Noname manuscript No.
(will be inserted by the editor)

Extreme-scale particle-based simulations on advanced HPC platforms

Lessons from PEZY-SC2, Sunway Taihulight and NVIDIA Volta

M. Iwasawa · D. Namekata · K. Nomura · M. Tsubouchi · J. Makino

Received: date / Accepted: date

Abstract We overview the current status and future development directions of our FDPS (Framework for Developing Particle Simulator) framework. Many of particle-based simulation codes share the same characteristic that the most time-consuming part of the simulation is the calculation of the interactions between particles, and a large fraction of programming effort is spent for procedures to make the force calculation efficient, such as the decomposition of computational domain, exchange of particles between domains, exchange of information necessary to calculate the interaction to particles in different domains, and efficient neighbor search. The basic idea of FDPS is to provide generic and high-performance library for these procedures. Using these procedures, researchers or application programmers in various fields can write their programs without taking care of parallelization and performance tuning. In order to make FDPS useful on advanced HPC platforms at present and (near) future, we investigated its performance on several modern platforms and learned what can be the bottleneck. In this paper we summarize what we learned.

Keywords Particle-based simulations · High-performance computing

Masaki Iwasawa
Center for Planetary Science, Graduate School of Science, Kobe University E-mail: iwasawa@people.kobe-u.ac.jp

Daisuke Namekata
RIKEN Center for Computational Science E-mail: daisuke.namekata@riken.jp

Kentaro Nomura
Center for Planetary Science, Graduate School of Science, Kobe University E-mail: kentaro.nomura@port.kobe-u.ac.jp

Miyuki Tsubouchi
RIKEN Center for Computational Science E-mail: miyuki.tsubouchi@riken.jp

Junichiro Makino
Department of Planetology, Graduate School of Science, Kobe University E-mail: makino@mail.jmlab.jp

Contents

1 Introduction

1.1 Background

Large-scale particle-based simulations are now used to model physical systems of all scales, from molecular scale to cosmological scale. As a result, a number of simulation programs have been developed and are being maintained. Just to give several examples, Amber[18], NAMD[13], and GROMACS[16] are used for classical molecular dynamics simulations of biomolecules, LAMMPS[14]. For cosmological N-body or N-body+SPH simulations, PKDGRAV[15], Gadget[19], and GreeM[7] have been available. It is certainly the case that for each research field several groups are developing their own codes for particle-based simulations to meet their specific needs.

On the other hand, it has become quite difficult to achieve good, or even moderate efficiency for large-scale parallel particle-based simulation code on modern platforms, and will be even more difficult in future. There are many reasons for this difficulty. To illustrate them, let us start with one of the most efficient and scalable code, GreeM. It is designed for large-scale cosmological N-body simulations on large HPC systems, and it was used for the cosmological simulation on K computer which was awarded the 2012 Gordon Bell Prize. It uses the multisection method with sampling algorithm[12] for the domain

decomposition, and parallel Barns-Hut treecode modified for periodic boundary (TreePM) based on the LET(local essential tree[17]). The force calculation within one MPI process is further parallelized by Barnes' vectorization algorithm, to make use of the SIMD execution units of modern computers. It achieved the efficiency of more than 50%, for the cosmological N-body simulation of 10240^3 particles on the entire K computer with 82944 nodes.

Each node of K computer has one 8-core SPARC64 VIIIfx CPU with the theoretical peak performance of 128 Gflops, and one MPI process runs on one node. K computer has very fast memory (peak B/F number of 0.5) and rich network of 6D torus ($24 \times 18 \times 16 \times 3 \times 2 \times 2$). Thus, each node has 10 links, and the speed of each link is 5GB/s bidirectional. Also, SPARC64 VIIIfx processor has two unique features. First one is that its L2 cache is physically shared by all eight cores, and the second one is that the barrier synchronization of cores is supported by hardware and thus extremely fast.

Many of other modern HPC systems lack these features. Table 1 shows some of the biggest modern HPC systems. We can see that K has exceptionally fast memory and rich network, and other systems have relatively weak memory and network. Even in the case of Fugaku, the relative network bandwidth is much less than that of K. Thus, applications which works fine on K (or Fugaku) do not necessarily run efficiently on other platforms, and we need to modify the implementation or introduce new algorithms to reduce the necessary memory/network bandwidth.

It is also necessary to rewrite the code on systems which offer platform-specific programming environment. For example, Cuda is used on NVIDIA GPGPUs. Though in principle Sunway Taihulight can be programmed with OpenACC, in many cases we need to write codes using Athread library. GYOUKOU offers PZCL, a dialect of OpenCL, as its only available programming environment.

The architectures, in particular the structures of the memory systems, are also all different on these machines. NVIDIA V100 has rather traditional host-accelerator architecture with cache-based memory systems on both sides. PEZY-SC2, at least as planned, would have single physical memory shared by both of its MIPS64 cores and proprietary SC2 cores, and both have cache memories (but non-coherent on SC2 side). The SW26010 processor of Sunway Taihulight also have a single physical memory shared by MPE (management processor element) and CPE (computing processor element). MPE and CPE share the same ISA. However, CPEs do not have data cache but local memories. In addition, processor architectures are also all different. NVIDIA Volta have relatively small number of "Streaming Multiprocessors", each with a number of floating-point arithmetic units. PEZY-SC2 is an MIMD processor with 2048 SC2 cores and 6 MIPS64 cores, and each of SC2 cores has one floating-point units. An SW26010 processor has four "core groups", each with one MPE with data cache and 64 CPEs with local memories. A unique and very important feature of CPEs is that they are organized as an 8×8 grid. Within each rows of columns of this grid, both of point-to-point communication and broadcast are supported.

Table 1 Performance parameters of modern HPC systems

System	Processor	memory B/F	network B/F
K	SPARC64 VIIIfx	0.5	0.12
Fugaku	A64fx	0.34	0.013
GYOUKOU	PEZY SC2	0.03	0.0005
Taihulight	SW26010	0.03	0.003
Summit	NVIDIA V100	0.12	0.0006

Thus, programs developped for machines with different processors, such as K/Fugaku, GYOUKOU, TaihuLight and Summits can easily become very different, because (a) programming environments are different (usual OpenMP, OpenCL, Athread and Cuda), (b) memory structures are all different, and (c) ISA and many other things are different.

