Accelerated FDPS — Algorithms to Use Accelerators with FDPS

Masaki IWASAWA[1]**, Daisuke N**AMEKATA[1]**, Keigo N**ITADORI[1]**, Kentaro N**OMURA[1]**, Long W**ANG[2,1]**, Miyuki T**SUBOUCHI[1] **and Junichiro M**AKINO[3,1,4]

[1]RIKEN Center for Computational Science

[2]Helmholtz Institut für Strahlen und Kernphysik

[3]Department of Planetology, Graduate School of Science, Kobe University

[4]Earth-Life Science Institute, Tokyo Institute of Technology

[*]E-mail: masaki.iwasawa@riken.jp, daisuke.namekata@riken.jp, keigo@riken.jp, kentaro.nomura@riken.jp, long.wang@riken.jp, miyuki.tsubouchi@riken.jp, makino@mail.jmlab.jp

Abstract

In this paper, we describe the algorithms we implemented in FDPS (Framework for Developing Particle Simulators) to make efficient use of accelerator hardware such as GPGPUs (General-purpose computing on graphics processing units). We have developed FDPS to make it possible for many researchers to develop their own high-performance parallel particle-based simulation programs without spending large amount of time for parallelization and performance tuning. The basic idea of FDPS is to provide a high-performance implementation of parallel algorithms for particle-based simulations in a "generic" form, so that researchers can define their own particle data structure and interparticle interaction functions and supply them to FDPS. FDPS compiled with user-supplied data type and interaction function provides all necessary functions for parallelization, and using those functions researchers can write their programs as though they are writing simple non-parallel program. It has been possible to use accelerators with FDPS, by writing the interaction function that uses the accelerator. However, the efficiency was limited by the latency and bandwidth of communication between the CPU and the accelerator and also by the mismatch between the available degree of parallelism of the interaction function and that of the hardware parallelism. We have modified the interface of

user-provided interaction function so that accelerators are more efficiently used. We also implemented new techniques which reduce the amount of work on the side of CPU and amount of communication between CPU and accelerators. We have measured the performance of N-body simulations on a systems with NVIDIA Volta GPGPU using FDPS and the achieved performance is around 27 % of the theoretical peak limit. We have constructed a detailed performance model, and found that the current implementation can achieve good performance on systems with much smaller memory and communication bandwidth. Thus our implementation will be good for future generations of accelerator system.

1 Introduction

In this paper, we describe new algorithms we implemented in FDPS [Framework for Developing Particle Simulators, (Iwasawa et al. 2016; Namekata et al. 2018)], to make efficient use of accelerators such as GPGPUs (General-purpose computing on graphics processing units). FDPS is designed to make it easy for many researchers to develop their own programs for particle-based simulations. To develop efficient parallel programs for particle-based simulations requires a very large amount of work, like the work of a large team of people for many years. This is of course true not only for particle-based simulations, but practically for any large-scale parallel applications in computational science. The main cause for this problem is that modern HPC (high-performance computing) platforms have become very complex, and thus it requires lots of efforts to develop complex programs to make efficient use of such platforms.

Typical modern HPC systems are actually a cluster of computing nodes connected through a network, each with typically one or two processor chips. Largest systems at present consists of around 10^5 nodes, and we will see even larger systems soon. This extremely large number of nodes has made the design of inter-node network very difficult, and the design of parallel algorithm also has become very difficult. The calculation times of all nodes must be accurately balanced. The time necessary for communication must be small enough so that the use of large systems is meaningful. The communication bandwidth between nodes is much lower than the main memory bandwidth, which itself is very small compared to the calculation speed of CPUs. Thus, it is crucial to avoid communications as much as possible. The calculation time can show increase, instead of showing

2

decrease as it should, when we use a large number of nodes, unless we are careful to achieve good load balance between nodes and to minimize communication.

In addition, the programming environments available on present-day parallel systems are not easy to use. What is most widely used is MPI, with which we need to write explicitly how each node communicates with others in the system. Just to write and debug the program is difficult, and it has become nearly impossible for any single person or even for a small group of people to develop large-scale simulation programs which run efficiently on modern HPC systems.

Moreover, this extremely large number of nodes is just one of the many difficulties of using modern HPC systems, since within one node, there are many levels of parallelisms which should be taken care of by programmers. To make the matter even more complicated, these multiple levels of parallelism are interwoven with multiple levels of memory hierarchy with varying bandwidth and latency. For example, the supercomputer Fugaku, which is under development in Japan as of the time of writing, will have 48 CPUs (cores) in one chip. These 48 cores are divided into four groups, each with 12 cores. Cores in one group share one level-2 cache memory. The cache memories in different groups communicate with each other through cache-coherency protocol. Thus, the access of one core to the data which happens to be in its level-2 cache is fast, but that in the cache of another group can be very slow. Also, the access to the main memory is much slower, and that to local level-1 cache is much faster. Thus, we need to take into account the number of cores and the size and speed of caches at each level to achieve an acceptable performance. To make the matter even worse, many of modern microprocessors have level-3 or even level-4 caches.

As the result of these difficulties, only a small number of researchers (or groups of researchers) can develop their own simulation programs. In the case of cosmological and galactic N-body and SPH (Smoothed Particle Hydrodynamics) simulations, Gadget (Springel 2005) and `pkdgrav` (Stadel 2001) are most widely used. For star cluster simulations, NBODY6++ (and NBODY6++GPU) (Nitadori & Aarseth 2012) is effectively the standard. For planetary ring dynamics, REBOUND (Rein & Liu 2012) has been available. There has been no public code for simulations of planetary formation process until recently.

This situation is clearly unhealthy. In many cases, the physics needs to be modeled is quite simple: particles interact through gravity, and with some other interactions such as physical collisions. Even so, almost all researchers are now forced to use existing programs developed by someone else, simply because HPC platforms have become too difficult to use. To add some functionality which is not already implemented in existing programs can be very difficult. In order to make it possible for researchers to develop their own parallel codes for particle-based simulations, we have developed FDPS (Iwasawa et al. 2016).

3

The basic idea of FDPS is to separate the code for parallelization and that for interaction calculation and numerical integration. FDPS provides the library functions necessary for parallelization, and using them researchers write programs very similar to what they would write for single CPU. Parallelization on multiple nodes and on multiple cores in single node are taken care of by FDPS.

FDPS provides three sets of functions. One is for the domain decomposition. Given the data of particles in each nodes, FDPS performs the decomposition of the computational domain. The decomposed domains are assigned to MPI processes. The second one is to let MPI processes exchange particles. Each particle should be sent to the appropriate MPI process. The third set of functions perform the interaction calculation. FDPS uses parallel version of Barnes-Hut algorithm, for both of long-range interactions such as gravitational interactions and short-range interactions such as inter-molecular force or fluid interaction. Application program gives the function to perform interaction calculation for two groups of particles (one group exerting forces to the other), and FDPS calculates the interaction using that function.

FDPS offers very good performance on large-scale parallel systems consisting of "homogeneous" multi-core processors, such as K computer and Cray systems based on x86 processors. On the other hand, the architecture of large-scale HPC systems is moving from homogeneous multi-core processors to accelerator-based systems and heterogeneous multi-core processors.

GPGPUs are most widely used accelerators, and are available on many large-scale systems. They offer the price-performance ratios and performance per watt numbers significantly better than those of homogeneous systems, primarily by integrating a large number of relatively simple processors on single accelerator chip. On the other hand, accelerator-based systems have two problems. One is that for many applications, the communication bandwidth between CPUs and accelerators becomes the bottleneck. The second one is that because CPUs and accelerators have separate memory spaces, the programming is complicated and we cannot use existing programs.

Though in general it is difficult to use accelerators, for particle-based simulations the efficient use of accelerators is not so difficult, and that fact is the reason why GRAPE families of accelerators specialized for gravitational N-body simulations had been successful (Makino et al. 2003). GPGPUs are also widely used both for collisional (Gaburov et al. 2009) and collisionless (Bédorf et al. 2012) gravitational N-body simulations. Thus, it is clearly desirable for FDPS to support accelerator-based architectures.

Though gravitational N-body simulation codes have achieved very good performance on large clusters of GPGPUs, to achieve high efficiency for particle systems with short-range interactions is difficult. For example, there exist many high-performance implementations of SPH algorithms on single GPGPU, or relatively small number of multiple GPGPUs (around six), but there are not

many high-performance SPH codes for large-scale parallel GPGPU systems. Practically all efficient GPGPU implementations of SPH algorithm uses GPGPU to run the entire simulation code, in order to eliminate the communication overhead of GPGPUs and CPUs. The calculation cost of particle-particle interactions dominates the total calculation cost of SPH simulations. Thus, as far as the calculation cost is concerned, it is sufficient to let GPGPUs evaluate the interactions, and let CPUs perform the rest of the calculation. However, because of relatively low communication bandwidth between CPUs and GPGPUs, we need to avoid the data transfer between them, and if we let GPGPUs do all calculations, it is clear that we can minimize the communication.

On the other hand, it is more difficult to develop programs for GPGPUs than for CPUs, and to develop MPI parallel programs for multiple GPGPUs is clearly more difficult. To make such MPI parallel program for GPGPUs run on a large cluster is close to impossible.

In order to add the support of GPGPU and other accelerators to FDPS, we decided to take a different approach. We keep the simple model in which accelerators do the interaction calculation only, and CPUs do all the rest. However, we try to minimize the communication between CPUs and accelerators as much as possible, without making the calculation on the side of accelerators very complicated.

In this paper, we describe our strategy of using accelerators, how application programmers can use FDPS to efficiently use accelerators, and achieved performance.

This paper is organized as follows. In section 2, we present the overview of FDPS. In section 3, we discuss the traditional approach of using accelerators for interaction calculation, and its limitations. In section 4 we summarize our new approach. In section 5 we show how the users can use new APIs of FDPS to make use of accelerators. In section 6 we give the result of performance measurement for GPGPU-based systems and give the performance prediction for hypothetical systems. Section 7 is for summary and discussion.

2 Overview of FDPS

The basic idea (or the ultimate goal) of FDPS is to make it possible for researchers to develop their own high-performance, highly-parallel particle-based simulation codes without spending too much time for writing, debugging, and performance tuning of the codes. In order to achieve this goal, we have designed FDPS so that it provides all necessary functions for efficient parallel program for particle-based simulations. FDPS uses MPI for inter-node parallelization and OpenMP for intra-node parallelization. In order to reduce the communication between computing nodes, the computational domain is divided using the recursive multisection algorithm (Makino 2004), but with weights for

particles to achieve the optimal load balancing (Ishiyama et al. 2009). The number of subdomains is equal to the number of MPI processes, and one subdomain is assigned to one MPI process.

Initially, particles are distributed to MPI processes in an arbitrary way. It is not necessary that the initial distribution is based on spatial decomposition, and it is even possible that initially just one process has all particles, if it has the sufficient amount of memory. After the spatial coordinates of subdomains are determined, for each particle, the MPI process to which it belongs is determined, and it is sent to that process. These parts can be achieved just by calling FDPS library functions. In order for FDPS functions to get information of particles and copy or move them, FDPS functions need to know the data structure of the particles. This is made possible by making FDPS "template-based", so that at compile time FDPS library functions know the data structure of particles.

After particles are moved to their new locations, the interaction calculation is done through parallel Barnes-Hut algorithm based on the local essential tree (Makino 2004). In this method, each MPI process first constructs the tree structure from its local particles (local tree). Then, it sends, to all other MPI process, its information necessary for that MPI process to evaluate the interaction with its particles. This necessary information is called the local essential tree (LET).

After one process received all LETs from all other nodes, it constructs the global tree by merging the LETs. In FDPS, merging is not actually done but LETs are first reduced to arrays of particles and superparticles (hereafter we call it "SPJ"), and a new tree is constructed from combined list of all particles. Here, a SPJ represents a node of Barnes-Hut tree.

Finally, the interaction calculation is done by traversing the tree for each particles. Using Barnes' vectorization algorithm (Barnes 1990), we traverse the tree for a group of local particles, and create the "interaction list" for that group. Then, FDPS calculates the interaction exerted from particles and superparticles in this interaction list to particles in the group, by calling used-supplied interaction function.

In the case of long-range interaction, we use the standard Barnes-Hut scheme for treewalk. In the case of short-range interaction such as SPH interaction between particles, we still use treewalk but with cell-opening criterion different from the standard opening angle.

Thus, users of FDPS can use the functions for domain decomposition, particle migration and interaction calculation, by passing their own particle data class and interaction calculation function to FDPS at the compile time. Interaction calculation function should be designed as receiving two arrays of particles, one exerting the "force" from to the other.

3 Traditional Approach to Use Accelerators and Its Limitation

As we have already stated in the introduction, accelerators have been used for gravitational N-body simulations, both on single and parallel machines, with and without Barnes-Hut treecode (Barnes & Hut 1986). In the case of the tree algorithm, the idea is to use Barnes' vectorization algorithm, which is what we defined as the interface between the user-defined interaction function and FDPS. Thus, in principle we can use accelerators just by replacing the user-defined interaction function with that uses the accelerators. In the case of GRAPE processors, that would be the only thing we need to do. At the same time, this would be the only thing we can do.

On modern GPGPUs, however, we need to modify the interface and algorithm slightly. There are two main reasons for this modification. The first one is that the software overhead of GPGPUs for data transfer and kernel startup is much larger than that for GRAPE processors. Another difference is in the architecture. GRAPE processors consist of relatively small number of highly pipelined, application-specific pipeline processor for interaction calculation, with hardware support for fast summation of results from multiple pipelines. On the other hand, GPGPUs consist of a very large number of programmable processors, with no hardware support for summation of the results obtained on multiple processors. Thus, to make efficient use of GPGPUs, we need to calculate interactions on a large number of particles by single call to GPGPU computing kernel. Vectorization algorithm has one adjustable parameter, n_{grp}, the number of particles which share one interaction list, and it is possible to make efficient use of GPGPUs by making this n_{grp} large. However, using excessively large n_{grp} causes the increase of the total calculation cost, and thus not desirable.

Hamada et al. (2009) introduced an efficient way to use GPGPUs which they called the "mutltiwalk" method. In their method, the CPU first constructs multiple interaction lists for multiple groups of particles, and then sends them to the GPGPU in a single kernel call. GPGPU performs the calculation of multiple interaction lists in parallel, and returns all results in a single data transfer. In this way, we can tolerate the large overhead of invoking computing kernels on GPGPUs and the lack of the support for fast summation.

Even though this multiwalk method is quite effective, there still remain rooms of improvements, and that means on modern accelerators the efficiency we can achieve with the mltiwalk method is rather limited.

The biggest remaining inefficiency comes from the fact that with the multiwalk method we send interaction lists for each particle group. One interaction list is an array of physical quantities (at least positions and masses) of particles. Typically, the number of particles in an interaction list is 10 times more than the number of particles for which that interaction list is constructed, and thus the

transfer time of the interaction list is around 10 times longer than that of the particles which receive the force. This means that we are sending same particles (and superparticles) multiple times when we send multiple interaction lists.

In the next section, we discuss how we can reduce the amount of communication and also further reduce the calculation cost for the parts other than the force calculation kernel.

4 New Algorithms

As we described in the previous section, to send all particles in the interaction list to accelerators is inefficient because we send same particles and SPJs multiple times. In section 4.1 we will describe new algorithm to overcome this inefficiency. In section 4.2, we will also describe new algorithm to further reduce the calculation cost for the parts other than the force calculation kernel. In section 4.3, we will describe the actual procedures with and without new algorithms.

