PROCEEDINGS OF SCIENCE

Bridge++: an object-oriented C++ code for lattice simulations

S. Ueda[*a], **S. Aoki**[b], **T. Aoyama**[c], **K. Kanaya**[d], **H. Matsufuru**[e], **S. Motoki**[f], **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] *Aizu University, Aizu-Wakamatsu 965-8580, Japan*

[g] *Center for Computational Sciences, University of Tsukuba, Tsukuba 305-8577, Japan*

E-mail: sueda@post.kek.jp

We are developing a new code set "Bridge++" for lattice simulations. It is aimed to be an extensible, readable, and portable workbench, while it keeps a high performance. Bridge++ covers conventional lattice actions and numerical algorithms. The code set is constructed in C++ with an object oriented programming. We explain our strategy and the basic design of Bridge++. We also present our current status of this project, including the sustained performance on several systems.

31st International Symposium on Lattice Field Theory - LATTICE 2013
July 29 - August 3, 2013
Mainz, Germany

*Speaker.

PoS(LATTICE 2013)412

1. Introduction

Lattice simulations have become an indispensable tool for non-perturbative analysis of gauge theories. In contrast, programming techniques have been more and more involved. On the hardware side, the hybrid parallelization with MPI and OpenMP is inevitable for the current large-scale system, such as K-Computer and Blue Gene/Q. GPGPUs require an additional development of the code with CUDA or OpenCL. On the software side, significant development of numerical algorithms makes a simulation code more complex. In addition, a code is tuned for each machine to obtain a high performance. These codes are hard to be understood for beginners. It leads a more serious problem that new ideas or new physical quantities cannot be tested quickly.

In view of this situation, we started a project to develop a new common code set "Bridge++" for lattice QCD simulations. We adopt C++ language to make use of the object-oriented design. Bridge++ has a great deal of readability, keeping a sufficient performance for frontier works. The machine dependent part of the code is hidden as much as possible. It is actively being developed to extend functions, brush up its design and the implementation, improve the performance, and provide more documents. Our code set is available from our website under the GNU General Public License [1, 2].

This paper is organized as follows. In the next section, our development policy of Bridge++ project is described. Section 3 summarizes our current status of the project and future prospects.

2. Lattice QCD code Bridge++

Bridge++ project starts on the following background. There are many lattice QCD codes, such as MILC code (in C, USA) [4], CPS++ (C++, USA) [5], Chroma (C++, Europe) [6], and Lattice QCD Toolkit (Fortran, Japan) [7]. In Japan, however, there has been no genuine code set other than Lattice QCD Toolkit, although Japan is one of centers of lattice QCD simulations. It is desirable to possess a code set under our management to reflect instantly a feedback of modifications, machine specific tunings, and other improvements. Another point is that a new code set has no black box. It does not have a phantasm from the development history. We can confirm and understand the code genuinely.

For these reasons, we have started a development of our new code set, named Bridge++, in 2009. We aimed that this general-purpose code set should have 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 research.

To achieve these goals, we adopt an object oriented design in C++ programming language. MPI is used for distributed memory machines. In Bridge++ code set, we use several simple idioms repeatedly. The repetition helps beginners to understand the idioms and the code structure. The code is managed with a repository and bug tracking system.

In July 2012, the first version of Bridge++ was released. The latest version is 1.1.2. It contains fundamental lattice actions, linear algebraic algorithms, and various physical quantities. The documents are compiled on wiki as well as TeX based reports. We also provide a HTML document generated by doxygen [9] from the comments in the source file.

2.1 Object-oriented programing

Object-oriented programing (OOP) is employed in Bridge++. It is based on "Objects", which are defined as sets of data fields and methods. A functionality is provided by an interaction between objects. Data field is a characteristic variable of the object, called a member data or a member variable. Method is a function that defines a behavior of the object.

OOP is characterized by the following properties.

- Encapsulation: data is handled through the interface, which prescribes the object usage.
- Inheritance: object is expandable by adding new functions.
- Polymorphism: objects with the same kind of behavior can be handled through a single interface.

These properties maximize reusability of the code. Using OOP, an interface is separated from details of the implementation. It localizes a machine specific part of the code, which is inevitable due to the optimization. The polymorphism allows us to implement an algorithm with a set of objects that have a common interface.

There is a compilation of wisdom to make use of these virtues of OOP. An example is so-called design patterns [10, 11]. The design pattern is a kind of programing idiom that frequently appears as a good solution to certain kind of problems. Their efficiency has been realized after the GoF's publication [10] that classified such idioms into 23 design patterns. The design pattern enables us to use the benefit of OOP easily, as well as to make our code transparent.

In the following, we introduce two design patterns and show how they are implemented in Bridge++.

2.2 Bridge pattern for a linear equation solver

Bridge pattern is a design pattern which decouples an interface from its implementation so that they can be switched independently. It has advantages for readability and further developments. Bridge++ employs this Bridge pattern.

A typical example of the Bridge pattern is an implementation of a solver algorithm with a fermion operator. There are numbers of fermion operators and solver algorithms. The best solver algorithm depends on a type of the fermion operator. The solver algorithm should be implemented so as to be applied to any kind of fermion operators, and simultaneously it must be changed easily.

