FDPS: A Novel Framework for Developing High-Performance Particle Simulation Codes for Distributed-Memory Systems

Masaki Iwasawa
RIKEN Advanced Institute for Computational Science
7–1–26,
Minatojima-minami-machi,
Chuo-ku, Kobe, Hyogo, Japan
masaki.iwasawa@riken.jp

Ataru Tanikawa
RIKEN Advanced Institute for Computational Science
7–1–26,
Minatojima-minami-machi,
Chuo-ku, Kobe, Hyogo, Japan
ataru.tanikawa@riken.jp

Natsuki Hosono
RIKEN Advanced Institute for Computational Science
7–1–26,
Minatojima-minami-machi,
Chuo-ku, Kobe, Hyogo, Japan
natsuki.hosono@riken.jp

Keigo Nitadori
RIKEN Advanced Institute for Computational Science
7–1–26,
Minatojima-minami-machi,
Chuo-ku, Kobe, Hyogo, Japan
keigo@riken.jp

Takayuki Muranushi
RIKEN Advanced Institute for Computational Science
7–1–26,
Minatojima-minami-machi,
Chuo-ku, Kobe, Hyogo, Japan
takayuki.muranushi@riken.jp

Junichiro Makino
RIKEN Advanced Institute for Computational Science
7–1–26,
Minatojima-minami-machi,
Chuo-ku, Kobe, Hyogo, Japan
jmakino@riken.jp

ABSTRACT

We have developed FDPS (Framework for Developing Particle Simulator), which enables researchers and programmers to develop high-performance particle simulation codes easily. The basic idea of FDPS is to separate the program code for complex parallelization including domain decomposition, redistribution of particles, and exchange of particle information for interaction calculation between nodes, from actual interaction calculation and orbital integration. FDPS provides the former part and the users write the latter. Thus, a user can implement, for example, a high-performance N-body code, only in 120 lines. In this paper, we present the structure and implementation of FDPS, and describe its performance on two sample applications: gravitational N-body simulation and Smoothed Particle Hydrodynamics simulation. Both codes show very good parallel efficiency and scalability on the K computer. FDPS lets the researchers concentrate on the implementation of physics and mathematical schemes, without wasting their time on the development and performance tuning of their codes.

Categories and Subject Descriptors

D.1.3 [**Software**]: PROGRAMMING TECHNIQUES—*Concurrent Programming*; I.6.7 [**Computing Methodologies**]: SIMULATION AND MODELING—*Simulation Support Systems*; J.2 [**Computer Applications**]: PHYSICAL SCIENCES AND ENGINEERING—*Astronomy*; J.2 [**Computer Applications**]: PHYSICAL SCIENCES AND ENGINEERING—*Earth and atmospheric sciences*

General Terms

Algorithm, Performance

Keywords

Framework for implementing parallel codes, high-performance computing, particle simulation

WOLFHPC2015, November 15-20 2015, Austin, TX, USA
ACM 978-1-4503-4016-8/15/11.
http://dx.doi.org/10.1145/2830018.2830019

1. INTRODUCTION

Particle-based simulations have been widely used in the field of computational science, and they are becoming more and more popular. In particle-based simulations, a system under study is regarded as a collection of mutually-interacting particles, or a particle system, and its evolution is described by the evolution (in many cases, motion) of individual particles. Examples of particle-based simulations include gravitational N-body simulation, molecular dynamics simulation, granular dynamics simulation, and element-free simulation of fluid dynamics or structural analysis. In the case of molecular dynamics and gravitational N-body simulations, the physical system under study is a collection of particles, and the particle-based method is a natural way to simulate the system. Element-free methods have advantages over structured (or unstructured) grid-based method, depending on the nature of the system and computer system to be used.

In order to improve the resolution and accuracy of particle-based simulations, it is necessary to utilize present-day HPC systems. However, to develop a calculation code for particle systems which can achieve high efficiency on present-day HPC systems is difficult and time-consuming. There are several reasons for this difficulty. In order to achieve high efficiency, we need to decompose the computational domain, assign subdomains to computing nodes, and redistribute

particles according to their positions. This decomposition should be dynamically changed to guarantee the good load balance. This division of the computational domain means that computing nodes need to exchange the information of particles to evaluate the interactions on their particles. To achieve high efficiency, the amount of the exchanged data must be minimized, and interaction calculation should also be efficient, making good use of cache memory and SIMD execution units. It is also important to make use of GPGPUs or other types of accelerators, if available.

