemented in the Communicator class that wraps MPI or machine-specific API (application programming interface). On the other hand, the intranode parallelization on the SMP (symmetric multiprocessor) systems via multi-threading had not been incorporated into Bridge++ until recently.

For multi-threading in Bridge++, two candidates are considered: OpenMP (Open Multi-Processing) and Pthreads (POSIX threads). OpenMP is a directive-based language extension with run-time environment: a user specifies which parts of the code should be parallelized by inserting directives, and a compiler generates a multi-threaded program automatically. Several API functions and environment variables are provided to control run-time behavior. On the other hand, Pthreads is an API-based library for which all the thread operations are managed through functional interfaces.

PoS(LATTICE2014)036

While Pthreads enables detailed control of the thread, programming becomes involved. For this reason, we decided to adopt OpenMP for multi-threading in Bridge++.

2.1 Object oriented programming and OpenMP

Bridge++ employs object oriented programming, in which programs are constructed based on *objects* that contain data members and methods to handle them. The objects are, in general, constructed and destructed arbitrarily in the program. OpenMP adopts a fork-join model in which threads are created at the beginning of the parallel region and destructed at the end. A minimal modification to incorporate OpenMP in this framework is to confine parallel region within the classes. However, if the creation and destruction of threads occur frequently, their overheads may become severe. In this regard, it would be better to let the parallel region wider. Another issue concerns object management. If an object is created inside the parallel region, it is thread-private and not shared by other threads. This is a problem for objects that represent fields, since the operations of them are to be parallelized by threads. We decide to create objects that are to be shared by the threads prior to starting the parallel region. The functions that may be called inside the parallel region are modified so as to be thread-safe.

The above prescription is applied to our code in version 1.2. As the first step, multi-threading is implemented in the Wilson and clover fermion operators and all solver algorithms. The parallel region encloses a call of the solver algorithm.

2.2 OpenMP and MPI

For the hybrid parallelization with OpenMP and MPI, subtlety lies in which thread executes the communication. The following four threading support levels are prepared in MPI.

- `MPI_THREAD_SINGLE`: Only one thread will execute.
- `MPI_THREAD_FUNNELED`: The process may be multi-threaded, but only the main thread can make MPI calls.
- `MPI_THREAD_SERIALIZED`: The process may be multi-threaded, and any thread can make MPI calls though they cannot execute MPI at the same time.
- `MPI_THREAD_MULTIPLE`: Multiple threads may call MPI without limitation.

The available support level depends on the system and implementation of MPI. In Bridge++, `MPI_THREAD_FUNNELED` is assumed to be supported, because it is the minimal requirement for the multi-thread support in MPI.

When a system provides a dedicated communication library that is more efficient than MPI, we adopt the library as an alternative implementation of the Communicator class and multi-thread supports, which users may choose to use. We applied this to an IBM Blue Gene/Q (BG/Q) system, for which low-latency communication library BGNET is available.

2.3 Present performance with OpenMP

We test the performance of Bridge++ ver.1.2 on an IBM BG/Q system with application of BGNET-OpenMP hybrid parallelization. The performance is measured for multiplication of Wilson and clover fermion operator with lexical and even-odd site ordering on $16^3 \times 32$ lattice. We

PoS(LATTICE2014)036

	Bridge++ code		w/ BG/Q Wilson library	
operator	4 threads	8 threads	4 threads	8 threads
Wilson(lex) [GFlops (%)]	335.5 (5.1)	329.9 (5.0)	885.4 (13.5)	920.0 (14.0)
Wilson(e-o) [GFlops (%)]	187.9 (2.9)	229.4 (3.5)	882.6 (13.5)	897.9 (13.7)
Clover(lex) [GFlops (%)]	398.2 (6.1)	386.9 (5.9)	585.5 (8.9)	571.3 (8.7)
Clover(e-o) [GFlops (%)]	189.5 (2.9)	195.4 (3.0)	517.7 (7.9)	470.3 (7.2)

Table 1: Performance of multiplication of Wilson and clover fermion operators with lexical ('lex') and even-odd ('e-o') site ordering on a 32-node job class of IBM Blue Gene/Q. The performance is compared for the cases with and without BG/Q Wilson library. The number of BGNET ranks times number of threads per node is set to be 64.

use a 32-node class of BG/Q at KEK whose peak performance is 204.8 GFlops/node. On BG/Q, at most 64 threads per node is available. We fixed the total number of threads per node to be 64 and varied the number of threads per BGNET rank. The result is summarized in Table 1[1]. The table displays the result of two cases, 4 threads and 8 threads. The sustained speeds do not depend on the number of threads, suggesting a weak scaling of MPI and OpenMP. For comparison, results with an optimized code of the Wilson fermion operator called BG/Q Wilson library [4] are also presented. While multi-threading successfully works, the Bridge++ code still has a large room to be tuned.