With such a wide variety of both hardware and software, it is now very difficult for one research group to develop efficient programs for different architectures. Thus, almost all widely used programs for HPC are optimized for Intel x86 processors, with some support for NVIDIA GPGPUs but nothing else, and porting them to other architectures and achieve reasonable performance requires years or work of a large group of researchers.

This situation is clearly not ideal. From the viewpoint of developers of large-scale simulation sofware, it is unpractical to develop and maintain multiple versions optimized for very different architectures, and concentrating on one or two most widely used ones makes perfect sense. From the viewpoint of developers of processors and HPC systems, it is necessary to change the architecture in many different ways to improve the performance. Thus, the "best" strategies are completely different.

The companies which develop the processors understand this problem very well, and try to port and optimize as many important application programs as possible, and in some cases such effort proved to be successful. The problem of this approach is that it requires huge amount of resources, in particular human resources, much more than what is necessary to develop hardware. The success of an architecture depends primarily on the amount of human resources available for application development.

1.2 Portability by language?

In principle, if we could develop portable application programs, which can achieve high efficiency on various architectures without significant rewriting, that would be the solution for this problem. The traditional approach in this direction is the standard language and/or libraries for parallel execution. Unfortunately, though the parallel languages have been and still is the target of the active research, they have never widely used with a handful of exceptions such as CM-Fortran (and later HPF). CM-Fortran was the language for TMC

CM-2 and CM-5, and HPF was one of the practical choices on distributed-memory vector-parallel machines such as Earth Simulator.

The problem with parallel language like HPF is that it's parallel operation is defined at the level of arithmetic operations on elements of distributed arrays. Thus, the generated machine code tends to loop over large arrays, and there is no easy way to make use of the cache hierarchy.

Modern approaches are limited to parallelization within one node, and there are several proposed open standards such as OpenCL, OpenACC and OpenMP. OpenCL is designed for the host-accelerator architecture with separate memory spaces, and the data transfer between the host and the accelerator is controlled explicitly by the application programmer. Both OpenACC and OpenMP support both shared- and separate-memory architectures, and rely on directives to implicitly specify the data transfer. Here, again, even when the application program is written with an open standard supported by different hardware platforms, to achieve high efficiency with a "portable" code is difficult, since in order to achieve high efficiency architecture-specific optimizations are necessary.

1.3 Portability by library/framework

We have been working on a different approach. Instead of trying to provide a solution for all application areas, we limit ourselves to particle-based simulation programs, since they have a sufficiently wide range of actual applications in many areas of science and engineering. In many particle-based applications, the interaction between particles can be expressed as

$$\mathbf{F}_i = \sum_j^N \mathbf{f}(\mathbf{u}_i, \mathbf{u}_j), \tag{1}$$

where N is the number of particles in the system, $\mathbf{u}_i$ is a vector which represents the state of particle i, $\mathbf{f}$ is a function which describes the contribution of particle j to the total "interaction" on particle i, and $\mathbf{F}$ is the total interaction used to update the state of particle i.

In the case of the gravitational N-body simulation, $\mathbf{u}_i$ contains position, velocity, mass, and other parameters of particle i, $\mathbf{f}$ is the gravitational force from particle j to particle i, and $\mathbf{F}_i$ represents the gravitational acceleration on particle i from all other particles in the system.

Our software, which we call FDPS (Framework for Developing Particle Simulator[11]), provides functions necessary for large-scale parallelization of particle-based simulations. The basic functions are

- domain decomposition
- exchange of particles between nodes
- interaction calculation

FDPS is designed as a template library in C++ language, so that it can accept the data structure of particle as a class definition and the function to calculate interaction between particles in written in C++. We have extended the FDPS so that now the struct in C language and interaction function written in C can be used. Thus, it is now possible to use FDPS from any language which can call functions written in C.

A parallel particle-based simulation program developped using FDPS works roughly as follows

1. Initial setup, including the generation of the initial condition for particles. Particles are stored in the usual array of particles allocated in the user memory. At this moment, particles can be in arbitrary MPI processes.
2. Generate the domain decomposition by calling FDPS functions.
3. Exchange the particles so that they are in their appropriate MPI process by calling FDPS functions..
4. Evaluate the interaction between particles by calling FDPS functions..
5. Update the data of particles and evaluate other necessary quantities such as total energy and other diagnostics, next times etc.
6. Go back to step (2) if the termination time has not been reached.

Note that operations other than the three operations listed above are done in the usual user-written code. Thus, I/O, time integration, diagnostic output are all done in the user code, to which data of particles are accessible as array elements.

At first sight, the requirement that the problem should be written in the form of equation (1) might look too restrictive. Contributions from different particles must be linearly added, and interactions on all particles should be calculated at each step. In practice, these are not a severe restriction, since many problems for which large-scale parallel calculation is currently applied satisfy the above conditions. FDPS is now being used by many researchers to develop their own high-performance parallel codes for their problems.

In the rest of this paper, we discuss what should be done to achieve high efficiency for particle-based simulations on modern and future HPC systems, based on our experience on evaluating and optimizing FDPS-based programs on TaihuLight, GYOUKOU, and NVIDIA GPGPUs. As discussed earlier, the potential bottlenecks for the performance are the bandwidth of main memory, interconnect and host-accelerator connection. In section 2, we discuss the cost of internode communication and possibilities to further reduce them. We will see that the limiting factor for the scalability to very large number of processes is that there are operations whose cost *increase* as we increase the number of processes p. Their cost is negligible when p is small, but becomes dominant when p is large, in particular when it becomes much larger than the number of particles per node n. We discuss if we can make the cost of operations less than $O(p)$, ideally $O(1)$ but in the worst cases $O(p^{1/d})$, where d is the number of dimensions. In section 3, we discuss the limiting factors which come from the node architecture, such as the main memory bandwidth and host-accelerator communication bandwidth, and overview the algorithms we developped and

implemented. In section 4 we overview the achieved performance on several systems. Section 5 is for summary and future directions.