4.1 Indirect addressing of particles

When we use the interaction list method on systems with accelerators, in the simplest implementation, for each group of particles and its interaction list, we send physical quantities necessary for interaction calculation, such as positions and masses in the case of gravitational force calculation. Roughly speaking, the number of particles in the interaction list is around ten times longer than that in one group. Thus, we are sending around $10n$ particles, where n is the number of particles per MPI process, at each timestep. Since there are only n local particles and the number of particles and tree nodes in LETs is generally much smaller than n, this means that we are sending the same data many times, and that we should be able to reduce the communication by sending particle and tree node data only once. Some GRAPE processors including GRAPE-2A, MDGRAPE-x and GRAPE-8, have hardware support for this indirect addressing (Makino & Daisaka 2012).

In the case of programmable accelerators, this indirect addressing can be achieved by first sending arrays of particles and tree nodes, and then sending the interaction list (here the indices of particles and tree nodes indicating the location of them in their arrays). The user-defined interaction calculation function should be modified so that it uses indirect addressing to access particles. Examples of such code is included in the current FDPS distribution (version 4.0 and later), and we plan to develop template routines which can be used to generate codes on multiple platforms from single user-supplied code for interaction calculation.

Interaction list is usually an array of 32-bit integers (four bytes), and one particle data is at least 16 bytes (when positions and masses are all in single precision numbers), but can be much larger

8

in the case of SPH and other method. Thus, with this method we can reduce the communication cost by a large factor.

One limitation of this indirect addressing method is that all particles in one process should fit in the memory of the accelerator. Most of accelerator have relatively small memories. In such cases, we can still use this method, by dividing the particles into blocks small enough to fit the memory of accelerator. For each block, we construct the "global" tree structure similar to that for all particles in the process, and interaction lists for all groups under the block.

4.2 Reuse of Interaction Lists

For both of long-range and short-range interactions, FDPS constructs the interaction lists for groups of particles. It is possible to keep using the same interaction lists for multiple timesteps, if particles do not move large distances in a single timestep. In the case of SPH or molecular dynamics simulations, it is guaranteed that particles move only a small fraction of interparticle distance in a single timestep, since the size of the timestep is limited by the stability condition. Thus, in such cases we can safely use the interaction lists for several timesteps.

Even in the case of gravitational many-body simulations, there are cases where the change of the relative distance of between particles in a single timestep is small. For example, both in the simulations of planetary formation processes or planetary rings, the random velocities of particles are very small, and thus, even though particles move large distances, there is no need to reconstruct the tree structure at each timestep, because the changes of the relative positions of particles are small.

In the case of galaxy formation simulation using Nbody+SPH technique, generally the timestep for the SPH part is much smaller than that for the gravity part, and thus we should be able to use the same tree structure and interaction lists for multiple SPH steps.

If we use this algorithm (hereafter we call it the reuse algorithm), the procedures of the interaction calculation for the step with tree construction and that without the tree construction are different. The procedure for the tree construction step is given by

1. Construct the local tree
2. Construct the LET for all other processes. These LETs should be the list of indices of particles and tree nodes, so that they can be used later.
3. Exchange LETs. Here, the physical information of tree nodes and particles should be exchanged
4. Construct the global tree
5. Construct the interaction lists
6. Perform the interaction calculation for each group using the constructed list.

The procedure for reusing steps is given by

1. Update the physical information of the local tree
2. Exchange LETs.
3. Update the physical information of the global tree
4. Perform the interaction calculation for each group using the constructed list.

In many cases we can keep using the same interaction list for around 10 timesteps. In the case of planetary ring simulation, using the same list for much larger number of timesteps is possible, because the stepsize of planetary ring simulation using "soft-sphere" method (Iwasawa et al. 2018) is limited by the hardness of the "soft" particles and thus much smaller than the usual timescale determined by the local velocity dispersion and interparticle distance.

With this reuse algorithm, we can reduce the cost of the following steps: (a) tree construction, (b) LET construction, (c) interaction list construction. The calculation costs of steps (a) and (c) are $O(N)$ and $O(N \log N)$, respectively. Thus they are rather large for simulations with large number of particles. Moreover, by reducing the cost of step (c), we can make the group size n_{grp} small, which results in the decrease of the calculation cost due to the use of interaction list. Thus, the overall improvement of the efficiency is quite significant.

The construction and maintenance of interaction lists and other necessary data structures are all done within FDPS. Therefore, user-developed application programs can use this reuse algorithm just by calling the FDPS interaction calculation function with one additional argument indicating reuse/construction. The necessary change of the application program is very small.

4.3 Procedures with or without New Algorithms

In this section, we describe the actual procedures of simulations using FDPS with or without new algorithms. Before describing the procedures, let us introduce four particle data types FDPS uses: FP (Full Particle), EPI (Essential Particle I), EPJ (Essential Particle J) and FORCE. FP is the data structure containing all information of a particle, EPI(J) is used for the minimal data of particles which receives (gives) the force, and FORCE type to store the calculated interaction. FDPS uses these additional three data types to minimize the memory access during the interaction calculation. We first describe the procedure for the calculation without the reuse algorithm and then describe that for the reuse algorithm.

At the beginning of one timestep, the computational domains assigned to MPI processes are determined and all processes exchange particles so that all particles belong to their appropriate domains. Then, the coordinates of the root cell of the tree are determined using the positions of all

particles. After the determination of the root cell, each MPI process constructs its local tree. The local tree construction consists of the following four steps.

1. Generate Morton keys for all particles.

2. Sort key-index pairs in Morton order by the radix sort.

3. Reorder FPs in Morton order referring the key-index pairs and copy the particle data from FPs to EPIs and EPJs.

4. For each level of the tree, from top to bottom, allocate tree cells and link their child cells. In each level, we use the binary search to find cell boundaries.

In the case of the reusing step, these steps are skipped.

After the construction of the local tree, multipole moments of all local tree cells are calculated, from the bottom to the top of the tree. Even at the reusing step, the calculation of the multipole moments is performed because the physical quantities of particles are updated at every timesteps.

After the calculation of the multipole moments of the local tree, each MPI process constructs LETs and send them to other MPI process. When the reusing algorithms is used, at the tree construction step, each MPI process saves the LETs and their destination processes.

After the exchange of LETs, each MPI process constructs the global tree from received LETs and its local tree. The procedure is almost the same as that for the local tree construction.

After the construction of the global tree, each MPI process calculates the multipole moments of all cells of the global tree. The procedure is the same as that for the local tree.

After the calculation of the moments of the global tree, each MPI process constructs the interaction lists and using them performs the force calculation. If we do not use the multiwalk method, each MPI process makes the interaction lists for one particle group and then the user-defined force kernel calculates the forces from EPJs and SPJs in the interaction list to EPIs in the particle group.

When we use the multiwalk method, each MPI process makes multiple interaction lists for multiple particle groups. When the indirect addressing method is not used, each MPI process gives multiple groups and multiple interaction lists to the interaction kernel on the accelerator. Thus we can summarize the procedure of the force calculation without the indirect addressing method for multiple particle groups as follows:

1. Construct the interaction list for multiple particle groups.

2. Copy EPIs and the interaction lists to the send buffer for the accelerator. Here, the interaction list consists of EPJs and SPJs.

3. Send particle groups and their interaction lists to the accelerator.

4. Let the accelerator calculate interactions on particle groups sent at step 3.

5. Receive the results calculated at step 4 and copy them back to FPs, integrate orbits of FPs and copy the data from FPs to EPIs and EPJs.

To calculate forces on all particles, the above steps are repeated until all particle groups are processed. Note that the construction of the interaction list (step 1), sending the data to the accelerator (step 3), actual calculation (step 4) and receiving the calculated result (step 5) can all be overlapped.

On the other hand, when the indirect addressing method is used, before the construction of the interaction lists, each MPI process sends the data of all cells of the global tree to the accelerator. Thus at the beginning of the interaction calculation, it should send them to the accelerator. After that, the accelerator receives the data of particle groups and their interaction lists but here the interaction list contains the indices of EPJs and SPJs and not their physical quantities. Thus, the calculation procedure with indirect addressing method is the same as that without the indirect addressing except that all data of the global tree are sent at the beginning of the calculation and the interaction lists sent during the calculation contains only indices of tree cells and EPJs.

Both with and without the indirect addressing method, we can use the reusing method. For the construction step, the procedures are the same. For the reusing steps, we can skip the steps for the interaction-list construction (step 1). When we use the indirect addressing method, we can also skip the sending of them since the lists of indices are unchanged during the reuse.

5 APIs to use Accelerators

In this section, we describe the APIs (application program interfaces) of FDPS to use accelerators and how to use them by showing sample codes developed for NVIDIA GPGPUs. Part of the user kernel is written in CUDA.

FDPS has high level APIs to perform all procedures for interaction calculation in single API call. For the multiwalk method, FDPS provides `calcForceAllAndWriteBackMultiWalk` or `calcForceAllAndWriteBackMultiWalkIndex`. The difference between these two functions is that the former dose not use the indirect addressing method. These two APIs can be used as the replacement of `calcForceAllAndWriteBack`, which is another top level API provided by FDPS distribution version 1.0 or later. A user must provide two force kernels: the "dispatch" and "retrieve" kernels. The "dispatch" kernel is used to send EPIs, EPJs and SPJs to accelerators and call the force kernel. The "retrieve" kernel is used to collect FORCEs from accelerators. The reason why FDPS needs two kernels is to allow the overlap of the calculation on the CPU with the force calculation on the accelerator as we described in the previous section.

The reusing method can be used with all of three top level APIs described above. The only

```
1    int main(int argc, char *argv[]) {
2
3      // initialize FDPS (omitted)
4
5      PS::ParticleSystem<FP> system;
6      PS::DomainInfo dinfo;
7      PS::TreeForForceLong<FORCE, EPI, EPJ>::Monopole tree;
8
9      // set initial condition (omitted)
10
11     PS::S32 n_loop =0;
12     while(time_sys < time_end){
13       PS::INTERACTION_LIST_MODE int_mode int_mode = PS::REUSE_LIST;
14       if(n_loop % 8 == 0){
15         dinfo.decomposeDomainAll(system);
16         system.exchangeParticle(dinfo);
17         int_mode = PS::MAKE_LIST_FOR_REUSE;
18       }
19
20       tree.calcForceAllAndWriteBackMultiWalkIndex(DispatchKernelIndex,
21                                                   RetrieveKernel,
22                                                   tag_max,
23                                                   system,
24                                                   dinfo,
25                                                   n_walk_limit,
26                                                   true,
27                                                   int_mode);
28
29       // integrate orbits of particles (omitted)
30
31       n_loop++;
32     }
33     return 0;
34   }
```

Fig. 1. Example of how to use the reusing method.

thing users do to use the reusing method is to give an appropriate FDPS-provided enum-type value to these functions so that the reusing method is enabled. The enum-type values FDPS provided are MAKE_LIST, MAKE_LIST_FOR_REUSE and REUSE_LIST. At the construction step the application program should give MAKE_LIST_FOR_REUSE to the top level APIs so that FDPS constructs the trees and the interaction lists and saves them. At the reusing step, the application program should give REUSE_LIST so that FDPS skips the construction of the trees and reuses the interaction lists constructed at the last construction step. In the case of MAKE_LIST, FDPS also constructs the trees and the interaction lists but dose not save them. Thus the users cannot use the reusing method. Figure 1 shows an example of how to use the reusing method. In this example, the trees and the interaction lists are constructed once in every eight steps. While the same list is being reused, particles should remain in the same MPI process as at the moment of the list construction. Thus exchangeParticle should be called only just before the tree construction step.

Figure 2 shows an example of the dispatch kernel without the indirect addressing method. FDPS gives the dispatch kernel the arrays of the pointers of EPIs, EPJs and SPJs as the arguments of the kernel (lines 3, 5 and 7). Each pointers point to the address of the first elements of the arrays of EPIs, EPJs and SPJs for one group and its interaction list. The sizes of these arrays are given by n_epi (line 4), n_epj (line 6) and n_spj (line 8). FDPS also gives "tag" (the first argument) and "n_walk" (the second argument). The argument "tag" is used to specify individual accelerators if multiple accelerators are available. However, in the current version of FDPS, "tag" is disabled and

FDPS always gives it 0. The argument "n_walk" is the number of the particle groups and interaction lists.

To overlap the actual force calculation on the GPGPU with the data transfer between the GPGPU and the CPU, we use CUDA stream, which is a sequence of operations executed on the GPGPU. In this example, we used N_STREAM CUDA streams. In this paper, we used 8 CUDA streams because even if we use more CUDA streams, the performance of our simulations is not improved. In each stream, n_walk/N_STREAM interaction lists are handled. The particle data types, EPIs (lines 28–33), EPJs (lines 36–41) and SPJs (lines 48–48) are copied to the send buffers for GPGPUs. Here, we use the same buffer for EPJs and SPJs because the types of the EPJ and SPJ are the same. In lines 55 and 56, the EPIs and EPJs are sent to the GPGPU. Then the force kernel is called in line 60.

Figure 3 shows an example of the dispatch kernel with the indirect addressing method. This kernel is almost the same as that without the indirect addressing method except for two differences. One difference is that at the beginning of one timestep, all data of the global tree (EPJs and SPJs) are sent to GPGPU (lines 15 and 16). Whether or not the application program sends EPJs and SPJs to GPGPU is specified by the 13th argument "send_flag". If "send_flag" is true, the application program sends all EPJs and SPJs. Another difference is that indices of EPJs and SPJs are sent (lines 48–58 and 67–69) instead of physical quantities of EPJs and SPJs. Here, we use user-defined global variable CONSTRUCTION_STEP to specify whether the current step is construction or reusing steps. At the construction step, CONSTRUCTION_STEP becomes unity and the user program sends the interaction list to the GPGPU and saves them in the GPGPU. On the other hand, at the reusing step, the user program dose not send the list and reuse the interaction list saved in the GPGPU.

Figure 4 shows an example of the retrieve kernel. The same retrieve kernel can be used with and without the indirect addressing method. In line 12, the GPGPU sends the interaction results to the receive buffer of the host. To let the CPU wait until all functions in the same stream on the GPGPU are completed, cudaStreamSynchronize is called in line 13. Finally, the interaction results are copied to FORCEs.