Figure 1 shows the class diagram of the Bridge pattern based on UML (Unified Modeling Language). The interface of the fermion operator is `Fopr`. `mult()` is a function that applies the fermion operator to a given vector and returns the resultant vector. The practical implementation of `mult()` is given in a subclass of `Fopr`, such as `Fopr_Wilson` for the Wilson fermion and `Fopr_Overlap` for the overlap fermion. Similarly, the base class `Solver` defines the interface of the solver algorithms. `solve()` function returns the solution of a linear equation for a

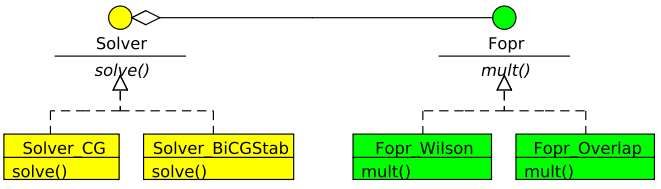

Figure 1: Class diagram of the Bridge pattern applied to a linear solver system.

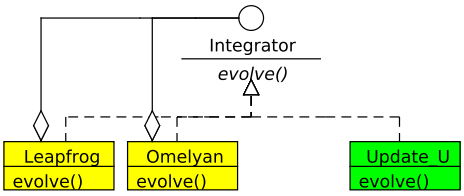

Figure 2: Class diagram of the Composite pattern applied to the HMC integrators.

given source vector. Again the practical implementation is given in subclasses, `Solver_CG` and `Solver_BiCGStab`. Bridge++ users can select any combination, such as `Fopr_Wilson` with `Solver_CG`.

Another example of the Bridge pattern is a link smearing. It is specified as a combination of a type of smearing paths and a procedure to project the resultant matrix onto SU(N). We have implemented the APE [12] and HYP-type paths [13] for the former, and the stout [14] and maximum projection for the latter. Applying the Bridge pattern, one can combine a smearing type and a projection procedure arbitrary. A similar structure is employed for a combination of a fermion operator and a link smearing.

2.3 Composite pattern and HMC integrator

Composite pattern is a design pattern that treats a group of objects as a single instance of an object. It is convenient to implement a tree structure or nested objects. In Bridge++, it is used in the multi-time step integrator for HMC,

$$V(\tau) = \left[V_P^{(F)}\left(\frac{\tau}{2N}\right) V_1\left(\frac{\tau}{N}\right) V_P^{(F)}\left(\frac{\tau}{2N}\right) \right]^N, \quad V_1(\tau) = \left[V_P^{(G)}\left(\frac{\tau}{2N_1}\right) V_U(\tau/N_1) V_P^{(G)}\left(\frac{\tau}{2N_1}\right) \right]^{N_1}, \quad (2.1)$$

where $V_P^{(G)}$ and $V_P^{(F)}$ are the integrator of P with the gauge and fermion forces, respectively. The leapfrog integrator can be replaced with an improved integrator, such as Omelyan integrator.

Figure 2 shows the class diagram of the Composite pattern applied to the integrator. The base class of the integrator is defined as `Integrator`, which has a virtual method `evolve()`. Its subclass `Update_U` plays the role of V_U. The evolution operators V and V_1 are implemented by `Leapfrog` or `Omelyan`. An object of these classes may have other objects of subclasses of `Integrator`. This relation is represented in Figure 2 as lines with diamond. In analogy to a file system, V play a role of the directory, and V_P and V_U are files in the directory.

This idiom is also applied to fermion operators. We can construct a smeared fermion operator or overlap fermion operator with arbitrary fermion operator.

2.4 Parallelization

For a current computational environment, parallelization is inevitable. Large-scale computers consist of many nodes, which communicate with each other via high speed network. Recently, each node possesses several processor cores. Hybrid parallelization is necessary.

In Bridge++, we adopt MPI (Message Passing Interface) for distributed memory parallelization, i.e. for parallel nodes. Inter-node communication such as broadcast and one-to-one data

Gauge action	plaquette, rectangle loops
Fermion action	Wilson, clover, twisted mass*, staggered*, domain-wall*, overlap*
	clover w/ isospin chemical potential
Smearing	(APE, HYP) $\times$ (maximal projection, stout)
Schrödinger functional	plaquette/rectangle for gauge, Wilson/clover for fermion
Linear algebra	CG, BiCGStab, GMRES(m)
Eigensolver	implicitly restarted Lanczos
Shiftsolver	multi-shift CG
Configuration generation	HMC, RHMC, multi-time step integrator w/ leapfrog and Omelyan
Physical observable	hadron spectrum, Wilson loop, Polyakov line

Table 1: Our development status. * mark means the function has already been implemented and being confirmed.

transfer is performed through the functions of the Communicator class, which wraps API functions of MPI. If a machine specific efficient library is available, the implementation of Communicator class can be switched with it. For an environment without MPI, so-called stub implementation of Communicator class is provided. Bridge++ works on one node as well as parallel nodes.