2 Reduction of the communication cost

In this section, we discuss the communication cost of parallel particle-based simulation code. We first consider the standard procedure used in FDPS, in which the Barnes-Hut tree algorithm is used, and we discuss mainly the long-range interactions. For simplicity, we assume that the distribution of particles is not too far from uniform. One advantage of FDPS is actually that it can handle highly inhomogenous particle distributions without significant increase in the calculation time, because of its use of the adaptive tree structure and adaptive domain decomposition. Thus, estimation of calculation time under the assumption of uniform particle distribution is not too bad.

2.1 Standard procedure

As discussed in Introduction, the basic steps for parallel particle-based simulation code on distributed-memory parallel platforms are

1. Domain decomposition
2. Particle exchange
3. Interaction calculation
4. Miscellany operations such as I/O, time integration and so on.

In the following, we overview the algorithms and performance characteristics of these procedures.

2.2 Domain decomposition

FDPS uses the 3D multisection algorithm [12] modified to achieve better load balance [7]. The multisection algorithm is the generalization of the orthogonal recursive bisection (ORB[17]). As its name suggests, ORB divides the computational domain recursively to two subdomains, in x, y, and z directions. In the multisection method, the division in one dimension can be of arbitrary size, and we divide the domains only once per one dimension. Thus, instead of recursively divide domains $\log_2 p$ times, where p is the number of MPI processes, we divide the domains three times for three-dimensional tree. The main advantage of the multisection method is that it can be used for number of MPI processes not an integer powers of two.

Early versions of FDPS uses the sampling method [4] to determine the coordinates of subdomains. In the sampling method, all MPI processes send randomly sampled subsets of their particles to the rank-0 process (or root process), and the rank-0 process performs necessary calculations to determine the coordinates of subdomains.

This sampling method works fine for thousands or even tens of thousands of MPI processes, or as far as the number of particles in one MPI process is significantly larger than the number of MPI processes. Since the root process receives sampled particles from all other processes, if the number of particles per process becomes less than the number of MPI processes, the amount of data the root processes receives becomes more than the amount of the particle data itself of the root process, and the time for the internode communication would become visible.

In the current version of FDPS, we adopted the hierarchical sampling [11, 10] in which the communication cost is $O(p^{1/d})$, where d is the number of spatial dimensions. The basic idea here is to re-sample particles at each dimension so that at each dimension the number of sample particles received is proportional to $p^{1/d}$.

Even with this scheme, the communication cost is still $O(p)$, where p is the total number of MPI processes, since all processes receives the physical dimensions of subdomains and for very large number of process, this $O(p)$ term can be the limitation of the scalability.

We therefore need a new algorithm in which the amount of data received by each process is less than $O(p)$. If we keep using the 3D multisection algorithm, one possibility is that each process has only one division data in each dimension. The division in x direction is shared by all processes, but that in y direction is shared only by processes in the same x coordinate in the processor grid, and z direction only by in the same x-y coordinates. This change reduces the amount of data received by each process from $O(p)$ to $O(dp^{1/d})$, where d is the number of dimensions. With this change, however, each no longer knows the physical dimensions of other processes, and cannot directly determine the destinations of particles moved or the LET. This scheme is not implemented yet in FDPS but in the following we also discuss how algorithms should be changed and how the communication cost scales with this scheme.

2.3 Particle exchange

The standard algorithm used in FDPS to exchange particles is simple and straightforward. After the new coordinates of all subdomains are determined and shared by all MPI processes, each process determines for each particle if it is still in its subdomain or not, and if not, which process it should be sent. Then with one call to MPI_alltoall, each process tells all other processes how many particles it wants to send, and then using MPI_alltoallv sends actual particle data. This scheme works fine when the number of MPI processes is small, and works reasonably well for around 100K nodes (K computer) if (a) the hardware vendor provides rich interconnect and fast implementation of alltoall(v) and (b) calculation time for one timestep is large, order of several seconds or more. However, we want to solve problems of practical size on big machines, and thus the calculation time per one timestep should be much shorter, much less than one second. However, even with ideal implementation the throughput of the

alltoall operation is determined by the bisection bandwidth, which does not scale on very large machines. Moreover, the actual performance of alltoall for short messages can be extremely low, while the total amount of data need to be sent/received is large simply because the number of MPI process is large. Thus, it has become critical to eliminate alltoall(v) operation completely.

For the particle exchange operation, though in principle particles can move from any node to any other node, in practice the actual movement is between subdomains which are physically nearby, and thus with our multi-section method between nearby nodes in the three-dimensional process grid. Our current implementation makes use of this fact. For each of six directions $(\pm x, \pm y, \pm z)$, we first determine the largest distance particles move in the process grid, and then we loop over all possible relative displacement within the process grid. If the maximum distance is independent of the total number of the processes, the communication time would not increase when the number of processes increases.

There are many other possible implementations of the particle exchange which might be more efficient on some types of interconnect. For example, consider the following algorithm. If process (i_1, j_1, k_1) wants to send its particles to process (i_2, j_2, k_2), it first sends its data to process (i_2, j_1, k_1). Similarly, all processes sends all their data to $\pm x$ directions, so that all data transfers in the direction of the x axis are finished. Then all processors send data in y direction, and then in z direction. In practice, on machines with direct torus network such as K and Fugaku, we should consider sending data in multiple directions in parallel, to make more efficient use of the communication links. With this algorithm the total number of point-to-point communications is at the maximum $O(p^{1/3})$ and is usually much smaller, and thus fairly good performance might be achieved.

When the scalable implementation of the domain decomposition in which each node does not have all information of subdomains is used, the above algorithm would still work without problem.

2.4 Interaction calculation

In FDPS (and also in many other parallel particle-based simulation codes), Barnes-Hut tree algorithm[2] is used for the calculation of long-range interaction. In some codes FMM[6] is used, but as far as the parallelization strategy and algorithm to exchange the necessary data between processes are concerned, there is no essential difference between the tree algorithm and FMM. Therefore, what is discussed here applies to FMM as well. In fact, we are currently working on the integration of a variation FMM into our FDPS framework.