6 Performance

In this section, we present the measured performance and the performance model of a simple N-body simulation code implemented using FDPS on CPU-only and CPU+GPGPU systems.

```
1   PS::S32 DispatchKernelStream(const PS::S32 tag,
2                                const PS::S32 n_walk,
3                                const EPI *epi[],
4                                const PS::S32 n_epi[],
5                                const EPJ *epj[],
6                                const PS::S32 n_epj[],
7                                const PS::SPJMonopole *spj[],
8                                const PS::S32 n_spj[]){
9     const int n_walk_ave = n_walk/N_STREAM;
10    for(int id_stream=0; id_stream<N_STREAM; id_stream++){
11      const int n_walk_in_stream = n_walk_ave + ((id_stream
      ↪ < n_walk%N_STREAM) ? 1 : 0);
12      const int id_walk_head = n_walk_ave*id_stream +
      ↪ std::min(id_stream, n_walk%N_STREAM);
13      const int id_walk_end  = id_walk_head +
      ↪ n_walk_in_stream;
14      ij_disp_h[id_stream][0].x = ij_disp_h[id_stream][0].y
      ↪ = 0;
15      for(int iw=0; iw<n_walk_in_stream; iw++){
16        const int iw_src = iw+id_walk_head;
17        ij_disp_h[id_stream][iw+1].x =
        ↪ ij_disp_h[id_stream][iw].x + n_epi[iw_src];
18        ij_disp_h[id_stream][iw+1].y =
        ↪ ij_disp_h[id_stream][iw].y +
        ↪ (n_epj[iw_src]+n_spj[iw_src]);
19      }
20      ij_disp_h[id_stream][n_walk_in_stream+1] =
      ↪ ij_disp_h[id_stream][n_walk_in_stream];
21      cudaMemcpyAsync(ij_disp_d[id_stream],
      ↪ ij_disp_h[id_stream],
      ↪ (n_walk_in_stream+2)*sizeof(int),
      ↪ cudaMemcpyHostToDevice, stream[id_stream]);
22      int ni_tot_reg =
      ↪ ij_disp_h[id_stream][n_walk_in_stream].x;
23      ni_tot_reg = ((ni_tot_reg-1)/N_THREAD_GPU +
      ↪ 1)*N_THREAD_GPU;
24      for(int iw=id_walk_head; iw<id_walk_end; iw++){
25        const int iw_tmp = iw - id_walk_head;
26        const int n_epi_tmp = n_epi[iw];
27        int i_dst = ij_disp_h[id_stream][iw_tmp].x;
28        for(int i=0; i<n_epi_tmp; i++, i_dst++){
29          epi_h[id_stream][i_dst].pos.x   =
          ↪ epi[iw][i].pos.x;
30          epi_h[id_stream][i_dst].pos.y   =
          ↪ epi[iw][i].pos.y;
31          epi_h[id_stream][i_dst].pos.z   =
          ↪ epi[iw][i].pos.z;
32          epi_h[id_stream][i_dst].id_walk = iw_tmp;
33        }
34        int j_dst = ij_disp_h[id_stream][iw_tmp].y;
35        const int n_epj_tmp = n_epj[iw];
36        for(int j=0; j<n_epj_tmp; j++, j_dst++){
37          epj_h[id_stream][j_dst].posm.x = epj[iw][j].pos.x;
38          epj_h[id_stream][j_dst].posm.y = epj[iw][j].pos.y;
39          epj_h[id_stream][j_dst].posm.z = epj[iw][j].pos.z;
40          epj_h[id_stream][j_dst].posm.w = epj[iw][j].mass;
41        }
42        const int n_spj_tmp = n_spj[iw];
43        for(int j=0; j<n_spj_tmp; j++, j_dst++){
44          epj_h[id_stream][j_dst].posm.x = spj[iw][j].pos.x;
45          epj_h[id_stream][j_dst].posm.y = spj[iw][j].pos.y;
46          epj_h[id_stream][j_dst].posm.z = spj[iw][j].pos.z;
47          epj_h[id_stream][j_dst].posm.w = spj[iw][j].mass;
48        }
49      }
50      const int ni_tot =
      ↪ ij_disp_h[id_stream][n_walk_in_stream].x;
51      const int nj_tot =
      ↪ ij_disp_h[id_stream][n_walk_in_stream].y;
52      for(int i=ni_tot; i<ni_tot_reg; i++){
53        epi_h[id_stream][i].id_walk = n_walk_in_stream;
54      }
55      cudaMemcpyAsync(epi_d[id_stream], epi_h[id_stream],
      ↪ ni_tot_reg*sizeof(EPI_GPU),
      ↪ cudaMemcpyHostToDevice, stream[id_stream]);
56      cudaMemcpyAsync(epj_d[id_stream], epj_h[id_stream],
      ↪ nj_tot*sizeof(EPJ_GPU), cudaMemcpyHostToDevice,
      ↪ stream[id_stream]);
57      int nblocks  = ni_tot_reg / N_THREAD_GPU;
58      int nthreads = N_THREAD_GPU;
59      const float eps2 = FPGrav::eps * FPGrav::eps;
60      ForceKernel <<<nblocks, nthreads, 0,
      ↪ stream[id_stream]>>> (ij_disp_d[id_stream],
      ↪ epi_d[id_stream], epj_d[id_stream],
      ↪ force_d[id_stream], eps2);
61    }
62    return 0;
63  }
```

Fig. 2. Example of the dispatch Kernel without the indirect addressing method.

```
1    PS::S32 DispatchKernelIndexStream2(const PS::S32 tag,
2                                       const PS::S32 n_walk,
3                                       const EPI *epi[],
4                                       const PS::S32 *n_epi,
5                                       const PS::S32
                                        ↪ *id_epj[],
6                                       const PS::S32 *n_epj,
7                                       const PS::S32
                                        ↪ *id_spj[],
8                                       const PS::S32 *n_spj,
9                                       const EPJ *epj,
10                                      const PS::S32
                                        ↪ n_epj_tot,
11                                      const PS::SPJMonopole
                                        ↪ *spj,
12                                      const PS::S32
                                        ↪ n_spj_tot,
13                                      const bool send_flag){
14   if(send_flag==true){
15     H2dJptcl(epj, n_epj_tot, epj_h, epj_d);
16     H2dJptcl(spj, n_spj_tot, epj_h, epj_d, n_epj_tot);
17     for(int i=0; i<N_STREAM; i++){
18       ID_EPJ_GLB_OFFSET[i] = 0;
19     }
20     return 0;
21   }
22   const int n_walk_ave = n_walk/N_STREAM;
23   for(int id_stream=0; id_stream<N_STREAM; id_stream++){
24     const int n_walk_in_stream = n_walk_ave + ((id_stream
       ↪ < n_walk%N_STREAM) ? 1 : 0);
25     const int id_walk_head = n_walk_ave*id_stream +
       ↪ std::min(id_stream, n_walk%N_STREAM);
26     ij_disp_h[id_stream][0].x = 0;
27     ij_disp_h[id_stream][0].y =
       ↪ ID_EPJ_GLB_OFFSET[id_stream];
28     for(int iw=0; iw<n_walk_in_stream; iw++){
29       const int iw_src = iw + id_walk_head;
30       ij_disp_h[id_stream][iw+1].x =
         ↪ ij_disp_h[id_stream][iw].x + n_epi[iw_src];
31       ij_disp_h[id_stream][iw+1].y =
         ↪ ij_disp_h[id_stream][iw].y +
         ↪ (n_epj[iw_src]+n_spj[iw_src]);
32     }
33     ij_disp_h[id_stream][n_walk_in_stream+1] =
       ↪ ij_disp_h[id_stream][n_walk_in_stream];
34     ij_disp_h[id_stream].htod(n_walk_in_stream+2,
       ↪ stream[id_stream]);
35     cudaMemcpyAsync(ij_disp_d[id_stream],
       ↪ ij_disp_h[id_stream],
       ↪ (n_walk_in_stream+2)*sizeof(int),
       ↪ cudaMemcpyHostToDevice, stream[id_stream]);
36     int ni_tot_reg =
       ↪ ij_disp_h[id_stream][n_walk_in_stream].x;
37     ni_tot_reg = ((ni_tot_reg-1)/N_THREAD_GPU +
       ↪ 1)*N_THREAD_GPU;
38     for(int iw=0; iw<n_walk_in_stream; iw++){
39       const int iw_src = id_walk_head + iw;
40       const int n_epi_tmp = n_epi[iw_src];
41       int i_dst = ij_disp_h[id_stream][iw].x;
42       for(int ip=0; ip<n_epi_tmp; ip++, i_dst++){
43         epi_h[id_stream][i_dst].pos.x   =
           ↪ epi[iw_src][ip].pos.x;
44         epi_h[id_stream][i_dst].pos.y   =
           ↪ epi[iw_src][ip].pos.y;
45         epi_h[id_stream][i_dst].pos.z   =
           ↪ epi[iw_src][ip].pos.z;
46         epi_h[id_stream][i_dst].id_walk = iw;
47       }
48       if(CONSTRUCTION_STEP==1){
49         int j_dst = ij_disp_h[id_stream][iw].y -
           ↪ ID_EPJ_GLB_OFFSET[id_stream];
50         const int n_epj_tmp = n_epj[iw_src];
51         for(int j=0; j<n_epj_tmp; j++, j_dst++){
52           id_epj_h[id_stream][j_dst] = id_epj[iw_src][j];
53         }
54         const int n_spj_tmp = n_spj[iw_src];
55         for(int j=0; j<n_spj_tmp; j++, j_dst++){
56           id_epj_h[id_stream][j_dst] =
             ↪ id_spj[iw_src][j]+n_epj_tot;
57         }
58       }
59     }
60     const int ni_tot =
       ↪ ij_disp_h[id_stream][n_walk_in_stream].x;
61     const int nj_tot =
       ↪ ij_disp_h[id_stream][n_walk_in_stream].y -
       ↪ ID_EPJ_GLB_OFFSET[id_stream];
62     for(int i=ni_tot; i<ni_tot_reg; i++){
63       epi_h[id_stream][i].id_walk = n_walk_in_stream;
64     }
65     dev_epi[id_stream].htod(ni_tot_reg,
       ↪ stream[id_stream]);
66     cudaMemcpyAsync(epi_d[id_stream], epi_h[id_stream],
       ↪ ni_tot_reg*sizeof(EPI_GPU),
       ↪ cudaMemcpyHostToDevice, stream[id_stream]);
67     if(CONSTRUCTION_STEP == 1){
68       cudaMemcpyAsync(
         ↪ id_epj_d[id_stream]+ID_EPJ_GLB_OFFSET[id_stream],
         ↪ id_epj_h[id_stream], nj_tot*sizeof(int),
         ↪ cudaMemcpyHostToDevice, stream[id_stream]);
69     }
70     ID_EPJ_GLB_OFFSET[id_stream] += nj_tot;
71     int nblocks  = ni_tot_reg / N_THREAD_GPU;
72     int nthreads = N_THREAD_GPU;
73     const float eps2 = FPGrav::eps * FPGrav::eps;
74     ForceKernelIndex <<<nblocks, nthreads, 0,
       ↪ stream[id_stream]>>> (ij_disp_d[id_stream],
       ↪ epi_d[id_stream], epj_d, id_epj_d[id_stream],
       ↪ force_d[id_stream], eps2);
75   }
76   return 0;
77 }
```

Fig. 3. Example of the dispatch Kernel with the indirect addressing method. Boxes with solid line indicate the differences from the kernel without indirect addressing method.

```
1    PS::S32 RetrieveKernel(const PS::S32 tag,
2                           const PS::S32 n_walk,
3                           const PS::S32 ni[],
4                           FORCE *force[]){
5      static int n_offset[N_WALK_LIMIT+1];
6      const int n_walk_ave = n_walk/N_STREAM;
7      for(int id_stream=0; id_stream<N_STREAM; id_stream++){
8        const int n_walk_in_stream  = n_walk_ave   +
     ↪  ((id_stream < n_walk%N_STREAM) ? 1 : 0);
9        const int id_walk_head = n_walk_ave*(id_stream) +
     ↪  std::min(id_stream, n_walk%N_STREAM);
10       const int id_walk_end  = id_walk_head +
     ↪  n_walk_in_stream;
11       const int ni_tot =
     ↪  ij_disp[id_stream][n_walk_in_stream].x;
12       cudaMemcpyAsync(force_h[id_stream],
     ↪  force_d[id_stream], ni_tot*sizeof(FORCE_GPU),
     ↪  cudaMemcpyDeviceToHost, stream[id_stream]);
13       cudaStreamSynchronize(stream[id_stream]);
14       n_offset[0] = 0;
15       for(int i=0; i<n_walk_in_stream; i++){
16         const int iw = id_walk_head + i;
17         n_offset[i+1] = n_offset[i] + ni[iw];
18       }
19       for(int iw=id_walk_head; iw<id_walk_end; iw++){
20         const int ni_tmp = ni[iw];
21         const int n_offset_tmp = n_offset[iw-id_walk_head];
22         for(int i=0; i<ni_tmp; i++){
23           const int i_src = i + n_offset_tmp;
24           force[iw][i].acc.x =
     ↪  force_h[id_stream][i_src].accp.x;
25           force[iw][i].acc.y =
     ↪  force_h[id_stream][i_src].accp.y;
26           force[iw][i].acc.z =
     ↪  force_h[id_stream][i_src].accp.z;
27           force[iw][i].pot   =
     ↪  force_h[id_stream][i_src].accp.w;
28         }
29       }
30     }
31     return 0;
32   }
```

Fig. 4. Example of the retrieve kernel.

6.1 Measured Performance

To measure the performance of FDPS, we performed simple gravitational N-body simulations both with and without the accelerator. The initial model is a cold uniform sphere. This system will collapse in a self-similar way. Thus we can use the reusing method. The number of particles (per process) n is 2^{22} (4M). The opening criterion of the tree, θ, is between 0.25 and 1.0, the maximum number of particles in a leaf cell is 16 and the maximum number of particles in EPI group, n_{grp}, is between 64 and 16384. We performed these simulations with three different methods. We listed all methods in table 1. In the case of the reusing method, the number of reusing steps between the construction steps n_{reuse} is between 2 and 16.

We used NVIDIA TITAN V as an accelerator. Its peak performance is 13.8 Tflops for single precision calculation. Host CPU is single Intel Xeon E5-2670 v3 with the peak speed of 883 Gflops for single precision calculation. The GPGPU is connected to the host CPU through PCI Express 3.0 bus with 16 lanes. The main memory of the host computer is DDR4-2133. The theoretical peak

Method	System	Multiwalk	Indirect Addressing	Reusing
C1	CPU	No	No	No
G1	CPU+GPGPU	Yes	No	No
G2	CPU+GPGPU	Yes	Yes	Yes

Table 1. All methods we used to perform simulations.

bandwidth is 68 GB/s for the host main memory and 15.75 GB/s for the data transfer between the host and GPGPU. All data of particles are in double precision. Force calculation on GPGPU and data transfer between the CPU and GPGPU are performed in single precision.

Figure 5 shows the average elapsed time per step for methods C1, G1 and G2 with the reuse interval n_{reuse} of 2, 4 and 16 plotted against the average number of particles which share one interaction list $\langle n_{\text{i}} \rangle$. The opening angle is $\theta = 0.5$. We also plot the elapsed times for method G2 at construction step and at reusing step. The difference between the elapsed time for method G1 and that for G2 at the construction step is due to the difference in the use of the indirect addressing method.

We can see that, in the case of method G2, the elapsed time becomes smaller as reuse interval n_{reuse} increases, and approaches to the time of the reusing step. The minimum time of method G2 at the reusing step is ten times smaller than that of method C1 and four times smaller than that of method G1. The performance of method G2 with n_{reuse} of 16 is 3.7 Tflops (27 % of the theoretical peak).

We can also see that the optimal values of $\langle n_{\text{i}} \rangle$ becomes smaller as n_{reuse} increases. When we do not use the reuse method, the tree construction and traversal are done at every step. Thus, their costs are large, and to make it small we should increase $\langle n_{\text{i}} \rangle$. In order to do so, we need to use large n_{crit}, which is the maximum number of particles in the particle group. If we make $\langle n_{\text{i}} \rangle$ too large, the calculation cost increases (Makino 1991). Thus there is an optimal $\langle n_{\text{i}} \rangle$. When we use the reuse method, the relative cost of tree traversal becomes smaller. Thus, the optimal $\langle n_{\text{i}} \rangle$ becomes smaller and the calculation cost also becomes smaller. We will give more detailed analysis in section 6.2.