Domain decomposition and load balance have been discussed in a number of works [30, 13, 37, 24, 21, 22]. The efficient use of SIMD units is discussed in [27, 34, 33], and GPGPUs in [18, 15, 16, 17, 7, 6].

Thus, to develop a code which has all of these necessary features for present-day HPC systems has become a big project which requires multi-year effort of a multi-person team. It has become difficult for researchers outside such a team to try any new experiment which requires nontrivial modification of the code. If one wants to develop a new numerical scheme for particle-based simulation or to apply it to new problem, it is necessary to write his/her own code. However, it is practically impossible for a single person, or even for a group of people, to develop a new code which can run efficiently on present-day HPC systems in a reasonable time. This difficulty, in our opinion, has slowed down the evolution of the entire field of computational science for the last two decades. Here we discussed the situation of particle-based codes. The situation of grid-based codes is similar.

In order to overcome the difficulty described above, we developed FDPS (Framework for Developing Particle Simulator)[1]. Our goal is to provide a software framework which enables researchers and programmers to develop high-performance particle simulation codes quickly. The basic idea of FDPS is to separate the above-described "difficult" part of the code, such as the domain decomposition and exchange of particles, from the description of the physical problem itself. The difficult part is implemented in FDPS as a library. A user of FDPS needs to define the data structure of particles and the interaction function of particles. Then FDPS receives these code, and generates an efficient code to perform domain decomposition and parallelized calculation of interaction. The user program first uses the functions of FDPS to do the interaction calculation, and then updates the physical data of particles, and repeats this loop until the end of simulation. Thus, the user of FDPS does not have to write complex code for parallelization.

FDPS supports particle simulations with arbitrary pairwise interactions. If the physical system includes many-body interactions, a user can calculate such interactions within his/her code, letting FDPS do the parallelization and calculation of pairwise interactions. In FDPS, the separation of parallelization and physical problem is done through the use of abstract data types implemented using C++ templates. Users of FDPS provide actual particle data class and interaction function to the class template defined in FDPS. Thus, users can develop a gravitational N-body code, Smoothed Particle Hydrodynamics (SPH) code, other particles-based fluid code, large-scale molecular dynamics code, a Discrete Element Method (DEM) code, and many other particle-based code using FDPS, and all of them will automatically parallelized with near-optimal load balancing. A similar approach to provide general-purpose framework is in [36], but it was limited to long-range interaction with $1/r$ potential.

One might think that, even though it is not impossible to develop general-purpose framework for parallel particle-based simulations, such a framework would be inevitably slow. Since the goal of FDPS is to make efficient use of present-day HPC systems, we designed FDPS so that it can achieve the performance not worse than that of existing large-scale parallel codes for particle simulations. We will discuss the achieved performance later.

The structure of this paper is as follows. In section 2, we overview how FDPS users implement a simulation code on FDPS. In section 3, we describe parallel algorithm implemented in FDPS. In section 4, we illustrate how FDPS actually simplifies the development of user programs, using two sample FDPS applications, and describe their performance. Finally, we summarize this study in section 5.

2. HOW FDPS WORKS

In this section, we describe how FDPS works in detail. In section 2.1, we describe the design concept of FDPS. In section 2.2, we present an N-body simulation code as an example, and describe how FDPS does the parallelization and what a user program should do.

2.1 Design and implementation

In this section, we describe the design and implementation of FDPS. We first present the abstract view of calculation codes for particle-based simulations on distributed-memory parallel computers, and then describe how such abstraction is realized in FDPS.

2.1.1 Abstract view of particle-based simulation codes

In a particle-based simulation code on a distributed-memory parallel computer, which uses spatial decomposition to reduce communication, time integration of the system proceeds in the following steps:

1. The entire computational domain is decomposed into subdomains, and usually one subdomain is assigned to one MPI process. Decomposition should achieve minimization of inter-node communication and good load-balance between nodes.

2. Particles are redistributed among the processes, so that each process owns particles in its subdomain.

3. Interactions between particles are calculated. Each process calculates interactions on its particles. To do so, it needs to receive the necessary data in other processes.

4. The data of particles are updated using the calculated interaction. This part is done without inter-process communication.