The parallel version of Barnes-Hut tree algorithm used in FDPS consists of following steps

1. Each process constructs its local tree.
2. Each process (process i) determines, for each of all other processes (process j), the information of the tree of process i that process j needs to evaluate

the total interaction on particles in process j. This can be done locally since each process already know the geometry of other subdomains. This information is called "local essential tree", or LET [20].
3. Each process receives the LETs from all other processors.
4. Each process constructs the "global" tree using LETs it received.
5. Each process evaluates the interaction on its particles using the global tree.

We can see that the internode communication takes place only at step 3. In the following, we discuss the several possible implementations of step 3 and their performance characteristics.

A simple and straightforward implementation would just use MPI_alltoallv for LET exchange. As in the case of particle exchange, alltoall communication should be avoided since even with rich interconnect and highly optimized software, it would be limiting factor of the performance if the number of MPI processes is not much smaller than the number of particles in one MPI process.

Unlike in particle exchange, where processes would actually exchange particles only with nearby processes, in LET exchange all processes have to receive some information from all other processes. Thus, at first sight alltoall communication looks unavoidable. However, actually there are several approaches to avoid it. The following is the list of known schemes.

a) Use TreePM
b) Use allgather for distant communication
c) Construct higher-level inter-process tree

We briefly discuss each of them below.

2.4.1 TreePM

When the periodic boundary condition is used, FFT is the natural way to solve the Poisson equation, and the TreePM algorithm[1] has been widely used. The basic idea of the TreePM method is to replace the particle-particle part of the P^3M scheme[5] by the tree algorithm, so that the calculation cost would not increase significantly when the distribution of particles becomes highly inhomogeneous. Also, since the PP part is replaced by the tree algorithm, the calculation cost of the tree part depends only weakly on the cutoff radius, and thus the necessary number of grid points for FFT is relatively small.

Consider the case where we have p processors, with n particles each, and use g grid points per process for FFT. Theoretically, the communication efficiencies of both the LET exchange and FFT are limited by the bisection bandwidth, and the amount of data passed in one dimension is $O((np)^{2/3})$ and $O(pg)$. Therefore, to make the cost of FFT smaller than LET exchange, g should satisfy

$$g \leq \frac{n^{2/3}}{p^{1/3}}. \tag{2}$$

Since we are dealing with very large systems with 10^5 or even 10^6 MPI processes, for most of practical applications $n < p$, and thus g should be very small. The fact that g is small implies the cutoff length is significantly larger than the size of subdomains assigned to MPI processes, and thus the processes should still communicate with a fairly large number of processes. Even so, the alltoall communication can be eliminated.

2.4.2 Use allgather for distant communication

The idea here is that when a very large number of processes is used, many subdomains would actually be regarded as points (or single multipole expansions). Thus, even in principle the LETs one process should send to other nodes are different for different nodes, many of them receive the single expansion, which is the same for all of them. Thus, we can replace most of the transfers of LETs by single allgather operation, through which each node receives the top-level multipole expansions of all other nodes. Then, each node send LETs only to nodes which require lower-level tree structures. Since in our current algorithm all processes know the physical dimensions of all other processes, we can use usual send/receive pairs to guarantee that all data are exchanged correctly.

With this scheme, only global communication is through allgather, and thus its bandwidth is not limited by the bisection bandwidth but by the injection bandwidth, and thus the scaling in the case of ideal implementation is much better. In the case of naive alltoall implementation, the necessary time is $O(p^2/b_{\mathrm{bisect}})$, where b_{bisect} is the bisection bandwidth. With allgather, it is $O(p/b_{\mathrm{inject}})$, where b_{inject} is the effective bandwidth of the network port of a single MPI process. Thus, even when we have the full-bisection network where $b_{\mathrm{bisect}} = pb_{\mathrm{inject}}$, the allgather-based implementation would show comparable or better performance than alltoall-based, and on large machines with $b_{bisect} \ll pb_{inject}$, the allgather-based algorithm is much better.

However, this algorithm still has the problem that the amount of the data received by each process is $O(p)$ and the cost of tree construction is $O(p \ln p)$. If $p \gg n$, which is likely the case for large systems, the single allgather operation can be the limiting factor of the scalability, as in the case of the domain decomposition. We will address this issue in the subsections below.

2.4.3 Construct higher-level inter-process tree

As seen in section 2.4.2, the problem with the LET scheme is that when the inter-process communication takes place, there is no information concerning the structure higher than those of local trees, and that information of higher levels of the tree is constructed at each process redundantly. Thus, if we construct the global higher-level structure and let each process access them, in principle the amount of data received by each process can be reduced.

With the multisection method, it is not clear how we can make higher-level tree. One possibility is to map eight neighboring subdomains to one higher-level tree node and apply that process recursively to top level.

One problem with this approach is that if the data of one tree node in this higher-level tree exists in one MPI process, that process will be accessed by many processes, and the communication bottleneck is not removed. This can be avoided if we use FMM or at least FMM-like communication pattern, in which tree nodes communicate only with the nodes in the same level. for all nodes the amount of data transfer can be independent (or only $O(\log p)$). In this scheme, LET is built at each level of the tree, and the information of the distant nodes is not directly sent but sent through high-level nodes.

2.4.4 Combination with new domain decomposition scheme

If we use the scalable algorithm for the domain decomposition discussed in section 2.2, we need to modify the LET based scheme, since one process no longer has the geometries of other nodes. We can use multi-stage LET algorithms, in which the LETs are constructed step by step.

First, each process construct LETs, not for each processes but for all "slabs" along the x directions (slabs extends to y and z directions). Now each node (i, j, k) has LETs from node $(1, j, k)$ to (n_x, j, k), where n_x, n_y, and n_z are number of divisions in x, y, and z directions, respectively. Then each node constructs the tree from both its local particles and received LETs. Here, the opening criterion of the tree cells must be for the slabs and not for its subdomain. Now, each slab has the information of the entire system. So the communication in x direction will not occur. In the second stage, we will do the same thing for "columns", created by cutting the slabs in y direction, and do the same thing: construct LETs for all other columns in the same slab and send them, and reconstruct the tree by adding new LETs. Finally, do the same thing in z direction. After this procedure each process has the information of entire system.