Table 2 shows the breakdown of the calculation time for different methods in the case of θ of 0.5. For the runs with C1 and G1, we show the breakdown at the optimal values of $\langle n_{\text{i}} \rangle$, which are 230 and 1500, respectively. For the run with G2, we show the breakdowns of the construction and the reusing steps for $\langle n_{\text{i}} \rangle$ of 230. We can see that the calculation time for reusing step is four times smaller than that for construction step. Thus if $n_{\text{reuse}} \gg 4$, the contribution of the construction step to the total calculation time is small.

Figure 6 shows the average elapsed time at optimal $\langle n_{\text{i}} \rangle$ plotted against θ for methods C1, G1

method	C1	G1	G2 (construction step)	G2 (reusing step)
$\langle n_{\mathrm{i}} \rangle$	230	1500	230	230
set root cell	0.0064	0.0066	0.0066	-
make local tree	0.091	0.092	0.095	-
calculate key	0.0084	0.0085	0.0093	-
sort key	0.042	0.043	0.044	-
reorder ptcl	0.030	0.030	0.030	-
link tree cell	0.011	0.010	0.011	-
calculate multipole moment of local tree	0.0053	0.0058	0.0053	0.0062
make global tree	0.094	0.094	0.096	0.0071
calculate key	0.0	0.0	0.0	-
sort key	0.040	0.041	0.042	-
reorder ptcl	0.036	0.036	0.037	0.0071
link tree cell	0.011	0.011	0.011	-
calculate multipole moment of global tree	0.0066	0.0064	0.0065	0.0068
calculate force	0.76	0.15	0.21	0.072
make interaction list (EPJ and SPJ)	(0.16)	0.063	-	-
make interaction list (id)	-	-	0.13	-
copy all particles and tree cells	-	-	(0.0065)	(0.0065)
copy EPI	-	(0.0037)	(0.0037)	(0.0037)
copy interaction list (EPJ and SPJ)	-	(0.020)	-	-
copy interaction list (id)	-	-	(0.013)	-
copy FORCES	-	(0.0060)	(0.0060)	(0.0060)
force kernel	(0.43)	(0.11)	(0.043)	(0.043)
H2D all particles and tree cells	-	-	(0.0073)	(0.0073)
H2D EPI	-	(0.0042)	(0.0042)	(0.0042)
H2D interaction list (EPJ and SPJ)	-	(0.034)	-	-
H2D interaction list (id)	-	-	(0.022)	-
D2H FORCE	-	(0.0067)	(0.0067)	(0.0067)
writ back + integration (+ copy ptcl)	0.015	0.016	0.021	0.025
total	0.98	0.36	0.42	0.095

Table 2. Breakdown of the total time per step for the runs with various methods in the case of θ of 0.5. The second row shows $\langle n_{\mathrm{i}} \rangle$ we used in this measurement. The times in parentheses are the estimated times by using the performance model we will discuss in the following sections, because the calculations corresponding to these times are hidden by other calculation and we can not measure them.
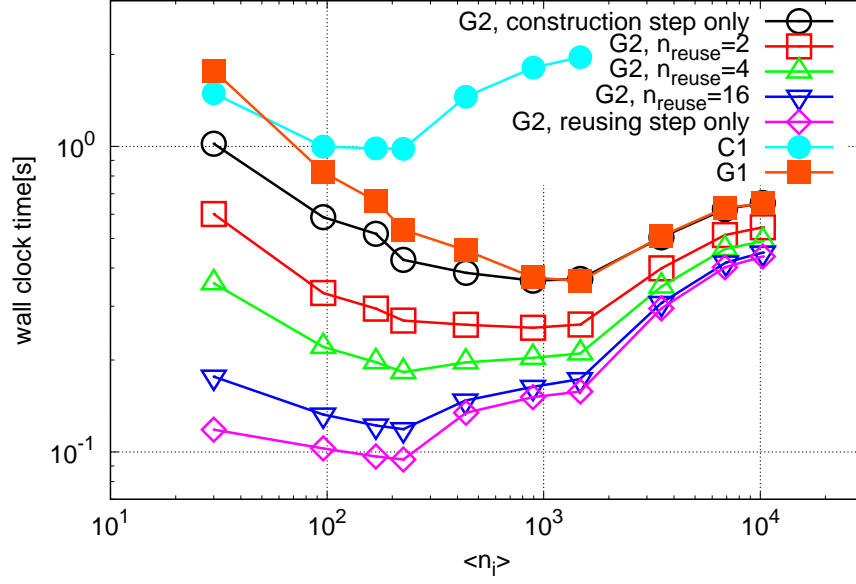
Fig. 5. Averaged elapsed time per step against $\langle n_i \rangle$. Open symbols indicate the results of the runs with method G2. Open squares, open upward triangles and open inverted triangles indicate the results with n_{reuse} of 2, 4 and 16, respectively. Open circles and open diamonds indicate the elapsed times of the construction step and the reusing step, respectively. Filled circles and filled squares indicate the elapsed times for methods C1 and G1, respectively.
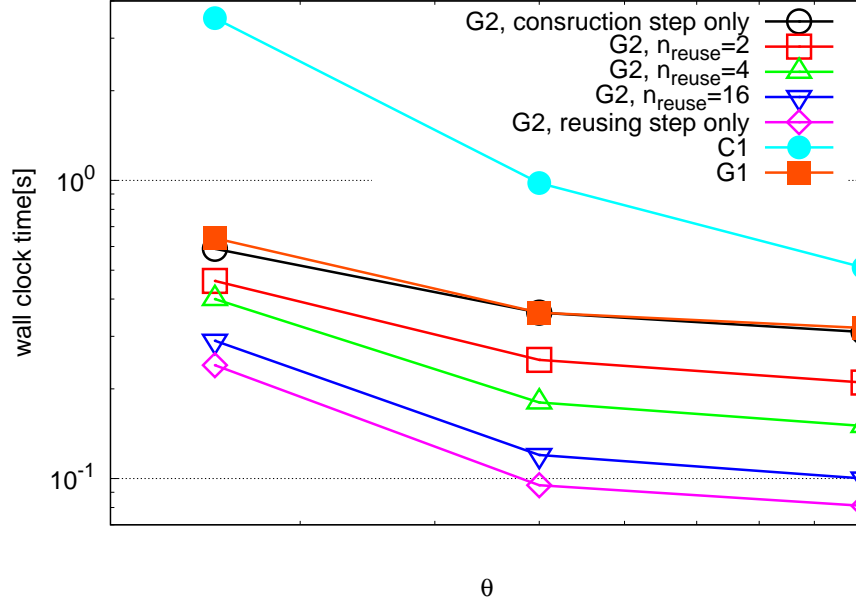
Fig. 6. Total times for various methods at the optimal $\langle n_i \rangle$ against θ. Symbols are the same as in figure 5.

and G2 with n_{reuse} of 2, 4 and 16. We also plot the elapsed times for construction and reusing steps of method G2. The slope for method C1 is steeper than those for other methods. This is because the time for the force kernel dominates the total time in the case of method C1 and it strongly depends on θ.

In the following, we present the performance model of the N-body simulation with FDPS with the monopole approximation on a single node with and without accelerator. The total execution times per step on single node for construction step $T_{\text{step,const}}$ and for reusing step $T_{\text{step,reuse}}$ are given by

$$T_{\text{step,const}} \sim T_{\text{root}} + T_{\text{const lt}} + T_{\text{mom lt}} + T_{\text{const gt}} + T_{\text{mom gt}} + T_{\text{force,const}}, \tag{1}$$

$$T_{\text{step,reuse}} \sim T_{\text{mom lt}} + T^{\text{reuse}}_{\text{reorder gt}} + T_{\text{mom gt}} + T_{\text{force,reuse}}, \tag{2}$$

where T_{root}, $T_{\text{const lt}}$, $T_{\text{mom lt}}$, $T_{\text{const gt}}$, $T_{\text{mom gt}}$, $T_{\text{force,const}}$, $T^{\text{reuse}}_{\text{reorder gt}}$ and $T_{\text{force,reuse}}$ are the times for the determination of the root cell of the tree, the construction of the local tree, the calculation of the multipole moments of the cells of the local tree, the construction of the global tree, the calculation of the multipole moments of the cells of the global tree, the force calculation for the construction step, the reorder of the particles for the global tree and the force calculation for the reusing step. The force calculation times $T_{\text{force,const}}$ and $T_{\text{force,reuse}}$ include the times for the construction of the interaction list, the actual calculation of the interactions, the copy of the interaction results from FORCEs to FPs, the integration of orbits of FPs and the copy of the particle data from FPs to EPIs and EPJs. In the reuse step, the tree is reused and therefore T_{root}, $T_{\text{const lt}}$ and $T_{\text{const gt}}$ do not appear in $T_{\text{step,reuse}}$. On the other hand, $T^{\text{reuse}}_{\text{reorder gt}}$ appears in $T_{\text{step,reuse}}$. This is because in the reusing step the particles are sorted in Morton order of the local tree and the particles should be reordered in Morton order of the global tree. In the following subsections, we will discuss the elapsed times of each component in equations (1) and (2).

6.2.1 Model of T_{root}

At the beginning of the construction step, the coordinate of the root cell of the tree is determined. In order to do so, CPU cores read the data of all particles (FPs) from the main memory, and on modern machines the effective main memory bandwidth determines the calculation time. The actual computation of determination of the root cell coordinate is much faster compared to the main memory access. Thus T_{root} is given by

$$T_{\text{root}} \sim \frac{n\alpha_{\text{root}} b_{\text{FP}}}{B_{\text{host}}}, \tag{3}$$

where n is the number of local particles, α_{root} is the coefficient for the memory access speed, b_{FP} is the data size of FP and B_{host} is the effective bandwidth of the main memory of the host which was measured by STREAM benchmark. Note that we used the "copy" kernel of STREAM benchmark. In other words, the bandwidth B_{host} indicates the mean bandwidth of reading and writing. On our system, the bandwidth of reading is slightly faster than that of writing. Thus for reading-dominant procedure, α would be smaller than unity. For our N-body code, we found $\alpha_{\text{root}} \sim 0.7$. The coefficients for

Table 3. Coefficients for equation (3)

α_{root}	0.70
b_{FP}	88 byte

equation (3) are listed in table 3.

6.2.2 Model of $T_{\mathrm{const\,lt}}$

The time for the construction of the local tree $T_{\mathrm{const\,lt}}$ is given by

$$T_{\mathrm{const\,lt}} \sim T_{\mathrm{key\,lt}} + T_{\mathrm{sort\,lt}} + T_{\mathrm{reorder\,lt}} + T_{\mathrm{link\,lt}}, \tag{4}$$

where $T_{\mathrm{key\,lt}}$, $T_{\mathrm{sort\,lt}}$, $T_{\mathrm{reorder\,lt}}$ and $T_{\mathrm{link\,lt}}$ are the elapsed times for the calculation of Morton keys, sorting of the key-index pairs, reordering of FPs, EPIs and EPJs by using the sorted key-index pairs and linking of tree cells.

The time for key construction is determined by the time to read FPs and write key-index pairs. Thus $T_{\mathrm{key\,lt}}$ is given by

$$T_{\mathrm{key\,lt}} \sim \frac{n\alpha_{\mathrm{key}}\left(b_{\mathrm{FP}} + b_{\mathrm{key}}\right)}{B_{\mathrm{host}}}, \tag{5}$$

where b_{key} is the data size of the key-index pair.

For sorting, we use the radix sort (Knuth 1997) with the chunk size of eight bits for the keys with 64-bit length. Thus we need to apply the basic procedure of the radix sort eight times. For each chunk, the data are read twice and written once. Thus the total number of memory access is 24. Therefore $T_{\mathrm{sort\,lt}}$ is given by

$$T_{\mathrm{sort\,lt}} \sim n\frac{\alpha_{\mathrm{sort}}24b_{\mathrm{key}}}{B_{\mathrm{host}}}. \tag{6}$$

For reordering, the key-index pair and FP are read once and FP, EPI and EPJ are written and

$$T_{\mathrm{reorder\,lt}} \sim n\frac{\alpha_{\mathrm{reorder\,lt}}\left(2b_{\mathrm{FP}} + b_{\mathrm{EPI}} + b_{\mathrm{EPJ}} + b_{\mathrm{key}}\right)}{B_{\mathrm{host}}}, \tag{7}$$

where the size parameters b_{EPI} and b_{EPJ} are those of the EPI and EPJ, respectively.

In the linking part, we generate the tree structure from the sorted keys. In order to do so, for each cell of the tree in each level, we determine the location of the first particle by the binary search method. Thus the cost is proportional to $n_{\mathrm{cell}}\log_2\left(n_{\mathrm{cell}}\right)$ where n_{cell} is the total number of tree cells. For usual Barnes-Hut tree, we have $n_{\mathrm{cell}} \sim n/4$. Thus $T_{\mathrm{link\,lt}}$ is given by

$$T_{\mathrm{link\,lt}} \sim \frac{n}{4}\log_2\left(\frac{n}{4}\right)\frac{\alpha_{\mathrm{link}}b_{\mathrm{key}}}{B_{\mathrm{host}}}. \tag{8}$$

The size parameters b and the memory access efficiency parameters α in equations (5) to (8) are listed in table 4. The reason why $\alpha_{\mathrm{link\,lt}}$ is much larger than unity is that in the binary search the

Table 4. Coefficient for equations (5) to (8)

α_{key}	0.85
α_{sort}	1.1
$\alpha_{\text{reorder lt}}$	1.1
α_{link}	3.4
b_{key}	16 byte
b_{EPI}	24 byte
b_{EPJ}	32 byte

Table 5. Coefficient for equation (9)

α_{mom}	1.8

address to access depends on the data just read.

6.2.3 Model of $T_{\text{mom lt}}$

The time for the calculation of the multipole moments of the local tree is determined by the time to read EPJ and therefore $T_{\text{mom lt}}$ is given by

$$T_{\text{mom lt}} \sim n \frac{\alpha_{\text{mom}} b_{\text{EPJ}}}{B_{\text{host}}}. \tag{9}$$

The coefficients for equation (9) are summarized in table 5.