3. Arithmetic accelerators

Arithmetic accelerators, such as GPGPUs and Xeon Phi coprocessor, are rapidly increasing their performance. They enable to acquire a large numerical power with less cost and electric power, and already are widely used in lattice simulations. There are indeed open source libraries such as QUDA [5] for NVIDIA GPUs. There are several programming frameworks including CUDA SDK for NVIDIA's GPUs. As a general framework, candidates are OpenACC [6] and OpenCL [7]. The former is a directive-based language extension, like OpenMP, and the latter is API based, like Pthreads. As the first implementation to make use of the accelerators, we adopt OpenCL to control general accelerator devices in various environments.

3.1 OpenCL

OpenCL (Open Computing Language) is an open standard framework for a parallel programming in heterogeneous platforms, such as CPUs, GPUs, FPGAs, and other processors. The specifications are maintained by Khronos Group with contributions by AMD, NVIDIA, and so on. The left panel of Figure 1 shows an image of OpenCL framework. OpenCL works on an abstract hardware layer (orange part) and controls accelerator hardwares (blue part) through it. Thus applications can be developed independently of specific architecture. This matches our design policy with respect to portability.

[1] In Ref [3], we reported higher performance data, caused by incorrect system Flops counter.

PoS(LATTICE2014)036

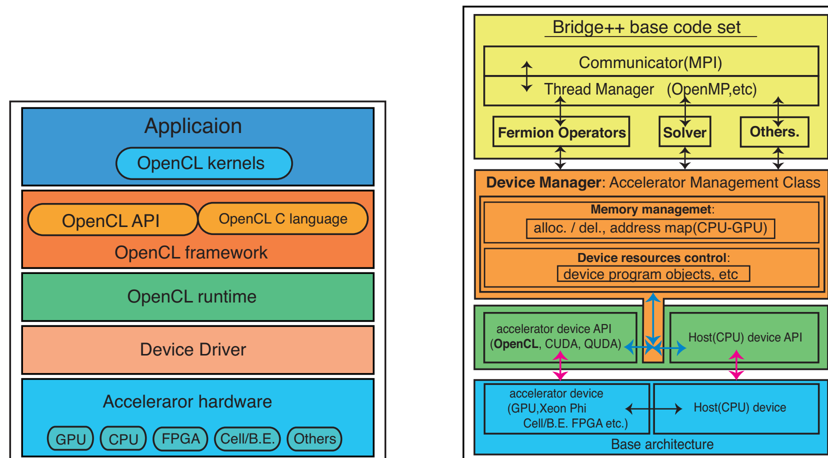

Figure 1: Working image of OpenCL framework (left panel) and schematic structure of Bridge++ to handle an accelerator (right).

3.2 Implementation in Bridge++

An application work flow by using accelerator devices is summarized as follows.

1. Allocate *buffer objects* at device memory.
2. Transfer data from host to device.
3. Execute *kernel code* on device.
4. Transfer data from device to host.
5. Delete unnecessary buffer object from device memory

Prior to these steps, some preparation for using devices is necessary such as creating *contexts* and compiling *kernel codes*. Each step of this work flow is handled through OpenCL APIs. Usually, steps 2 and 4 become bottlenecks, because of a narrow bandwidth between host and device. To avoid explicit appearance of these OpenCL APIs in individual classes, we develop a device manager class. It encapsulates the OpenCL APIs, so as to simplify the procedures and to accommodate device differences easily. The right panel of Figure 1 schematically expresses the adopted design. This device manager class manages memory objects on the devices and data transfer between the host processor and the devices (orange part of the right panel in Fig. 1).

As another abstraction, we implement a class that controls field data on the device. This class contains functions to transfer data between the host and device and to perform linear algebraic operations using device codes. At the construction of an object of this class, an associated memory space on the device is allocated through the device manager. Using the objects of this class, for example, solver algorithms can be written without referring to the device specific APIs. It also manages conversion of data layout of fields between host code and device code. It may be crucial for optimization to choose appropriate data layout, as explained below.