This scheme does remove the $O(p)$ part of the LET algorithm and reduce the total number of communications from $O(p)$ to $O(dp^{1/d})$. One drawback of this scheme is that the tree need to be constructed four times per timestep: initial local tree and communications in x, y, and z directions. However, the dominant part of the calculation cost of the tree construction is sorting, and for global trees the sorting cost can be made practically $O(n)$ since the local particles are already sorted, or even smaller if we keep the local tree and tree for the global information separate. Therefore, we believe the additional cost is acceptable and the gain by removing the $O(p)$ communication is larger.

3 The effect of node architecture

In this section we discuss the limiting factors of the node-level performance of particle-based codes on present and near-future machines and new algorithms to improve the performance. The factors we need to consider include the following factors.

1. Main memory bandwidth for sequential and random accesses.

2. In the case of accelerators, communication bandwidth between the host and the accelerator.
3. Also in the case of accelerators or heterogeneous architecture, the ratio of host performance and accelerator performance.

In the following we discuss these factors.

3.1 Main memory bandwidth

For many large-scale simulation codes, the effective main memory bandwidth tends to be the limiting factor of the efficiency. Let us first derive the theoretical minimum necessary bandwidth to achieve a reasonable efficiency for particle-based simulation codes, assuming that we are using explicit time stepping,.

As in section 2, we consider the procedures for domain decomposition, particle exchange, and interaction calculation using Barnes-Hut tree algorithm.

3.1.1 Domain decomposition

As far as the memory access is concerned, this part is negligible since the calculation time here is dominated by the communication bandwidth which is much smaller than the memory bandwidth.

3.1.2 Particle exchange

For all particles we need to check if it is still in its subdomain. Thus, we read the data of all particles a least once per timestep.

3.1.3 Interaction calculation

As discussed in section 2.4, the LET-based parallel Barnes-Hut tree algorithm with Barnes' vectorization for interaction calculation consists of the following steps:

1. Local tree construction
2. LET exchange
3. Global tree construction
4. Creation of the interaction lists
5. Interaction calculation using the interaction lists

The dominant part of the tree construction is the sorting of particles using keys. With any algorithm, we need to read each particle at least once to make its key, and write once to store the sorted result. With our current implementation the cost of sorting is much higher [9], and that implies there is rooms to improve algorithms, We ignore the cost of LET exchange, since that part is smaller compared to the actual interaction calculation.

The cost of the construction of the interaction lists depends on many factors, in particular the choice of the parameter n_g, the maximum number of particles to share the same interaction list. We should chose this parameter so that the total calculation time is minimized, and that means the optimal choice of this parameter depends on the processor architecture. To make rough estimate, let us assume that the length of the interaction list is $n_0 + n_g$, where n_0 is a constant which depends on the required accuracy, and the ratio between the calculation time of one interaction and that for adding one entry in the interaction list is X. The total cost of interaction list and interaction calculation per particle in unit of the time for single interaction calculation is then given by

$$C = X(n_0 + n_g)/n_g + n_0 + n_g, \tag{3}$$

and the optimal value of n_g is given by

$$n_g = \sqrt{Xn_0}. \tag{4}$$

If we assume that the data of one particle is around 30bytes and single interaction calculation 30 floating-point operations, we have

$$X = 1/b, \tag{5}$$

where b is the B/F number for the random access of the main memory, which on modern machines around 1/30 or less of the B/F number for the sequential access. Thus, for a machine with B/F=0.03, we have $X = 1000$, and that means the optimal value of n_g would be around 1000. This is, however, quite a bit too large, since such a large value of n_g results in the significant increase of the total calculation cost. We thus need some new approach to reduce the main memory access.

One method to reduce the main memory access is to use the same tree structure and interaction lists for multiple time steps. This method, which we call the reuse method, turned out to be quite effective on all of modern platforms we have so far tested, and helped us to achieve near-peak performance on all of them.

With the reuse method, the necessary memory access per particle are one for local tree (momentum update), two for global tree (reordering and /momentum update), and a few times for the construction of the interaction list and interaction calculation. Thus, the amount of memory access per particle per timestep is less than ten times, and most of them are fast, near-sequential access. Thus, the ratio between the memory access time and the calculation time is now given by

$$R = \frac{10}{bn_0}. \tag{6}$$

Since n_0 is of the order of 1,000, roughly speaking the reuse method would work for systems with $B/F > 0.01$. Right now all HPC processors provide $B/F > 0.02$, but in near future we might see machines with $B/F < 0.01$, and how we can make use of such systems is an important question.

The absolute minimum of the amount of the memory access necessary per timestep per particle is one read and one write. Therefore, there is still rooms of improvement by a factor of five.

The reuse method is quite effective when we can use it. However, there are physical systems for which the reuse method does not work. How we can minimize the main memory access is an important research direction for such systems. One possibility is to make better use of on-chip memories. If we divide the systems not just by the number of processes but to smaller blocks which can fit into on-chip memories of processors, in principle we can reduce the main memory access by a large factor by processing one block at a time. In a naive implementation we would still need one read for particle exchange and another read for LET, and yet another read/write for the interaction calculation and time integration. Thus, we could reduce the memory access to three reads and one write per particle per timestep. Such implementation would be advantageous on machines of near future.

3.2 Host-Accelerator communication

The standard way to use accelerators in FDPS or other large-scale parallel particle-based simulation codes is to use them only for the interaction calculation using the interaction lists. The amount of data transferred between the host and accelerators is essentially one read and one write per timestep per particle, since interaction list can be the list of the indices of particles and thus its total size in bytes is smaller than that of particles. Very roughly, unless the interaction is of very short-range nature and inexpensive, for host-accelerator interface the necessary B/F number is around 0.002. Unfortunately, interface bandwidth of modern GPUs are approaching to this value. For example, NVIDIA Tesla V100s, used with PCIe gen3 interface, has the interface B/F number of around 0.003.

If all particles of one process, or those processed by one accelerator card, can fit to the memory of the accelerator card, we can in principle reduce the necessary interface bandwidth in several ways, at least when the reuse method is used.