6.2.4 Model of $T_{\text{const gt}}$

In the current implementation of FDPS, even if MPI is not used, the global tree is constructed. The procedures of the construction of the global tree is essentially the same as those of the local tree, except for reordering particles. In reordering particles, EPJ and SPJ in all LETs are first written to separate arrays. The indices of the arrays for EPJ and SPJ are also saved in order to efficiently reorder the particles at the reusing step. Thus $T_{\text{const gt}}$ is given by

$$T_{\text{const gt}} \sim T_{\text{key gt}} + T_{\text{sort gt}} + T_{\text{reorder gt}}^{\text{const}} + T_{\text{link gt}} \tag{10}$$

$$T_{\text{key gt}} \sim \frac{n_{\text{LET}} \alpha_{\text{key}} \left(b_{\text{key}} + b_{\text{EPJ}} \right)}{B_{\text{host}}}, \tag{11}$$

$$T_{\text{sort gt}} \sim \frac{\left(n + n_{\text{LET}} \right) \alpha_{\text{sort}} b_{\text{key}}}{B_{\text{host}}}, \tag{12}$$

$$T_{\text{reorder gt}}^{\text{const}} \sim \frac{\left(n + n_{\text{LET}} \right) \alpha_{\text{reorder gt}}^{\text{const}} \left(2 b_{\text{EPJ}} + b_{\text{index}} \right)}{B_{\text{host}}}, \tag{13}$$

$$T_{\text{link gt}} \sim \frac{\frac{(n+n_{\text{LET}})}{4} \alpha_{\text{link}} \log_2 \left(\frac{(n+n_{\text{LET}})}{4} \right)}{B_{\text{host}}}, \tag{14}$$

23

Table 6. Coefficient for equations (11) to (14)

α_{key}	0.85
$\alpha_{\mathrm{reorder\,gt}}^{\mathrm{const}}$	5.0
b_{index}	4 byte

Table 7. Coefficient for equation (15)

$\alpha_{\mathrm{reorder\,gt}}^{\mathrm{reuse}}$	1.0

where n_{LET} is the number of LETs and b_{index} is the size of one index for EPJ and SPJ in bytes. Note that for the case of center-of-mass approximation used here, the type of SPJ is the same as that of EPJ and thus the size of SPJ is equal to b_{EPJ}. The coefficients for equations (11) to (14) are listed in table 6. We can see that $\alpha_{\mathrm{reorder\,gt}}^{\mathrm{const}}$ is larger than $\alpha_{\mathrm{reorder\,lt}}$ because for each node of LET we need to determine whether it is EPJ or SPJ.

6.2.5 Model of $T_{\mathrm{reorder\,gt}}$

At the reusing step, we also reorder the particle in Morton order of the global tree by using the index of arrays for EPJ and SPJ constructed at the construction step. Thus $T_{\mathrm{const\,gt}}$ is dominated by the times to read the indices, EPJ and SPJ once and that to write EPJ and SPJ once and therefore $T_{\mathrm{const\,gt}}$ is given by

$$T_{\mathrm{reorder\,gt}}^{\mathrm{reuse}} \sim \frac{(n + n_{\mathrm{LET}})\,\alpha_{\mathrm{reorder\,gt}}^{\mathrm{reuse}}\,(2b_{\mathrm{EPJ}} + b_{\mathrm{index}})}{B_{\mathrm{host}}}. \tag{15}$$

The coefficients for equation (15) are listed in table 7. We can see that $\alpha_{\mathrm{reorder\,gt}}^{\mathrm{reuse}}$ is much smaller than $\alpha_{\mathrm{reorder\,gt}}^{\mathrm{const}}$ because we use the indices of the arrays of EPJ and SPJ saved at the construction step to reorder the particles.

6.2.6 Model of $T_{\mathrm{mom\,gt}}$

The procedure of the calculation of the multipole moments of the cells of the global tree is almost the same as that of the local tree. Thus $T_{\mathrm{mom\,gt}}$ is given by

$$T_{\mathrm{mom\,gt}} \sim \frac{(n + n_{\mathrm{LET}})\,\alpha_{\mathrm{mom}}\,b_{\mathrm{EPJ}}}{B_{\mathrm{host}}}. \tag{16}$$

6.2.7 Models of $T_{\mathrm{force,const}}$ and $T_{\mathrm{force,reuse}}$

The elapsed times for the force calculation at the construction step $T_{\mathrm{force,const}}$ and at the reusing step $T_{\mathrm{force,reuse}}$ are given by

$$T_{\mathrm{force,const}} \sim \max\left(T_{\mathrm{cp\,all}},\, T_{\mathrm{H2D\,all}}\right)$$
$$+ \max\left(T_{\mathrm{kernel}},\, T_{\mathrm{H2D\,EPI}} + T_{\mathrm{H2D\,list}},\, T_{\mathrm{D2H\,FORCE}},\, T_{\mathrm{const\,list}} + T_{\mathrm{cp\,EPI}} + T_{\mathrm{cp\,list}} + T_{\mathrm{wb+int+cp}}\right)$$

$$+T_{\mathrm{cp\,FORCE}}, \tag{17}$$

$$T_{\mathrm{force,reuse}} \sim \max\left(T_{\mathrm{cp\,all}}, T_{\mathrm{H2D\,all}}\right)$$
$$+ \max\left(T_{\mathrm{kernel}}, T_{\mathrm{H2D\,EPI}}, T_{\mathrm{D2H\,FORCE}}, T_{\mathrm{cp\,EPI}} + T_{\mathrm{wb+int+cp}}\right)$$
$$+ T_{\mathrm{cp\,FORCE}}, \tag{18}$$

where $T_{\mathrm{cp\,all}}$, $T_{\mathrm{H2D\,all}}$, T_{kernel}, $T_{\mathrm{H2D\,EPI}}$, $T_{\mathrm{H2D\,list}}$, $T_{\mathrm{D2H\,FORCE}}$, $T_{\mathrm{const\,list}}$, $T_{\mathrm{cp\,EPI}}$, $T_{\mathrm{cp\,list}}$ and $T_{\mathrm{wb+int+cp}}$ are the times for copying of EPJs and SPJs to the send buffer, sending EPJs and SPJs from the host to GPGPU, the force kernel on GPGPU, sending EPIs to GPGPU, sending the interaction lists to GPGPU, receiving FORCEs from GPGPU, constructing the interaction list, copying EPIs to the send buffer, copying the interaction lists, and copying the data of FORCEs to FPs, integrating orbits of FPs and copying the data of FPs to EPIs and EPJs, respectively.

Each components in equations (17) and (18) are given by

$$T_{\mathrm{cp\,all}} \sim \frac{(n + n_{\mathrm{LET}})\alpha_{\mathrm{cp\,all}}\left(b_{\mathrm{EPJ}} + b_{\mathrm{EPJ\,buf}}\right)}{B_{\mathrm{host}}}, \tag{19}$$

$$T_{\mathrm{H2D\,all}} \sim \frac{(n + n_{\mathrm{LET}})\alpha_{\mathrm{H2D\,all}}b_{\mathrm{EPJ\,buf}}}{B_{\mathrm{transfer}}}, \tag{20}$$

$$T_{\mathrm{kernel}} \sim n\langle n_{\mathrm{list}}\rangle\left(\frac{\alpha_{\mathrm{GPU,kernel}}n_{\mathrm{op}}}{F_{\mathrm{GPU}}} + \frac{\alpha_{\mathrm{GPU,mm}}\left(b_{\mathrm{ID}} + b_{\mathrm{EPJ}}\right)}{\langle n_i\rangle B_{\mathrm{GPU}}}\right), \tag{21}$$

$$T_{\mathrm{H2D\,EPI}} \sim \frac{n\alpha_{\mathrm{H2D\,EPI}}b_{\mathrm{EPI\,buf}}}{B_{\mathrm{transfer}}}, \tag{22}$$

$$T_{\mathrm{H2D\,list}} \sim \frac{n\langle n_{\mathrm{list}}\rangle\alpha_{\mathrm{H2D\,list}}b_{\mathrm{ID}}}{\langle n_i\rangle B_{\mathrm{transfer}}}, \tag{23}$$

$$T_{\mathrm{D2H\,FORCE}} \sim \frac{n\alpha_{\mathrm{D2H,FORCE}}b_{\mathrm{FORCE\,buf}}}{B_{\mathrm{transfer}}}, \tag{24}$$

$$T_{\mathrm{const\,list}} \sim \frac{n\langle n_{\mathrm{list}}\rangle\alpha_{\mathrm{const\,list}}b_{\mathrm{ID}}}{\langle n_i\rangle B_{\mathrm{host}}}, \tag{25}$$

$$T_{\mathrm{cp\,EPI}} \sim \frac{n\alpha_{\mathrm{cp\,EPI}}\left(b_{\mathrm{EPI}} + b_{\mathrm{EPI\,buf}}\right)}{B_{\mathrm{host}}}, \tag{26}$$

$$T_{\mathrm{cp\,list}} \sim \frac{n\langle n_{\mathrm{list}}\rangle\alpha_{\mathrm{cp\,list}}b_{\mathrm{ID}}}{\langle n_i\rangle B_{\mathrm{host}}}, \tag{27}$$

$$T_{\mathrm{wb+int+cp}} \sim \frac{n\alpha_{\mathrm{wb+int+cp}}\left(b_{\mathrm{FORCE}} + b_{\mathrm{FP}} + b_{\mathrm{EPI}} + b_{\mathrm{EPJ}}\right)}{B_{\mathrm{host}}}, \tag{28}$$

$$T_{\mathrm{cp\,FORCE}} \sim \frac{n\alpha_{\mathrm{cp\,FORCE}}\left(b_{\mathrm{FORCE}} + b_{\mathrm{FORCE\,buf}}\right)}{B_{\mathrm{host}}}, \tag{29}$$

$$\langle n_{\mathrm{list}}\rangle \sim \langle n_i\rangle + \frac{14\langle n_i\rangle^{2/3}}{\theta} + \frac{21\pi\langle n_i\rangle^{1/3}}{\theta^2} + \frac{28\pi}{3\theta^3}\log_2\left[\frac{\theta}{2.8}\left(n^{1/3} - \langle n_i\rangle^{1/3}\right)\right], \tag{30}$$

where $b_{\mathrm{EPI(J)\,buf}}$, b_{ID}, b_{FORCE} and $b_{\mathrm{FORCE\,buf}}$ are the data size of EPI(J) in the send buffer, index of EPJ and SPJ in the interaction list, FORCE and FORCE in the receive buffer, B_{transfer} and B_{GPU} are the effective bandwidths of the data bus between the host and GPGPU and the main memory of

GPGPU, F_{GPU} is the peak speed of floating point operation of GPGPU, n_{op} is the number of floating point operation per interaction and $\langle n_{\mathrm{list}} \rangle$ is the average size of the interaction list. The elapsed times are determined by the bandwidth of the main memory of the host or the data bus between the host and the GPGPU, except for the time for the force calculation, T_{kernel}. The model of T_{kernel} is a bit complicated. We will describe how to construct this model in appendix A.

We summarize the coefficients for equations (19) to (28) in table 8. We can see that α_{list} and $\alpha_{\mathrm{GPU\,transfer}}$ are much larger than unity because the random access of the main memory is slow both the host and the GPGPU.

Table 8. Coefficients for equations (19) to (28)

$\alpha_{\mathrm{cp\,all}}$	0.87
$\alpha_{\mathrm{H2D\,all}}$	1.0
$\alpha_{\mathrm{cp\,EPI}}$	1.0
$\alpha_{\mathrm{H2D\,EPI}}$	1.0
$\alpha_{\mathrm{cp\,list}}$	1.0
$\alpha_{\mathrm{H2D\,list}}$	0.86
$\alpha_{\mathrm{cp\,FORCE}}$	1.2
$\alpha_{\mathrm{D2H\,FORCE}}$	1.2
$\alpha_{\mathrm{GPU\,calc}}$	1.7
$\alpha_{\mathrm{GPU\,transfer}}$	22
$\alpha_{\mathrm{const\,list}}$	19
$\alpha_{\mathrm{wb+int+cp}}$	1.4
$b_{\mathrm{EPJ\,buf}}$	16 byte
$b_{\mathrm{EPI\,buf}}$	12 byte
b_{ID}	4 byte
$b_{\mathrm{ID\,buf}}$	4 byte
b_{FORCE}	32 byte
$b_{\mathrm{FORCE\,buf}}$	16 byte
n_{op}	23

### 6.2.8 Model of $T_{\mathrm{step,const}}$ and $T_{\mathrm{step,reuse}}$

In the previous sections, we made the performance models of each step of the interaction calculation. By using these models, the total execution time is expressed by equations (31) for the construction step and by equation (32) for the reusing step.

$$
\begin{aligned}
T_{\text{step,const}} \sim \Big[& n(\alpha_{\text{root}} b_{\text{FP}} + \alpha_{\text{key}}\,(b_{\text{FP}} + b_{\text{key}}) + \alpha_{\text{reorder}}\,(2b_{\text{FP}} + b_{\text{EPI}} + b_{\text{EPJ}} + b_{\text{key}}) \\
& + \alpha_{\text{cp\,FORCE}}\,(b_{\text{FORCE}} + b_{\text{FORCE\,buf}})) \\
& + (n + n_{\text{LET}})\left(\alpha_{\text{key}}\,(b_{\text{EPJ}} + b_{\text{key}}) + \alpha_{\text{reorder\,gt}}^{\text{const}}\,(2b_{\text{EPJ}} + b_{\text{index}})\right) \\
& + (2n + n_{\text{LET}})(\alpha_{\text{sort}} 24 b_{\text{key}} + \alpha_{\text{mom}} b_{\text{EPJ}}) \Big]/B_{\text{host}} \\
& + \alpha_{\text{link}}\left((n/4)\log_2(n/4) + ((2n + n_{\text{LET}})/4)\log_2((2n + n_{\text{LET}})/4)\right)/B_{\text{host}} \\
& + (n + n_{\text{LET}})\max\Big[\alpha_{\text{cp\,all}}\,(b_{\text{EPJ}} + b_{\text{EPJ\,buf}})/B_{\text{host}}, \alpha_{\text{H2D\,all}} b_{\text{EPJ\,buf}}/B_{\text{transfer}}\Big] \\
& + n\max\Bigg[n_{\text{list}}\left(\frac{\alpha_{\text{GPU\,calc}} n_{\text{op}}}{F_{\text{GPU}}} + \frac{\alpha_{\text{GPU}}\,(b_{\text{ID}} + b_{\text{EPJ}})}{\langle n_i \rangle B_{\text{GPU}}}\right), \\
& \left(\alpha_{\text{H2D\,EPI}} b_{\text{EPI\,buf}} + \frac{n_{\text{list}}}{\langle n_{\text{i}} \rangle}\alpha_{\text{H2D\,list}} b_{\text{ID\,buf}}\right)/B_{\text{transfer}}, \\
& \alpha_{\text{D2H\,FORCE}} b_{\text{FORCE\,buf}}/B_{\text{transfer}}, \\
& \left(\alpha_{\text{cp\,EPI}}\,(b_{\text{EPI}} + b_{\text{EPI\,buf}}) + \frac{n_{\text{list}}}{\langle n_{\text{i}} \rangle}(\alpha_{\text{cp\,list}} b_{\text{ID}} + \alpha_{\text{const\,list}} b_{\text{ID}})\right. \\
& + \alpha_{\text{wb+int+cp}}\,(b_{\text{FORCE}} + b_{\text{FP}} + b_{\text{EPI}} + b_{EPJ})\Big)/B_{\text{host}}\Bigg].
\end{aligned}
\tag{31}
$$

$$
\begin{aligned}
T_{\text{step,reuse}} \sim \Big[& n\alpha_{\text{cp\,FORCE}}\,(b_{\text{FORCE}} + b_{\text{FORCE\,buf}}) + (n + n_{\text{LET}})\alpha_{\text{reorder\,gt}}^{\text{reuse}}\,(2b_{\text{EPJ}} + b_{\text{index}}) \\
& + (2n + n_{\text{LET}})\alpha_{\text{mom}} b_{\text{EPJ}}\Big]/B_{\text{host}} \\
& + (n + n_{\text{LET}})\max\Big[\alpha_{\text{cp\,all}}\,(b_{\text{EPJ}} + b_{\text{EPJ\,buf}})/B_{\text{host}}, \alpha_{\text{H2D\,all}} b_{\text{EPJ\,buf}}/B_{\text{transfer}}\Big] \\
& + n\max\Bigg[n_{\text{list}}\left(\frac{\alpha_{\text{GPU\,calc}} n_{\text{op}}}{F_{\text{GPU}}} + \frac{\alpha_{\text{GPU\,transfer}}\,(b_{\text{ID}} + b_{\text{EPJ}})}{\langle n_i \rangle B_{\text{GPU}}}\right), \\
& \alpha_{\text{H2D\,EPI}} b_{\text{EPI\,buf}}/B_{\text{transfer}}, \\
& \alpha_{\text{D2H\,FORCE}} b_{\text{FORCE\,buf}}/B_{\text{transfer}}, \\
& \left(\alpha_{\text{cp\,EPI}}\,(b_{\text{EPI}} + b_{\text{EPI\,buf}}) + \alpha_{\text{wb+int+cp}}\,(b_{\text{FORCE}} + b_{\text{FP}} + b_{\text{EPI}} + b_{\text{EPJ}})\right)/B_{\text{host}}\Bigg].
\end{aligned}
\tag{32}
$$