PoS(LATTICE2014)036

3.3 Present performance on arithmetic accelerators

On the accelerator device such as GPGPUs, we have to use memory bandwidth efficiently by applying so-called coalesced memory access. This requires changing the data layout suitable for devices that may be different from host code. As explained previously, the conversion of data layout is implemented in the class for the device fields. In addition, several techniques to reduce data transfer between device memory and processing elements are adopted, such as reconstructing the third column of $SU(3)$ matrix from the other columns on-the-fly. (Note that the sustained performance demonstrated below does not include the floating-point operations for the reconstruction.)

In Table 2, we summarize the present performance of our code on NVIDIA and AMD GPU. The performance is measured for multiplication of Wilson operator (represented as "mult" in the table) and CG solver on $16^3 \times 32$ lattice using a single accelerator device. The double precision sustained performance values for both devices are 12% for mult, and 4-6% for solver. Although the peak performance in the single precision is higher than that in the double precision by a factor of four, the sustained speed is only doubled. It suggests the memory bandwidth is a bottleneck. Further tuning of the code and architecture dependent optimization are underway.

Accelerator Specifications:		
Device name	Radeon HD 7970	Tesla K40
Vendor	AMD	NVIDIA
Architecture	Southern Islands	Kepler
Core clock[MHz]	925	745
Peak SP performance[GFlops]	3789	4290
Peak DP performance[GFlops]	947	1430
Global memory size[Gbytes]	3	12
Peak memory B/W [Gbyte/s]	254	288
Results for mult:		
Single precision [GFlops (%)]	222.2 (5.9)	360.8 (8.2)
Double precision [GFlops (%)]	113.2 (12.0)	188.5 (12.9)
Results for solver:		
Single precision [GFlops (%)]	54.52 (1.4)	163.8 (3.7)
Double precision [GFlops (%)]	39.28 (4.1)	87.0 (5.9)

Table 2: Accelerator specifications and results of performance measurement for multiplication of Wilson operator and CG solver.

4. Summary

Two major extensions in Bridge++ are presented, that are supports for multi-threading in release version 1.2 and arithmetic accelerators in develop version. OpenMP is adopted for multi-threading, which requires simpler coding than Pthreads. Hybrid parallelization by BGNET and OpenMP is tested on IBM BG/Q. The weak scaling of mult is confirmed. Arithmetic accelerators are covered in Bridge++ by use of OpenCL. We create a device manager class to conceal OpenCL API, and control data layout and transfer between host and device. Sustained speeds on accelerator

PoS(LATTICE2014)036

devices of AMD and NVIDIA are measured. We achieve 12% for mult, and 4-6% for CG solver in the double precision, suffering a bottleneck of the memory bandwidth. Additional optimizations are in progress.

Acknowledgment

We thank J. Doi of IBM Japan for useful discussion on the multi-threading and T. Doi on simultaneous use of MPI and BGNET on Blue Gene/Q system. The code was developed and tested on Hitachi SR16000 and IBM System Blue Gene/Q at KEK under a support of its Large-scale Simulation Program (No.13/14-19), Hitachi SR16000 at YITP in Kyoto University, K-computer at RIKEN Advanced Institute for Computational Science, HA-PACS at University of Tsukuba under a support for its Interdisciplinary Computational Science Program, and FX10 at University of Tokyo. This work was supported in part by MEXT SPIRE and JICFuS. This work was supported by Grants-in-Aid for Scientific Research Grant Numbers 24540250, 25400284.

References

[1] Bridge++ website, `http://bridge.kek.jp/Lattice-code/`.

[2] S. Motoki *et al.*, Procedia Computer Science 29, 1701-1710 (2014).

[3] S. Ueda *et al.*, PoS LATTICE **2013**, 412 (2014);

[4] `http://sourceforge.net/projects/bgqwilson/`.

[5] M. A. Clark, R. Babich, K. Barros, R. C. Brower and C. Rebbi, Comput. Phys. Commun. **181**, 1517 (2010) [arXiv:0911.3191 [hep-lat]].

[6] OpenACC website, `http://www.openacc-standard.org/`

[7] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems.[Online]. `http://www.khronos.org/opencl`

PoS(LATTICE2014)036