When we use the reuse method, it is straightforward to move all operations except for communications in reuse steps to the accelerator side, since they can be expressed in simple loops without conditional branches. In this case, the necessary communication is only for LETs, and if the number of particles per accelerator card is large enough, we can reduce the necessary bandwidth of host-accelerator communication by a large factor.

Another possibility is to use some form of data compression for the host-accelerator communication. Since the necessary bandwidth of the host-accelerator communication so small, we can pay fairly large calculation cost for data compression and restoration. On the other hand, efficient compression is not easy for the data from numerical simulation, in particular when it is being used for simulation. One possibility is to construct the same “predictor” on both side,

and to send only the correction terms. Whether or not such a strategy works or not depends very strongly on the nature of the system and the methods used for the interaction calculation and time integration. When we use the reusing method, we can expect that the orbits of particles are generally smooth and predictable, since otherwise the reuse method would not work.

In the case simulation of galaxy formation, for typical galaxies, we will use the timestep of around 10^5 years for particles which represent stars and gas. Their dominant motion is the orbital motion within the galaxy with the timescale of 10^8 years. Thus, if we can construct second-order prediction polynomial, its local relative truncation error is around 10^{-9}. For the interaction calculation, this accuracy might be already sufficient and we may be able to send only a small number of bits as the correction terms of the prediction, and we should be able to apply a similar procedure to the calculated interaction. Also, similar consideration can be applied to other systems like planetary rings and protoplanetary nebulae.

In prediction we are using the data redundancy in the temporal domain. It should also be possible to use the redundancy in the spatial domain. For example, the difference of Morton keys of near neighbors is usually small, and thus we can compress the sequence of sorted Morton keys by taking the difference of two consecutive keys and use variable-length integer format, or by using any general-purpose compression algorithms. The advantage of this algorithm is that it does not require that the orbits of particles are smooth.

3.3 Host-Accelerator Performance Difference

We can regard both of machines with host-accelerator architecture and heterogeneous manycore architecture as consisting of small numbers of general-purpose cores and large number of specialized cores. In the case of GPGPU systems, the general-purpose side is usually Intel x86 processors, and the difference of their theoretical peak performance is usually not very large. Typically around ten or so and only in some extreme systems the ratio exceeds 30. In the case of heterogeneous many-core systems, This ratio is generally much larger, such as 64 for Sunway 26010 processor, which has 64 CPEs per MPE. When we ignore the limitation from the host-accelerator communication bandwidth, if this ratio is not very large, like ten or less, it makes sense to use accelerator only for the interaction calculation, since the calculation cost of the rest of the calculation is not very large and it can be performed on the host computer without affecting the overall efficiency too much. However, if this ratio is very large, such as a factor of 100 or more, practically all calculations must be done on the accelerator side, even when we use the reusing method.

In many cases, we need to rewrite the application program to move part of calculations from host to accelerator, or from general-purpose cores to specialized cores. With frameworks like FDPS, we can hide these architecture-specific codes in the FDPS framework and keep the application program machine in-

Table 2 Optimizations used

	Taihulight	GYOUKOU	V100
$O(p^{1/d})$ domain decomposition	Yes	Yes	Yes
Higher-level tree*	Yes	Yes	-
LET exchange without alltoallv	Yes	Yes	(Yes)
Accelerator for all calculations	Yes	Yes	No
Interaction list reuse	Yes	Yes	Yes

dependent. At present, we have not yet reached this goal, but we are working in this direction.

4 Achieved performance

In this section, we briefly discuss the achieved performance of FDPS-based applications on three platforms: Sunway Taihulight, GYOUKOU, and NVIDIA V100 GPGPU. For the first two machines, the performance numbers are obtained with several machine- and problem-specific optimizations not included in the public release of FDPS yet. The code used for NVIDIA V100 is in github[1]. The operation of GYOUKOU was terminated on March 31st, 2018. We have made further optimization of our calculation code and measured the final performance numbers on a smaller system, Shoubu B, with 512 PEZY-SC2 chips.

4.1 Test problems and optimizations used

For Taihulight and GYOUKOU, we developped the code for large-scale simulations of planetary rings, such as Saturn's ring. The realistic simulation of inner rings of Saturn can be done using 10^{11-12} particles, and such simulation is becoming feasible with the peak speed of 100 PF or more. For NVIDIA V100, we report the single-card performance for the simple three-dimensional cold collapse problem.

Table 2 shows the algorithms used on each platform. Not all algorithms discussed in sections 2 and 3 have been actually implemented. The domain decomposition without $O(p)$ communication is not complete yet, since the current implementations for Taihulight and GYOUKOU are specialized to simulations of planetary rings. Thus, improvements related to this change are not available in the released versions of FDPS yet. However, LET exchange without alltoallv is available in the current FDPS. For the benchmark on V100 this feature is not used since we measured single-node performance only. Also, in the code for Taihulight and GYOUKOU, calculations other than interaction calculation using the interaction list are moved to accelerator (CPE in the case of Taihulight and PEZY-SC2 processors on GYOUKOU). These porting are

[1] https://github.com/FDPS/FDPS

Table 3 Breakdown of calculation time

System	# processes	interaction	comm.	others
TaihuLight	160000	2.31	0.090	0.476
Gyoukou	8192	0.453	0.147	0.222
Shoubu B	512	0.360	0.030	0.135

relatively easy but requires machine-specific codes, since we need to make use of special features of these processors to achieve acceptable performance. Such porting is a bit difficult on GPGPU and has not been done yet.

4.2 Performance Results

Detailed performance and scalability numbers are given in [9,10]. Here we present "best" numbers achieved so far. On Taihulight, the measured performance for the run with 1.6×10^{12} particles on 160k processes (40000 nodes) of TaihuLight is 47.9 PF, or 39.7% of the theoretical peak performance. On PEZY-SC2 based systems, we achieved 10.6PF for 8×10^9 particles on 8K SC2 chips, or efficiency of 23.3% of the theoretical peak performance. On 512-chip Shoubu System B, we achieved the speed of 1.01 PF, or 35.5%. In all cases, the number of particles per MPI process is 10M. The calculation time per timestep is 2.88, 0.822 and 0.525 second, on Taihulight, GYOUKOU, and Shoubu B. Note that the difference of performance between GYOUKOU and Shoubu B is not due to the scalability limitation but purely due to the difference in the calculation code used. As we stated earlier the operation of GYOUKOU was terminated before we completed the optimization.