Thus, the mean execution time per step $T_{\text{step,single}}$, substituting the efficiency parameters αs and the size parameters bs with the measured values listed in tables is given by

$$
\begin{aligned}
T_{\text{step,single}} \sim & \frac{1}{n_{\text{reuse}}}\left(T_{\text{step,const}} + (n_{\text{reuse}} - 1)T_{\text{step,reuse}}\right) \\
\sim & \, 68\,(n + n_{\text{LET}})/B_{\text{host}} + (174n + 58n_{\text{LET}})\max(42/B_{\text{host}}, 16/B_{\text{transfer}}) \\
& + n\max\left(n_{\text{list}}(39.1/F_{\text{GPU}} + 440/(\langle n_i \rangle B_{\text{GPU}})), 19.2/B_{\text{transfer}}, 282/B_{\text{host}}\right) \\
& + \frac{1}{n_{\text{reuse}}}\Big[\big(1580n + 1157n_{\text{LET}}
\end{aligned}
$$
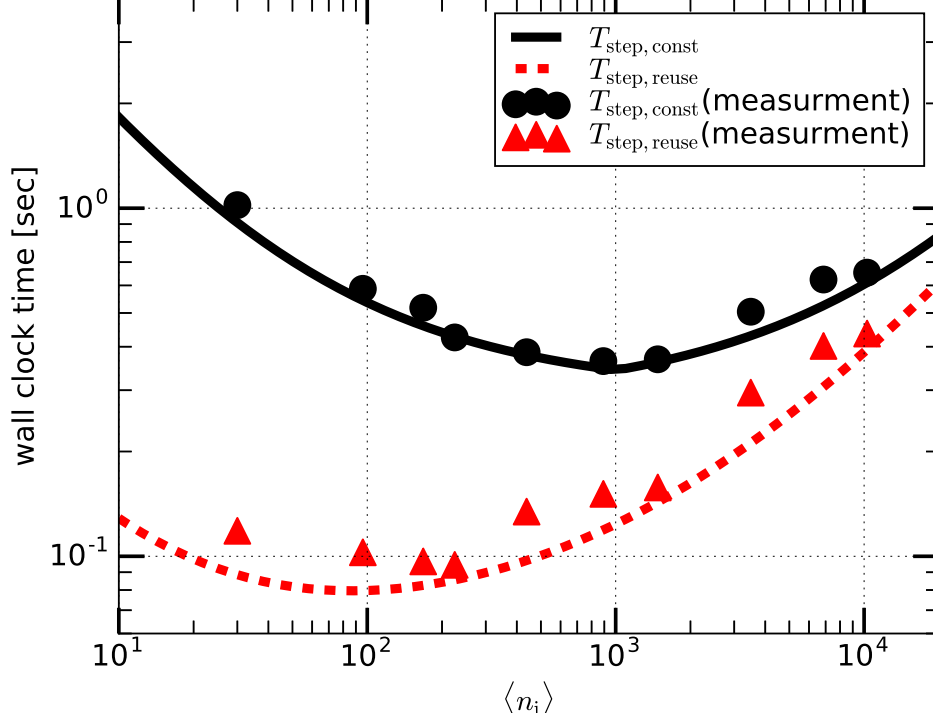
Fig. 7. Total elapsed time in the cases of both the construction and reusing steps. Solid (dashed) curve and Filled circles (triangles) indicate respectively the predicted time by using the model and the measured times, in the case of the construction (reusing) step.

$$+54.4\left((n/4)\log_2(n/4)+((2n+n_{\text{LET}})/4)\log_2((2n+n_{\text{LET}})/4))\right)/B_{\text{host}}$$

$$+n\bigg(\max\Big(n_{\text{list}}\left(39.1/F_{\text{GPU}}+440/(\langle n_i\rangle B_{\text{GPU}})\right),(12+3.4n_{\text{list}}/\langle n_i\rangle)/B_{\text{transfer}},$$

$$19.2/B_{\text{transfer}},(282+80n_{\text{list}}/\langle n_i\rangle)/B_{\text{host}}\Big),$$

$$-\max\Big(n_{\text{list}}\left(39.1/F_{\text{GPU}}+440/(\langle n_i\rangle B_{\text{GPU}})\right),19.2/B_{\text{transfer}},282/B_{\text{host}}\Big)\bigg)\bigg] \quad (33)$$

In order to check whether the model we constructed is reasonable or not, we compared the time predicted by equations (31) and (32) with the measured times in figure 7. The predicted times agree with the measured times very well.

In the following, we analyze the performance of the N-body simulations on hypothetical systems with various B_{host}, B_{transfer}, F_{GPU} and B_{GPU} by using the performance model. Unless otherwise noted, we assume the hypothetical system with $B_{\text{host}} = 100$ GB/s, $B_{\text{transfer}} = 10$ GB/s, $B_{\text{GPU}} = 500$ GB/s and $F_{\text{GPU}} = 10$ Tflops as a reference system. This reference system can be regarded as a modern HPC system with a high-end accelerator.

Figure 8 shows the calculation time per timestep on the reference system for 4M particles and $\theta = 0.5$ for $n_{\text{reuse}} = 1$, 4 and 16. We can see that the difference in the performance for $n_{\text{reuse}} = 4$ and
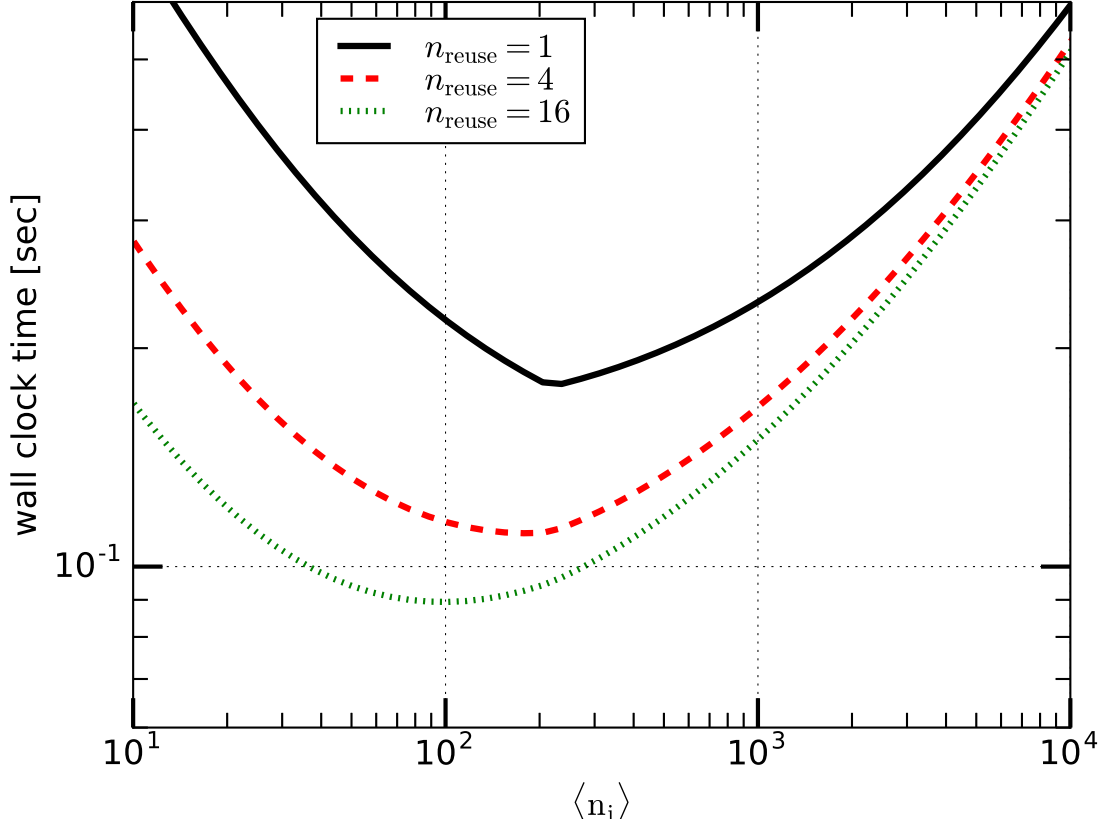
Fig. 8. Calculation time per timestep on the reference system for various n_{reuse}.

system	F_{GPU}	B_{host}	B_{GPU}	B_{transfer}
reference	10 Tflops	100 GB/s	500 GB/s	10 GB/s
GPU2X	20 Tflops	100 GB/s	1 TB/s	10 GB/s
GPU4X	40 Tflops	100 GB/s	2 TB/s	10 GB/s
LINK4X	10 Tflops	100 GB/s	500 GB/s	40 GB/s
LINK0.5X	10 Tflops	100 GB/s	500 GB/s	5 GB/s

Table 9. Parameters of hypothetical systems.

$n_{\text{reuse}} = 16$ is relatively small. In the rest of the section, we use $n_{\text{reuse}} = 16$ and $\theta = 0.5$.

We consider four different types of hypothetical systems: GPU2X, GPU4X, LINK4X and LINK0.5X. Their parameters are listed in table 9. Figure 9 shows the calculation times per timestep for the four hypothetical systems. We can see that increasing the bandwidth between CPU and accelerator (LINK4X) has relatively small effect on the performance. On the other hand, increasing overall accelerator performance has fairly large impact.
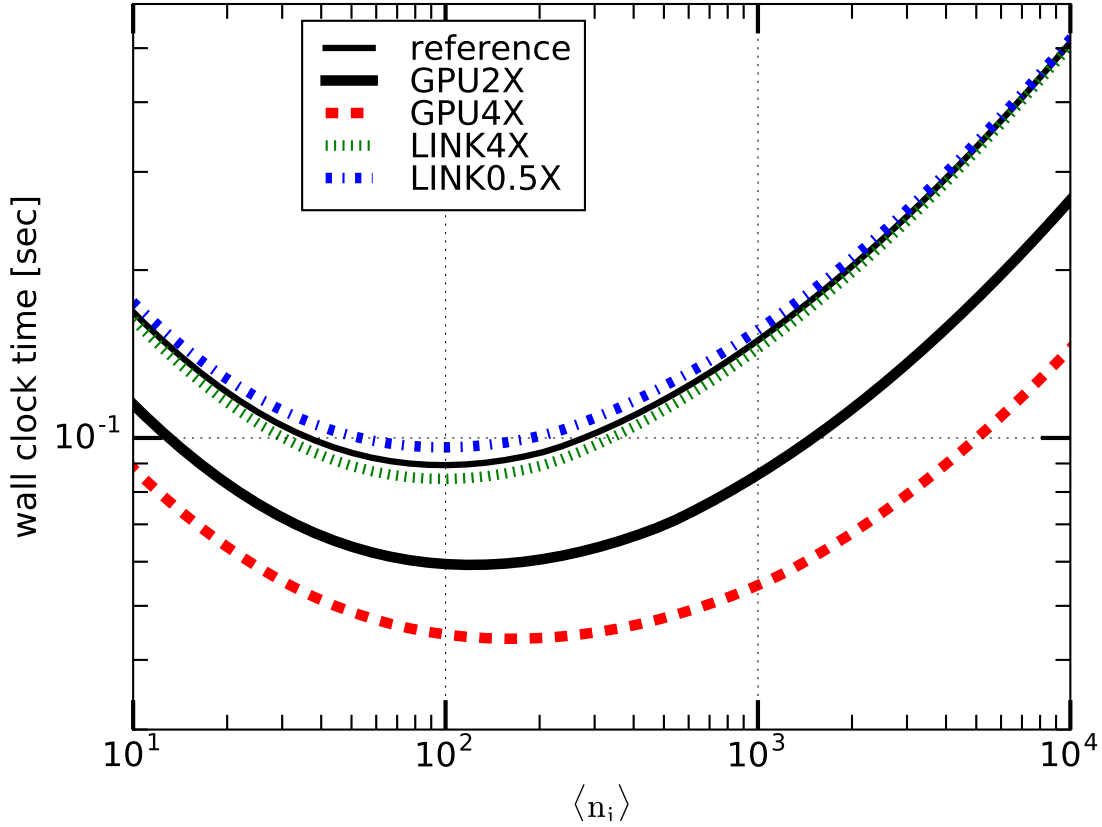
Fig. 9. The mean execution time per step, $T_{\mathrm{step\,single}}$, on various hypothetical systems against $\langle n_i \rangle$.

Figure 10 shows the relative execution time of hypothetical systems in the two-dimensional plane of B_{host} and B_{transfer}. We can see that the increase of B_{host} or B_{transfer}, or even both, would not give significant performance improvement, while the increase of the accelerator performance gives significant speedup. Even when both B_{host} and B_{transfer} are reduced by a factor of 10, overall speed is reduced by a factor of 4. Thus, if the speed of accelerator is improved by a factor of 10, the overall speedup is 4. Thus we can conclude that the current implementation of FDPS can provide good performance not only on the current generation of GPGPUs but also future generations of GPGPUs or other accelerator-based systems, which will have relatively poor data transfer bandwidth compare to the calculation performance.