It is noted that a shared memory parallelization of Bridge++ is now under development. Hybrid parallelization with MPI and multi-threading has been considered. Two multi-thread libraries, Pthread and OpenMP, are compared based on a working implementation of the Wilson fermion operator. While Pthread can accomplish better performance, it is not easy to apply it to the entire code set. OpenMP has been selected as a primary method to multi-threading and performing performance tuning toward incorporating in the next release.

2.5 I/O format

In Bridge++, simulation parameters are given by ASCII files in YAML format. The parameters are held in `Parameter` object. It passes the parameters to each object. Extraction of values from files is handled by a parameter manager class. For the I/O of field objects, several formats are available including ILDG standard format for the gauge configuration. ILDG (International Lattice Data Grid) is an activity to promote sharing configuration data and standardizing data format and description of metadata [8]. The standard output is classified with a verbose level. At present 4 levels are prepared and the output level is selected for each object, such as a linear solver or an observable. There is a extra mode to generate a message with pragma for ILDG metadata generation.

3. Current status and future prospects

3.1 Status of development

Our development status of Bridge++ is summarized in Table 1. The current public version contains major algorithms and observables, though the fermion types are limited to Wilson and clover fermions. Several other fermion actions have been implemented and now being tested. For a link

Machine	Released ver.		Developing ver.	
	Wilson mult	Wilson solver	Wilson mult	Wilson solver
SR16000 1 node (32 cores)	5%	5%	-	-
BG/Q (32 nodes)				
1 rank/node $\times$ 64 threads	2-3%	2-3%	12-13%	7-8%
8 ranks/node $\times$ 8 threads	2-3%	2-3%	14-15%	12-13%

Table 2: Current status of our code performance.

smearing method, we provide any number of the APE and HYP smearing with maximal projection and stout for any fermion operator and force. For linear algorithms, iterative algorithms such as CG, BiCGStab, GMRES(m) are available. Multi-shift solver with CG algorithm and eigensolver with Implicitly restarted Lanczos algorithm are also ready. The gauge configuration generations can be performed with a multi-time step leapfrog and Omelyan integrator of HMC and RHMC.

In addition to these functions, we supply many test modules and a test manager. A test module gives a basic implementation of each function in Bridge++. The test manager controls these test modules. It provides an interactive test environment.

3.2 Performance

Our current status of code performance is summarized in Table 2. We quote the sustained performance for the HMC with clover fermions. On Hitachi SR16000, the rate to the peak performance is about 5% on 1 node (32 cores). On Blue Gene/Q, the public version runs with 2-3% on 32-nodes, while the latest code under development has increased the most time consuming solver part to more than 10%. Toward the next public release, we are now incorporating the multi-threading to Bridge++. Performance tuning is now in progress. An active investigation to make use of GPGPU is also underway.

3.3 Document

We provide a lot of documents on wiki, such as first step guide, implementation notes, verification notes and bug report/feedback. Trac wiki also provides a timeline and roadmap of our code development. Users can obtain our real time information of Bridge++. As another option, HTML document can be generated by doxygen from Bridge++ source files.

Acknowledgment

In addition to the authors of this paper, this project has been contributed by many of our colleagues, as listed in a document enclosed in the source code. This project is supported by H20 Grant-in-Aid for Scientific Research on Innovative Areas 'Research on the Emergence of Hierarchical Structure of Matter by Bridging Particle, Nuclear and Astrophysics in Computational Science', Joint Institute for Computational Fundamental Science and HPCI Strategic Program Field 5 'The origin of matter and the universe'. 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.12/13-15), 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 (No.13a-25) and FX10 at University of Tokyo. This work is supported in part by the Grand-in-Aid for Scientific Research of the Japan (Nos.20105005, 24540250, 25400284)

References

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

[2] S. Ueda et al., proceedings of 15th International Workshop on advanced computing and analysis techniques in physics, May 16-21, 2013, Beijing, China.

[3] K. G. Wilson, Phys. Rev. D **10** (1974) 2445.

[4] `http://www.physics.utah.edu/~detar/milc/`

[5] `http://qcdoc.phys.columbia.edu/cps.html`

[6] `http://usqcd.jlab.org/usqcd-docs/chroma/`

[7] `http://nio-mon.riise.hiroshima-u.ac.jp/~LTK/LTKf90.html`

[8] `http://www.usqcd.org/ildg/`

[9] `http://www.stack.nl/~dimitri/doxygen/index.html`

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* (Addison-Wesley, 1995).

[11] A. Trott and J. R. Shalloway, *Design Patterns Explained: A New Perspective on Object-Oriented Design, 2nd ed.* (Addison-Wesley Professional, 2004)

[12] T. Blum, C. E. Detar, S. A. Gottlieb, K. Rummukainen, U. M. Heller, J. E. Hetrick, D. Toussaint and R. L. Sugar *et al.*, Phys. Rev. D **55**, 1133 (1997) [hep-lat/9609036].

[13] A. Hasenfratz and F. Knechtli, Phys. Rev. D **64**, 034504 (2001) [hep-lat/0103029].

[14] C. Morningstar and M. J. Peardon, Phys. Rev. D **69**, 054501 (2004) [hep-lat/0311018].