Table 3 shows the breakdown of calculation time. On both platforms, we can see that the interaction calculation dominates the total calculation time, and the fraction of time spent for communication is relatively small, less than 1/10 of the total time. We should admit that part of the reason why the communication time is small is that we simulate planetary rings which is very thin, and thus communication is effectively limited to x and y directions. Thus, the surface area of one subdomain, which is usually proportional to $n^{2/3}$ where n is the number of particles per domain, is in this case proportional to $n^{1/2}$, and the amount of communication is small. However, even so, efficient calculation on these machines would have been impossible without the new algorithms discussed in previous sections. Thus an important guideline for extreme-scale parallel calculation is to avoid any global communication or anything whose calculation time is $O(p)$, even if the coefficient is very small. In particular, the use of communication pattern of alltoall(v) must be eliminated since it will result in $O(p)$ execution time, even when the message size is very small, and should be replaced with communications with execution time at the maximum $O(p^{1/d})$ or ideally $O(1)$.

In terms of the number of particles integrated per second, we have achieved 5.5×10^{11} particles per second on Taihulight, which is more than 10 times faster than the results of previous works on K computer[8] or ORNL Titan [3]. The

Table 4 GPGPU performance

Algorithm	time per timesteps(sec)
CPU only	0.98
GPU without index	0.36
GPU with index	0.42
GPU with reuse	0.095

number of particles used is similar to that used on K computer[8], and that means the calculation time per one timestep is reduced by more than a factor of 10.

Table 4 shows the calculation time per timestep for the cold collapse calculation with 4M particles on single NVIDIA V100 card with Intel Xeon E5-2670 v3 host CPU. Here, "GPU without index" denote the use of interaction list with physical quantities of particles in the list. Thus, communication cost is large. With "GPU with index" algorithm, the data of particles are sent only once per timestep. Thus, the communication time is reduced, but the calculation time becomes somewhat longer. With reusing, however, the calculation time becomes much shorter. The number of reusing in this case is 16. In this case, the effect of reusing is not the reduction of the communication cost but mostly the reduction in the calculation cost of operations still done on CPU. The achieved performance is around 40M particles per timestep. Currently, we are sending particles for tree data (x,y,z and mass) and particles which receive force (x,y,z only) all in single precision. Thus, the data sent per particle is 40 bytes. Thus, the data sent to GPU in one second is only 1.6GB, while the bandwidth of CPU-GPU connection is close to 15GB/s.

5 Summary and Future directions

5.1 Lessons Learned

In this paper, we overviewed the current status of our effort to make large-scale modern HPC platforms usable for large-scale particle-based simulations. The difficulties include extremely large number of cores and MPI processes, small memory bandwidth, even smaller network bandwidth. Also, in the case of accelerator architecture and/or heterogeneous many-core architectures, very large ratio of performance of accelerator cores and general-purpose cores, and small host-accelerator communication bandwidth.

What we have observed is that it is not impossible to design the framework, not a specific application, for particle-based simulations so that the applications developed using that framework can achieve high efficiency on modern HPC platforms.

In order to deal with the extremely large number of cores, it is the most important to eliminate global communications, in particular those with the $O(p)$ term in the communication cost. Examples of such communications are

MPI_alltoall(v) and MPI_allgather(v). This means no single process should communicate with all other processes, even if it is the rank-0 node.

In our case, such $O(p)$ communications were originally used in domain decomposition, particle exchange and LET exchange. We have replaced them with $O(p^{1/d})$ communications.

Compared to mesh-based simulations, where the memory bandwidth tends to be the bottleneck, particle-based simulations do not require very high memory bandwidth, since the calculation cost of particle-particle interaction is large and thus calculation is not very memory intensive. Even so, with B/F numbers of 0.01 or less, we need new approaches including the interaction list reuse. With the reuse algorithm, the number of memory access per particle per timestep is still of the order of ten. On the other hand, the lower bound of the main memory access is one read and one write per timestep. Therefore, it is at least theoretically possible to reduce the necessary memory bandwidth by another factor of 10, so that we can use machines with B/F ~ 0.001.

To reduce the necessary communication bandwidth is also quite important. The necessary communication bandwidth, however, depends strongly on the target physical systems and difficult to provide universal solutions. Solutions specific to physical systems and also to network architecture of the machine will be necessary.

First let us consider the communication due to migration of particles. For a nearly uniform distribution of particles, we can expect that particles move a small fraction of the average interparticle distance in one timestep. Thus, the number of particles which migrate from one subdomain to other (usually neighboring) subdomains is $O(p^{2/3})$ with relatively small coefficient, and that means the required network bandwidth is much smaller than the required main memory bandwidth. However, in many astrophysical simulations the situation is quite different.

For example, systems like planetary rings, protoplanetary disks, disk galaxies have the nature that the global rotation velocity is much larger than the local velocity dispersion. Thus, if we have the domain geometry fixed to the inertial frame, particles moves the distance much larger than the typical interparticle distance, and with very large number of domains, it can occur that all particles in one domain moves to other domains at every timestep. It is clear that the network bandwidth would limit the scalability. In our simulation of planetary rings discussed in section 4, we introduced rotating reference frame so that the circular motion of particles do not cause migration of particles. In addition, we adopted the domain geometries defined in cylindrical coordinates. This approach is quite effective for narrow rings, where the range of the angular velocity is small. However, for wider rings or disks, this strategy does not work since the angular velocity can be quite different. If the topology of the interprocessor network is mesh, there is no simply way to reduce the communication. However, in the case of networks with fat-tree or similar topologies, we can reduce the communication by let each ring of domains, rotate at their local rotation velocities. In the case of the fat-tree network, it is possible to

maintain the bandwidth of communication between two rings of domains with different rotation speeds, by assigning low-level trees to rings.