6.3 Performance Model on Multiple Nodes

In this section, we discuss the performance model of the parallelized N-body simulation with method G2. Here, we assume the network is the same as that of K computer. Detailed communication algorithms are given in Iwasawa et al. (2016).
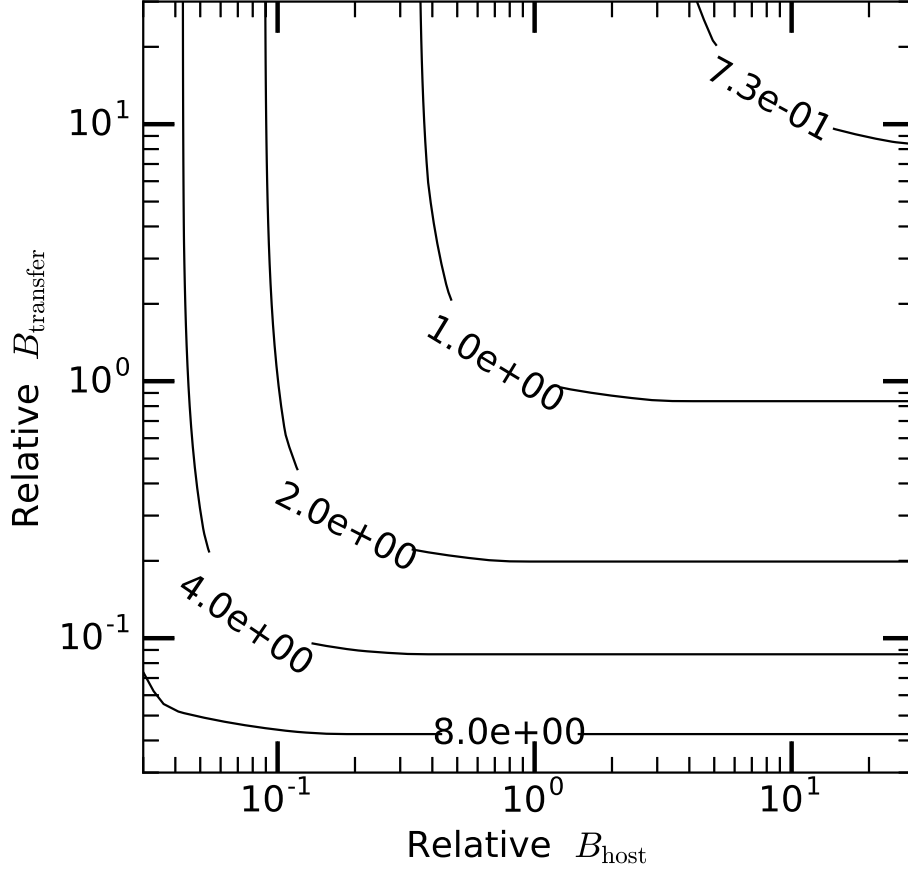
Fig. 10. Relative execution time of hypothetical systems in the two-dimensional plane of relative B_{host} and relative B_{transfer}. The values of contours, x- and y- axes are normalized by those of the reference system.

The time per step is given by

$$T_{\text{step,para}} = T_{\text{step,single}} + T_{\text{exch}}/n_{\text{reuse}} + T_{\text{dc}}/n_{\text{dc}} + T_{\text{LET,const}}/n_{\text{reuse}} + T_{\text{LET,exch}}, \qquad (34)$$

where T_{exch}, T_{dc} $T_{\text{LET,const}}$ and $T_{\text{LET,exch}}$ are the times for the exchange of particles, the domain decomposition, the construction of LETs and the exchange of LETs and n_{dc} is the number of steps for which the same domain decomposition is used. We consider the case when particles do not move much in a single step and thus ignore T_{exch}.

The time for the domain decomposition is given by

$$T_{\text{dc}} \sim T_{\text{dc,collect}} + T_{\text{dc,sort}}, \qquad (35)$$

where $T_{\text{dc,collect}}$ and $T_{\text{dc,sort}}$ are the times to collect sample particles to root processes and to sort particles on the root processes.

According to Iwasawa et al. (2016), $T_{\text{dc,collect}}$ and $T_{\text{dc,sort}}$ are given by

31

$$T_{\mathrm{dc,collect}} \sim n_{\mathrm{p}}^{1/6} \tau_{\mathrm{gather,startup}} + \frac{n_{\mathrm{smp}} n_{\mathrm{p}}^{2/3} \alpha_{\mathrm{gather}} b_{\mathrm{pos}}}{B_{\mathrm{inj}}}, \tag{36}$$

$$T_{\mathrm{dc,sort}} \sim \frac{n_{\mathrm{smp}} n_{\mathrm{p}}^{2/3} \log\left(n_{\mathrm{smp}}^3 n_{\mathrm{p}}^{5/3}\right) \alpha_{\mathrm{dc,sort}} b_{\mathrm{pos}}}{B_{\mathrm{host}}}, \tag{37}$$

where n_{p} is the number of processes, n_{smp} is the number of sample particles to determine the domains, b_{pos} is the data size of the position of the particle and B_{inj} is the effective injection bandwidth, $\tau_{\mathrm{gather,startup}}$ is the startup time for `MPI_Gather` and α_{gather} is the efficiency parameter of communicating data with `MPI_Gather`. We will describe how to measure the parameters $\tau_{\mathrm{gather,startup}}$ and α_{gather} in appendix B. The coefficients for equation (37) are listed in table 10. Since we used the quick sort here, $\alpha_{\mathrm{dc\,sort}}$ is much larger than unity.

Table 10. Coefficients for equations (36) and (37)

$\alpha_{\mathrm{dc\,sort}}$	7.7
α_{gather}	0.62
$\tau_{\mathrm{gather,startup}}$	1.2×10^{-5} sec
b_{pos}	24 byte

In the original implementation of FDPS, `MPI_Alltoallv` was used for the exchange of LETs and it was the main bottleneck of the performance for large n_{p} (Iwasawa et al. 2016). Thus, recently, we developed a new algorithms for the exchange of LETs to avoid the use of `MPI_Alltoallv` (Iwasawa et al. 2018). The new algorithm is as follows:

1. Each process sends the multipole moment of the top level cell of its local tree to all processes using `MPI_Allgather`.
2. Each process calculates the distances from all other domains.
3. If the distance between process i and j is large enough so that process i can be regarded as one cell from process j, that domain already has the necessary information. If not, process i construct LET for process j and send it to process j.

With this new algorithm, the times for the exchange LET is expressed as

$$T_{\mathrm{LET,const}} \sim \frac{(n_{\mathrm{LET}} - n_{\mathrm{p}} + n_{\mathrm{p,close}}) \alpha_{\mathrm{const\,list}} b_{\mathrm{ID}}}{B_{\mathrm{host}}}, \tag{38}$$

$$T_{\mathrm{LET,exch}} \sim T_{\mathrm{LET,allgather}} + T_{\mathrm{LET,p2p}}, \tag{39}$$

$$T_{\mathrm{LET,allgather}} \sim n_{\mathrm{p}}^{1/4} \tau_{\mathrm{allgather,startup}} + \frac{n_{\mathrm{p}} \alpha_{\mathrm{allgather}} b_{\mathrm{EPJ}}}{B_{\mathrm{inj}}}, \tag{40}$$

$$T_{\mathrm{LET,p2p}} \sim n_{\mathrm{p,close}} \tau_{\mathrm{p2p,startup}} + \frac{2\left(n_{\mathrm{LET}} - n_{\mathrm{p}} + n_{\mathrm{p,close}}\right) \alpha_{\mathrm{p2p}} b_{\mathrm{EPJ}}}{B_{\mathrm{inj}}}, \tag{41}$$

$$n_{\text{LET}} \sim \frac{14n^{2/3}}{\theta} + \frac{21\pi n^{1/3}}{\theta^2} + \frac{28\pi}{3\theta^3}\log_2\left\{\frac{\theta}{2.8}\left[(nn_{\text{p}})^{1/3} - n^{1/3}\right]\right\} + n_{\text{p}} - n_{\text{p,close}} \tag{42}$$

$$n_{\text{p,close}} \sim 6\left(\frac{1}{\theta}+1\right) + 3\pi\left(\frac{1}{\theta}+1\right)^2 + \frac{4\pi}{3}\left(\frac{1}{\theta}+1\right)^3, \tag{43}$$

where $T_{\text{LET,allgather}}$ and $T_{\text{LET,p2p}}$ are the time for the time for exchange LETs using `MPI_Allgather` and `MPI_Isend/Irecv`, $n_{\text{p,close}}$ is the number of the processes to exchange LETs with `MPI_Isend/Irecv`, $\tau_{\text{p2p,allgather}}$ and $\tau_{\text{p2p,startup}}$ are the startup times for `MPI_Allgather` and `MPI_Isend/Irecv` and $\alpha_{\text{allgather}}$ and α_{p2p} are the efficiency parameters for exchange LETs with `MPI_Allgather` and `MPI_Isend/Irecv`, respectively.

The parameters for equations (40) and (41) are listed in table 11.

Table 11. Coefficients for equations (40) and (41)

$\alpha_{\text{allgather,ptcl}}$	0.62
α_{p2p}	1.0
$\tau_{\text{allgather,startup}}$	1.2×10^{-5} sec
$\tau_{\text{p2p,startup}}$	1.0×10^{-5} sec

Figure 11 shows the weak scaling performance for N-body simulations in the case of the number of particles per process of $n = 2^{20}$. Here we assume that $n_{\text{reuse}} = n_{\text{dc}} = 16$, $n_{\text{smp}} = 30$, $\langle n_{\text{i}}\rangle = 250$ and $\theta = 0.5$. We can see that $T_{\text{step,para}}$ is almost constant for $n_{\text{p}} \lesssim 10^5$. For $n_{\text{p}} \gtrsim 10^5$, $T_{\text{step,para}}$ slightly increases because $T_{\text{LET,allgather}}$ becomes large. Roughly speaking, when n is much larger than n_{p}, the parallel efficiency is high.

Figure 12 shows the strong scaling performance, in the case of the total number of particles N=2^{30} (left panel) and N =2^{40} (right panel). In the case of $N=2^{40}$, $T_{\text{step,para}}$ scales well up to $n_{\text{p}}=10^6$. On the other hand, in the case of $N=2^{30}$, for $n_{\text{P}} \gtrsim 3000$, the slope of $T_{\text{step,para}}$ becomes shallower because of $T_{\text{LET,p2p}}$ becomes relatively large. For $n_{\text{P}} \gtrsim 50000$, $T_{\text{step,para}}$ increases because $T_{\text{LET,allgather}}$ becomes relatively large. We can also see that $T_{\text{step,single}}$ increases linearly with n_{p} for $n_{\text{p}} \gtrsim 10^5$. This is because $T_{\text{step,single}}$ depends on n_{LET} which is proportional to n_{p} for large n_{p}. Thus to improve the scalability further, we need to reduce $T_{\text{LET,allgather}}$ and $T_{\text{LET,p2p}}$. We will discuss the ideas to reduce them in sections 7.1 and 7.2.

7 Discussion and Summary

7.1 Further Reduction of Communication

For the simulations on multiple nodes, the communication of the LETs with neighboring nodes ($T_{\text{LET,p2p}}$) would become bottlenecks for very large n_{p}. Thus, it is important to reduce the amount of
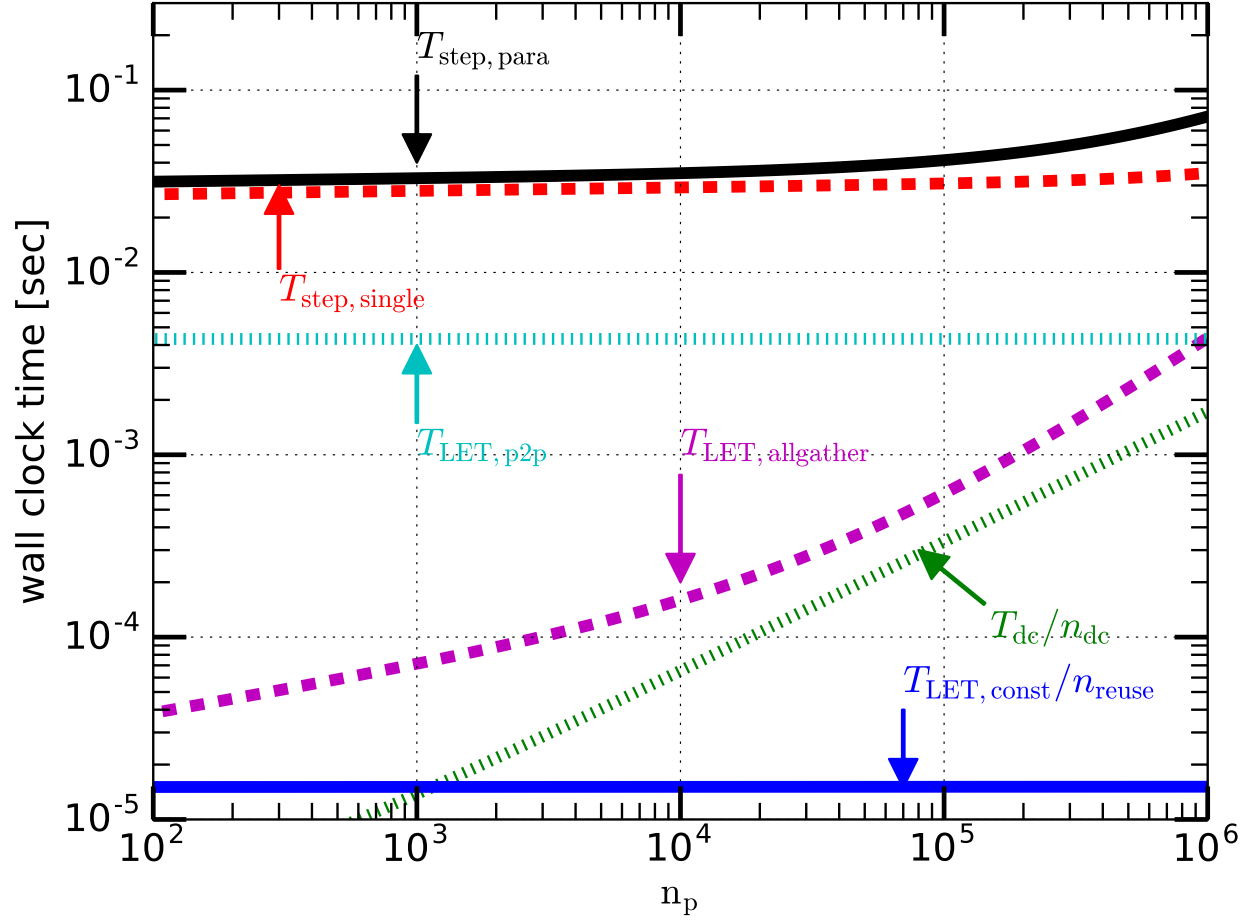
Fig. 11. Weak-scaling calculation time plotted as a function of $n_{\rm p}$. The number of particles per node is 2^{20}.
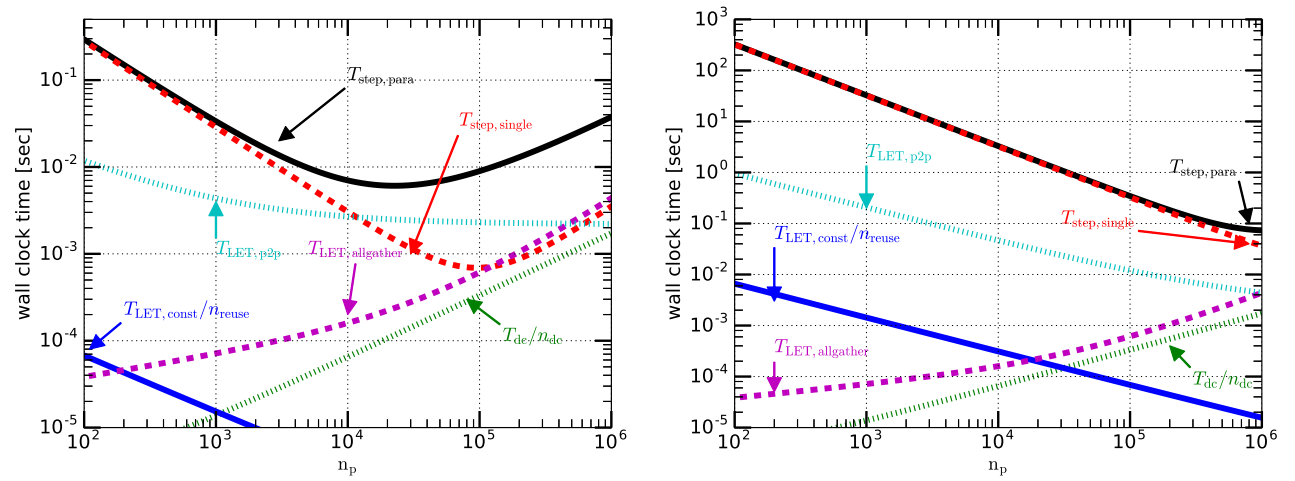

Fig. 12. Strong-scaling calculation time plotted as a function of $n_{\rm p}$. Left and right panels show the results in the cases of the total number of particles N of 2^{30} and 2^{40}, respectively.

the data size for this communication.