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

Steps 1, 2, and 3 involve parallelization and inter-process communications. FDPS provides library functions to perform these parts. Therefore, users of FDPS do not have to write their own code to perform parallelization and inter-process communication.

Let us now explain how FDPS provides the libraries to perform steps 1, 2, and 3 for arbitrary systems of particles, in other words, how FDPS provides the abstraction of these steps.

The particle-based simulation codes for which FDPS can be used is described by the initial value problem of the following set of ordinary differential equations:

$$\frac{d\vec{u}_i}{dt} = \vec{g}\left(\sum_j^N \vec{f}(\vec{u}_i, \vec{u}_j), \vec{u}_i\right), \quad (1)$$

where N is the number of particles in the system, $\vec{u}_i$ is a physical quantity vector of a particle, the function $\vec{f}$ indicates the contribution of particle j to the time derivative of physical quantities of particle i, and $\vec{g}$ is a function which converts the sum of the contributions to actual time derivative. For example, in the case of gravitational N-body simulation, $\vec{f}$ is the mutual gravity between particles, $\vec{g}$ would add, for example, the acceleration due to some external field, and $\vec{u}_j$ contains all necessary data of particles such as position, velocity, mass, and other parameters.

Hereafter, we call a particle receiving the interaction "i-particle", and a particle exerting that interaction "j-particle". The actual content of vector $\vec{u}_i$, and the functional forms of $\vec{f}$ and $\vec{g}$ depends on the physical system and how it is discretized.

Equation (1) implies that FDPS can only handle pairwise interactions which can be added to obtain the total contributions of other particles. If many-body interaction is important, one can still use FDPS to perform parallelization and calculation of pairwise interactions. Many-body interaction, such as angle and torsion of bonding force in molecular dynamics simulation, can be implemented in the user code.

2.1.2 *Design concept of FDPS*

FDPS provides a template library in C++ language, which receives the user-defined particle (vector $\vec{u}_i$) and functions to calculate pairwise interaction (function $\vec{f}$). The functions provided by this template library perform the steps 1, 2, and 3 described above. The function $\vec{g}$, and the actual time integration of $\vec{u}_i$ are done entirely in the user program. Thus, FDPS provides a powerful, and yet very flexible framework for the development of calculation codes for particle-based simulations.

A user of FDPS can develop the simulation code in the following three steps:

1. Define the data structure for $\vec{u}_i$, as a class in C++ language.
2. Define the function $\vec{f}$. It should be a function object in C++ language[2], which receives arrays of i-particles and j-particles, and calculates and accumulates $\vec{f}$ on i-particles.
3. Write the user program using the data class and functions provided by FDPS. Currently, the user program should also be written in C++.

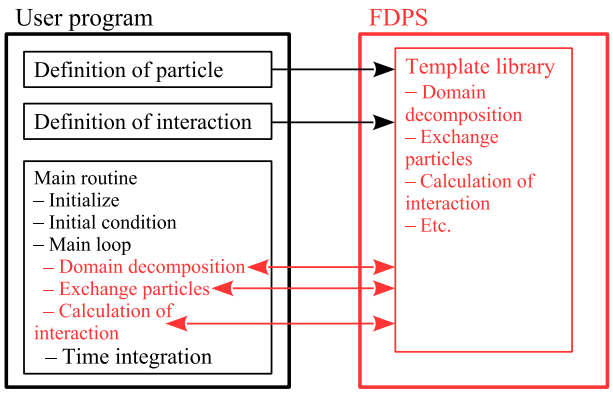

Figure 1: The basic concept of FDPS. The user program gives the definitions of particle and interaction to FDPS, and calls FDPS APIs.

Figure 1 illustrates how the user-defined code and FDPS functions interact. The user program gives the definition of particle and particle-particle interaction to FDPS at the compile time. When executed, the user program first does the initialization (the setup of MPI communication is done through a single call to an FDPS initialization function), and the setup of the initial condition. Then, the main integration loop is executed. In the main loop, first the domain decomposition and exchange of particles are performed, and then the calculation of interactions is performed. These are all done through library calls to FDPS functions. Finally, the time integration of particles using the calculated interaction is performed.

In the above description, the interaction calculation is done once per one iteration of the main loop. It is possible to use integration schemes which requires multiple evaluations of interaction within a single timestep, such as Runge-Kutta schemes. One can just call interaction calculation API of FDPS, with $\vec{u}_i$ containing necessary intermediate values.