The communication for interaction calculation is currently more expensive than that for the migration of particles. However, here problem-independent strategies such as data compression will be quite effective.

5.2 Future directions

As we have summarized in the previous subsection, in order to realize particle-based simulations with high efficiency on future machines, we will have to further reduce the necessary bandwidths of main memory and inter-node network. For the main memory bandwidth, more efficient use of the on-chip memory (either cache or local memory) will become more and more important. For network bandwidth, it will be necessary to investigate problem- and network-specific strategies.

Acknowledgements This work was supported by The Large Scale Computational Sciences with Heterogeneous Many-Core Computers in grant-in-aid for High Performance Computing with General Purpose Computers in MEXT (Ministry of Education, Culture, Sports, Science and Technology-Japan), by JSPS KAKENHI Grant Number JP18K11334 and JP18H04596 and also by JAMSTEC, RIKEN, and PEZY Computing, K.K./ExaScaler Inc. In this research computational resources of the PEZY-SC2 based systems supercomputer, developed under the Large-scale energy-efficient supercomputer with inductive coupling DRAM interface project of NexTEP program of Japan Science and Technology Agency, has been used.

Conflict of interest

The authors declare that they have no conflict of interest.

References

1. Bagla, J.S.: TreePM: A Code for Cosmological N-Body Simulations. Journal of Astrophysics and Astronomy **23**, 185–196 (2002). DOI 10.1007/BF02702282
2. Barnes, J., Hut, P.: A hiearchical o(nlogn) force calculation algorithm. Nature **324**, 446–449 (1986)
3. Bédorf, J., Gaburov, E., Fujii, M.S., Nitadori, K., Ishiyama, T., Zwart, S.P.: 24.77 pflops on a gravitational tree-code to simulate the milky way galaxy with 18600 gpus. In: SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 54–65 (2014). DOI 10.1109/SC.2014.10
4. Blackston, D., Suel, T.: Highly portable and efficient implementations of parallel adaptive n-body methods. In: Proceedings of SC97, pp. (CD–ROM). ACM (1997)
5. Eastwood, J.W., Hockney, R.W., Lawrence, D.N.: P3M3DP-the three-dimensional periodic particle-particle/particle-mesh program. Computer Physics Communications **35**, C–618 (1984). DOI 10.1016/S0010-4655(84)82783-6
6. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. Journal of Computational Physics **73**, 325–348 (1987)

7. Ishiyama, T., Fukushige, T., Makino, J.: GreeM: Massively Parallel TreePM Code for Large Cosmological N -body Simulations. Publications of the Astronomical Society of Japan **61**, 1319– (2009)
8. Ishiyama, T., Nitadori, K., Makino, J.: 4.45 pflops astrophysical n-body simulation on k computer: The gravitational trillion-body problem. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pp. 5:1–5:10. IEEE Computer Society Press, Los Alamitos, CA, USA (2012). URL http://dl.acm.org/citation.cfm?id=2388996.2389003
9. Iwasawa, M., Namekata, D., Nitadori, K., Nomura, K., Wang, L., Tsubouchi, M., Makino, J.: Accelerated FDPS — Algorithms to Use Accelerators with FDPS. arXiv e-prints arXiv:1907.02290 (2019)
10. Iwasawa, M., Namekata, D., Sakamoto, R., Nakamura, T., Kimura, Y., Nitadori, K., Wang, L., Tsubouchi, M., Makino, J., Liu, Z., Fu, H., Yang, G.: Implementation and Performance of Barnes-Hut N-body algorithm on Extreme-scale Heterogeneous Many-core Architectures. arXiv e-prints arXiv:1907.02289 (2019)
11. Iwasawa, M., Tanikawa, A., Hosono, N., Nitadori, K., Muranushi, T., Makino, J.: Implementation and performance of FDPS: a framework for developing parallel particle simulation codes. Publications of the Astronomical Society of Japan **68**, 54 (2016). DOI 10.1093/pasj/psw053
12. Makino, J.: A Fast Parallel Treecode with GRAPE. Publications of the Astronomical Society of Japan **56**, 521–531 (2004)
13. Phillips, J.C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R.D., Kale, L., Schulten, K.: Scalable molecular dynamics with NAMD. ournal of Computational Chemistry **26**, 1781–1802 (2005)
14. Plimpton, S.J.: Fast Parallel Algorithms for Short-Range Molecular Dynamics. Journal of Computational Physics **117**, 1–19 (1995)
15. Potter, D., Stadel, J., Teyssier, R.: PKDGRAV3: beyond trillion particle cosmological simulations for the next era of galaxy surveys. Computational Astrophysics and Cosmology **4**(1), 2 (2017). DOI 10.1186/s40668-017-0021-1
16. Pronk, S., Pll, S., Schulz, R., Larsson, P., Bjelkmar, P., Apostolov, R., Shirts, M.R., Smith, J.C., Kasson, P.M., van der Spoel, D., Hess, B., Lindahl, E.: GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. Bioinformatics **29**(7), 845–854 (2013). DOI 10.1093/bioinformatics/btt055. URL https://doi.org/10.1093/bioinformatics/btt055
17. Salmon, J., Quinn, P.J., Warren, M.: Using Parallel Computers for Very Large N-Body Simulations: Shell Formation Using 180 K Particles, pp. 216–218. Springer Berlin Heidelberg, Berlin, Heidelberg (1990). DOI 10.1007/978-3-642-75273-5_51. URL http://dx.doi.org/10.1007/978-3-642-75273-5_51
18. Salomon-Ferrer, R., A., C.D., C., W.: An overview of the Amber biomolecular simulation package. WIREs Comput Mol Sci (2012). DOI 10.1002/wcms.1121
19. Springel, V., Yoshida, N., White, S.D.: Gadget: A code for collisionless and gasdynamical cosmological simulations. New Astronomy **6**, 79–117 (2001)
20. Warren, M.S., Salmon, J.K.: Astrophysical N-body simulations using hierarchical tree data structures. In: Supercomputing '92, pp. 570–576. IEEE Comp. Soc., Los Alamitos (1992)