An obvious way to reduce the amount of data transfer is to apply some data compression technique. For example, physical coordinates of neighboring particles are similar, and thus there is clear room for compression. However, for many particle-based simulations, compression in the time domain might be more effective. In the time domain, we can make use of the fact that the trajectories of most of particles are smooth. For smooth trajectories, we can construct fitting polynomials from several previous timesteps. When we send the data of one particle at new timestep, we send only the difference between the prediction from the fitting polynomial and the actual value. Since this difference is many orders of magnitude smaller than the absolute value of the coordinate itself, we should be able to use much shorter word format for the same accuracy. We probably need some clever way to use variable-length word format. We can apply the compression in the time domain not only for coordinates but for any physical quantities of particles, including the calculated interactions such as gravitational potential and acceleration. We can also apply the same compression technique to communication between the CPU and the accelerator.

7.2 Tree of Domains

As we have seen in figure 12, for large n_p, the total elapsed time increases linearly with n_p because the elapsed times for the exchange of LETs and the construction of the global tree are proportional to n_p if n_p is very large. To remove this linear dependency on n_p, we can introduce the tree structure to the computational domains (tree of domains) (Iwasawa et al. 2018). By using the tree of domains and exchanging the locally combined multipole moments between distant nodes, we can remove `MPI_Allgather` among all processes to exchange LETs. It means that the times for the exchange of LETs and the construction of the global tree do not increase linearly with n_p.

7.3 Further Improvement on Single Node Performance

Considering the trends in HPC systems, the overall performance of the accelerator (F_{GPU} and B_{GPU}) increases faster than the bandwidths of the host main memory (B_{host}) and the data bus between the host and the accelerator (B_{transfer}). Therefore, in the near future, the main bottleneck of the performance could be B_{host} and B_{transfer}.

The amount of data copy in the host main memory and data transfer between the host and the accelerator for the reusing step are summarized in tables 12 and 13. We can see that the amounts of copying data and transferring data are about $15n$ and $3n$. One reason of these large amount of data access is that there are four different data types of particles (FP, EPI, EPJ and FORCE) and data are

Procedure	Amount of copying data
a) Calculate multipole moments of local tree	n
b) Reorder particles for global tree	$3n$
c) Calculate multipole moments of global tree	n
d) Copy EPJ and SPJ to the send buffer	$2n$
e) Copy EPI to the send buffer	$2n$
f) Copy FORCE from the receive buffer	$2n$
g) Write back FORCEs to FPs, integrate orbits of FPs and copy FPs to EPIs and EPJs	$4n$

Table 12. Amount of coping particle particle data in the host main memory for specified procedure (left column).

Procedure	Amount of transferring data
A) Send EPJ and SPJ	n
B) Send EPI	n
C) Receive FORCE	n

Table 13. Amount of transferring particle data between the host and the accelerator for specified procedure (left column).

copied between different data types. If we use only one data type of particle, we could avoid to copy data of the procedures e) and g) in table 12 and the procedure B) in table 13. If we do so, the amount of copying data in the main memory and transferring data between the host and the accelerator could be reduced by 40% and 33%, respectively.

Another way to improve the performance is to implement all procedures on the accelerator because the bandwidth of the device memory (B_{GPU}) is much faster than B_{host} and B_{transfer}. In this case, the performance would be determined by the overall performance of the accelerator.

7.4 Summary

In this paper, we described the algorithms we implemented to FDPS to allow efficient and easy use of accelerators. Our algorithm is based on Barnes' vectorization, which has been used both on general-purpose computers (and thus previous versions of FDPS), and also on GRAPE special-purpose processors and GPGPUs. However, we have minimized the amount of the communication between the CPU and the accelerator by indirect addressing method, and we further reduce the amount of calculation on the CPU side by interaction list reusing. The performance improvement over the simple method based on Barnes' vectorization on CPU can be as large as a factor of 10 on the current generation of accelerator hardware. We can expect fairly large performance improvement also on future

Table A.1. Coefficient for equation (A.1)

$\alpha_{\mathrm{GPU\,calc}}$	1.7
$\alpha_{\mathrm{GPU\,transfer}}$	2.7
n_{op}	23

hardware, even if the relative communication performanceis expected to degrade.

The version of FDPS described in this paper is available at https://github.com/FDPS/FDPS.

## Appendix A Performance model of force kernel on GPGPU

Here, we construct the performance model of the force kernel on the GPGPU. Figure A.1 shows the measured time for the force kernel per interaction against $\langle n_{\mathrm{i}} \rangle$ for various θ. We can see that the elapsed times are independent of θ (i.e. independent of $\langle n_{\mathrm{list}} \rangle$) and depend on $\langle n_{\mathrm{i}} \rangle$. For small $\langle n_{\mathrm{i}} \rangle$, the time decreases as $\langle n_{\mathrm{i}} \rangle$ increases. This is because the times are determined by the bandwidth of the main memory of GPGPUs (B_{GPU}). For large $\langle n_{\mathrm{i}} \rangle$, the elapsed times are almost constant because these times are determined by the speed of the floating point operation of GPGPUs (F_{GPU}). Thus the time for the force kernel T_{kernel} is given by

$$T_{\mathrm{kernel}} = n\langle n_{\mathrm{list}} \rangle \left(\frac{\alpha_{\mathrm{GPU\,calc}} n_{\mathrm{op}}}{F_{\mathrm{GPU}}} + \frac{\alpha_{\mathrm{GPU\,transfer}}\,(b_{\mathrm{EPJ}} + b_{\mathrm{ID}})}{\langle n_i \rangle B_{\mathrm{GPU}}} \right). \tag{A.1 }$$

To determine the coefficients for equation (A.1), we assume that F_{GPU} is 13.8 Tflops and B_{GPU} is about 550 GB/s, which is measured with `bandwidthTest` in the NVIDIA SDK. These coefficients are listed in table A.1.

Appendix B Performance of `MPI_Gather` and `MPI_Allgather` on K Computer

Here, we construct the performance models of `MPI_Gather` and `MPI_Allgather` on K computer. On K computer, the performance of `MPI_Gather` is almost the same as that of `MPI_Allgather`. Thus in
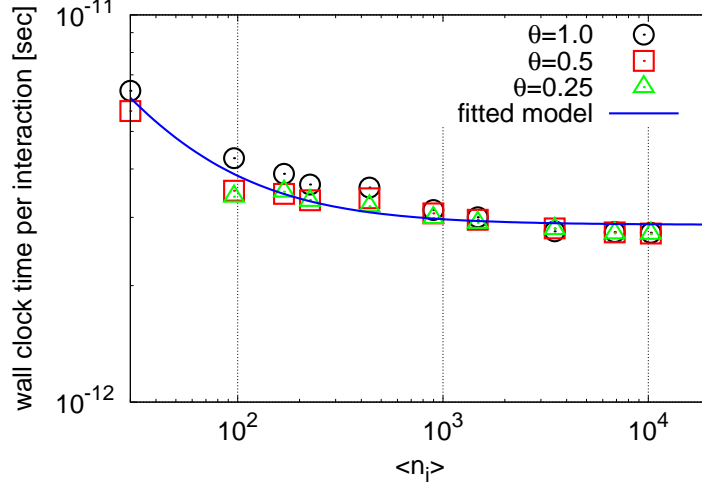
Fig. A.1. Elapsed time for interaction calculations per interaction. Circles, squares and triangles indicate the results for θ=1.0, 0.5 and 0.25, respectively. Solid curve indicates the fitted model.

the following, we only consider `MPI_Allgather`.

The elapsed time for `MPI_Allgather` $T_{\text{allgather}}$ as a function of message size b and the number of processes n_{p} is given by

$$T_{\text{allgather}}(b, n_{\text{p}}) = T_{\text{allgather,startup}}(n_{\text{p}}) + T_{\text{allgather,words}}(b, n_{\text{p}}), \tag{B.2}$$

where $T_{\text{allgather,startup}}$ is the start up time which depends on only n_{p} and $T_{\text{allgather,words}}$ is the time for transfer of the message which depend on both n_{p} and b. For small message size, $T_{\text{allgather}} \sim T_{\text{allgather,startup}}$. Thus to determine $T_{\text{allgather,startup}}$ we measured the times for `MPI_Allgather` with a short message size.

Figure B.2 shows the elapsed time for `MPI_Allgather` to send message of two bytes against n_{p}. We can see that $T_{\text{allgather,startup}}$ is proportional to $n_{\text{p}}^{1/4}$. Thus $T_{\text{allgather,startup}}$ is given by

$$T_{\text{allgather,startup}} \sim n_{\text{p}}^{1/4} \tau_{\text{allgather,startup}}. \tag{B.3}$$

For large message size, $T_{\text{allgather}}$ should be determined by the injection bandwidth B_{inj}. Thus $T_{\text{allgather,words}}$ is give by

$$T_{\text{allgather,words}} \sim \frac{\alpha_{\text{allgather}} b n_{\text{p}}}{B_{\text{inj}}}. \tag{B.4}$$

Thus the elapsed time for `MPI_Allgather` is given by

$$T_{\text{allgather}} \sim n_{\text{p}}^{1/4} \tau_{\text{allgather,startup}} + \frac{\alpha_{\text{allgather}} b n_{\text{p}}}{B_{\text{inj}}}. \tag{B.5}$$

Figure B.3 shows the measured and predicted times for `MPI_Allgather` against the message size. Here, we assume B_{inj}=4.8 GB/s from the result of point-to-point communication test. The
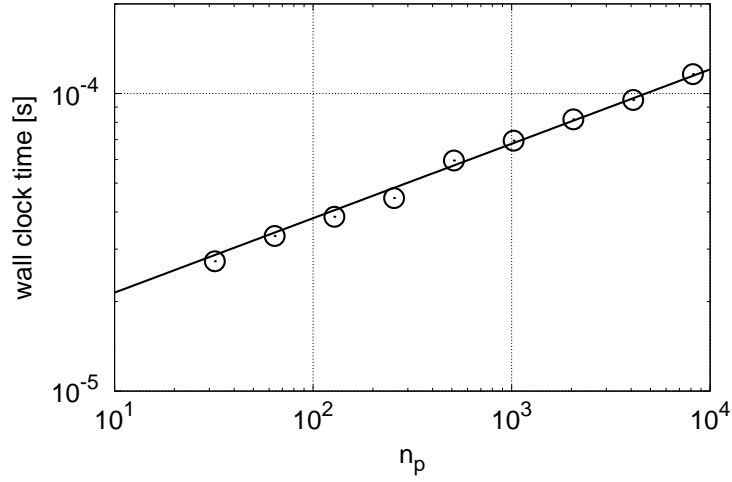
Fig. B.2. Elapsed time for `MPI_Allgather` to send message of 2 bytes against the number of processes. Circles indicate the measurement values and solid curve indicates our fitting model.
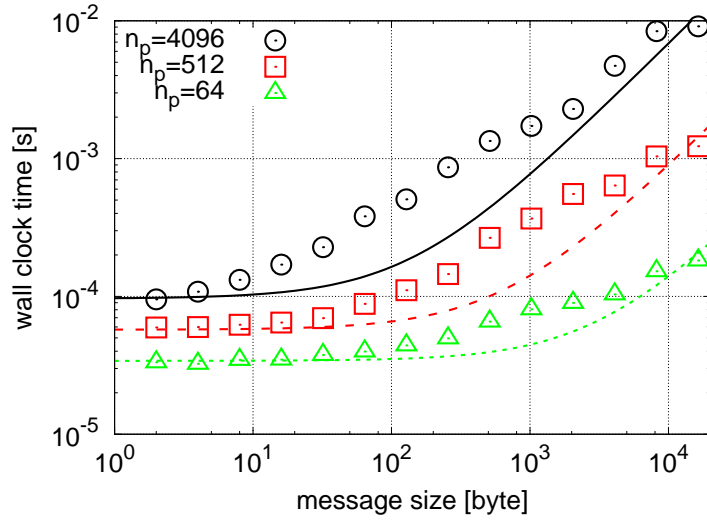
Fig. B.3. Elapsed times for `MPI_Allgather` plotted against the message size. Circle, square and triangle indicate the result with n_{p}=4096, 512 and 64, respectively. Solid, dashed and dotted curves indicate our fitting models for n_{p}=4096, 512 and 64, respectively.

parameters in equation (B.5) are listed in table B.2. Our model agrees with the measured data.

Table B.2. Coefficient for equation (B.5)

$\tau_{\mathrm{allgather,startup}}$	1.2×10^{-5} sec
$\alpha_{\mathrm{allgather}}$	0.62

References

Ballouz, R.-L., Richardson, D. C., & Morishima, R. 2017, AJ, 153, 146

Barnes, J., & Hut, P. 1986, Nature, 324, 446

Barnes, J. E. 1990, Journal of Computational Physics, 87, 161

Bédorf, J., Gaburov, E., & Portegies Zwart, S. 2012, Journal of Computational Physics, 231, 2825

Bédorf, J., Gaburov, E., Fujii, M. S., et al. 2014, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 54-65, 54

Gaburov, E., Harfst, S., & Portegies Zwart, S. 2009, New Astronomy, 14, 630

Hamada, T., Narumi, T., Yokota, R., Yasuoka, K., Nitadori, K., & Taiji, M. 2009, Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 62, 12

Ishiyama, T., Fukushige, T., & Makino, J. 2009, PASJ, 61, 1319

Ishiyama, T., Nitadori, K., & Makino, J. 2012, Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 5, 1

Ishiyama, T. 2014, ApJ, 788, 27

Iwasawa, M., Tanikawa, A., Hosono, N., et al. 2016, PASJ, 68, 54

Iwasawa, M., Wang, L., Nitadori, K., et al. 2018, Lecture Note in Computational Science, 10860, 483

Knuth, D. 1997, The Art of Computer Programming, Volume 3

Makino, J. 1991, PASJ, 43, 621

Makino, J., Fukushige, T., Koga, M., & Namura, K. 2003, PASJ, 55, 1163

Makino, J. 2004, PASJ, 56, 521

Makino, J., & Daisaka 2012, SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis,

Michikoshi, S., & Kokubo, E. 2017, ApJL, 837, L13

Namekata, D., Iwasawa, M., Nitadori, K., et al. 2018, PASJ

Nitadori, K., & Aarseth, S. J. 2012, MNRAS, 424, 545

Portegies Zwart, S., & Bédorf, J. 2014, arXiv:1409.5474

Rein, H., & Latter, H. N. 2013, MNRAS, 431, 145

H. Rein & S.-F. Liu, 2012, A&A, 537, 128

Springel, V. 2005, MNRAS, 364, 1105

Stadel, J. G. 2001, Ph.D. Thesis, 3657

Sugimoto, D., Chikada, Y., Makino, J., et al. 1990, Nature, 345, 33

Wisdom, J., & Tremaine, S. 1988, AJ, 95, 925

Zebker, H. A., Marouf, E. A., & Tyler, G. L. 1985, Icarus, 64, 531