FDPS takes care of parallelization using MPI, and it can also use OpenMP parallelization for internal operations and also for interaction calculation. Thus, an FDPS user does not have to worry about these issues. The efficient use of the cache memory and the SIMD execution unit is not directly taken care by the FDPS libraries, but handled through the interface to the user-defined interaction calculation function. The interface is defined so that the interaction calculation is done for multiple j-particles and multiple i-particles. Thus, it performs a large amount of calculation, on a small amount of data, since the calculation cost is the product of the numbers of i- and j-particles, and data amount is sum of them.

[2] A function pointer of C language can be operable.

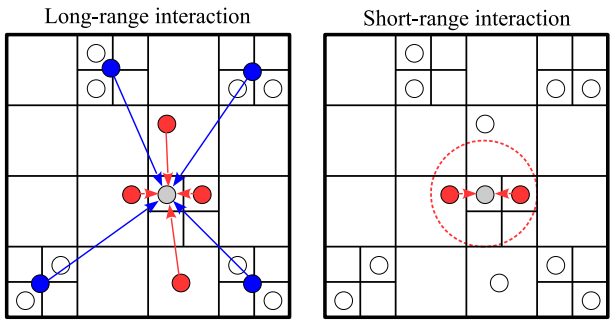

Figure 2: Long-range interaction (left) and short-range interaction (right). Gray, red, and blue points are i-particle, j-particle, and superparticle, respectively.

In order to make efficient use of the SIMD execution unit, the innermost loop should be written in such a way that can be recognized as the candidate of vectorization by the compiler used. The interface is defined as taking AoS (array of structures) arguments. Some compilers for some architecture might not be able to generate the code to utilize the SIMD unit for AoS data. In such a case, the user should write the interaction function in which the AoS data is converted internally to SoA (structure of arrays) data, and converted back to the AoS form after the interaction is actually calculated.

One might think that the definition of the system given in equation (1) implies that FDPS has to calculate N^2 interactions. If that were the case, FDPS would only be useful for small problems. We implemented $O(N)$ and $O(N \log N)$ calculation algorithms, for short-range and long-range interactions.

Within FDPS, the difference between long-range and short-range interactions is slightly different from the physical one of infinite and finite effective ranges. When we can and want to apply multipole-like expansion to the contribution from distant particles, we regard that interaction as long-range. The example of the long-range interaction is gravity and Coulomb interaction in the open boundary. When periodic boundary is used, they are usually evaluated using $\mathrm{P^3M}$ or PME method [20], in which the long-range part is evaluated using FFT, and only the interaction with long-range cutoff is evaluated directly. Even in this case, we can still apply multipole interaction as used in TreePM method [38, 10, 2, 14, 31, 39, 21, 22], and in this case the interaction is treated as long-range in FDPS.

For long-range interactions, FDPS uses standard Barnes-Hut tree algorithm [4, 5] parallelized for distributed-memory machines and optimized for cache memory and SIMD units [21, 22]. For short-range interactions, interaction list, or "neighbor list" for a group of i particles is constructed using a tree-based search, and that list is used for the actual interaction calculation. Figure 2 illustrates the long-range and short-range interactions and how they are calculated in FDPS

For the long-range interaction, a multipole-like interaction is used. Thus, equation (1) is modified to

$$\frac{d\vec{u}_i}{dt} = \vec{g}\left(\sum_j^{N_{\mathrm{J},i}} \vec{f}(\vec{u}_i, \vec{u}_j) + \sum_{j'}^{N_{\mathrm{S},i}} \vec{f}'(\vec{u}_i, \vec{u'}_{j'}), \vec{u}_i\right), \quad (2)$$

where $N_{\mathrm{J},i}$ and $N_{\mathrm{S},i}$ are, the number of j-particles and superparticles for which we apply multipole-like expansion, the vector $\vec{u'}_{j'}$ is the physical quantity vector of a superparticle, and the function $\vec{f}'$ indicates the interaction exerted on particle i by the superparticle j'. In simulations with a large number of particles N, $N_{\mathrm{J},i}$ and $N_{\mathrm{S},i}$ are many orders of magnitude smaller than N. The user should also specify how superparticles are constructed from ordinary particles, and also from superparticles in the lower level of the tree. For $1/r$ potential, which is the typical usage of the long-range interaction, FDPS provides the default way of construction of superparticles up to the quadrupole moment.

In the case of the short-range interaction, the calculation of contribution of distant particles is suppressed. Thus, equation (1) is modified to

$$\frac{d\vec{u}_i}{dt} = \vec{g}\left(\sum_j^{N_{\mathrm{J},i}} \vec{f}(\vec{u}_i, \vec{u}_j), \vec{u}_i\right), \quad (3)$$

As in the case of the long-range force, $N_{\mathrm{J},i}$ is much smaller than N, and usually independent of N.

2.2 A working example — gravitational N-body problem

In this section, we present the complete working example of a simulation code written using FDPS, to illustrate how a user actually uses FDPS. As the target problem, we use the gravitational N-body problem with an open boundary. Within the terminology of FDPS, the interaction between particles in the gravitational N-body problem is of the "long-range" type. Therefore, we need to specify the function to calculate interactions for both the ordinary particles and superparticles. For the sake of brevity, we use the center-of-mass approximation for superparticles, which means that we can actually use the same function for both types of particles.

The physical quantity vector $\vec{u}_i$ and interaction functions $\vec{f}$, $\vec{f}'$, and $\vec{g}$ for the gravitational N-body problem is now given by:

$$\vec{u}_i = (\vec{r}_i, \vec{v}_i, m_i) \quad (4)$$

$$\vec{f}(\vec{u}_i, \vec{u}_j) = \frac{Gm_j(\vec{r}_j - \vec{r}_i)}{(|\vec{r}_j - \vec{r}_i|^2 + \epsilon_i^2)^{3/2}} \quad (5)$$

$$\vec{f}'(\vec{u}_i, \vec{u'}_j) = \frac{Gm'_j\left(\vec{r'}_j - \vec{r}_i\right)}{\left(|\vec{r'}_j - \vec{r}_i|^2 + \epsilon_i^2\right)^{3/2}} \quad (6)$$

$$\vec{g}(\vec{F}_i, \vec{u}_i) = (\vec{v}_i, \vec{F}_i, 0), \quad (7)$$

$$\vec{F}_i = \sum_j^{N_{\mathrm{J},i}} \vec{f}(\vec{u}_i, \vec{u}_j) + \sum_{j'}^{N_{\mathrm{S},i}} \vec{f}'(\vec{u}_i, \vec{u'}_{j'}) \quad (8)$$

where m_i, $\vec{r}_i$, $\vec{v}_i$, and ϵ_i are, the mass, position, velocity, and gravitational softening of particle i, m'_j and $\vec{r'}_j$ are, the mass

and position of a superparticle j, and G is the gravitational constant. Note that the shapes of the functions $\vec{f}$ and $\vec{f}'$ are the same.

Listing 1: A sample code of N-body simulation

```
#include <particle_simulator.hpp>
using namespace PS;

class Nbody{
public:
    F64    mass, eps;
    F64vec pos, vel, acc;
    F64vec getPos() const {return pos;}
    F64 getCharge() const {return mass;}
    void copyFromFP(const Nbody &in){
        mass = in.mass;
        pos  = in.pos;
        eps  = in.eps;
    }
    void copyFromForce(const Nbody &out) {
        acc = out.acc;
    }
    void clear() {
        acc = 0.0;
    }
    void readAscii(FILE *fp) {
        fscanf(fp,
               "%lf%lf%lf%lf%lf%lf%lf",
               &mass, &eps,
               &pos.x, &pos.y, &pos.z,
               &vel.x, &vel.y, &vel.z);
    }
    void predict(F64 dt) {
        vel += (0.5 * dt) * acc;
        pos += dt * vel;
    }
    void correct(F64 dt) {
        vel += (0.5 * dt) * acc;
    }
};

template <class TPJ>
struct CalcGrav{
    void operator () (const Nbody * ip,
                      const S32 ni,
                      const TPJ * jp,
                      const S32 nj,
                      Nbody * force) {
        for(S32 i=0; i<ni; i++){
            F64vec xi  = ip[i].pos;
            F64    ep2 = ip[i].eps
                * ip[i].eps;
            F64vec ai = 0.0;
            for(S32 j=0; j<nj;j++){
                F64vec xj = jp[j].pos;
                F64vec dr = xi - xj;
                F64 mj  = jp[j].mass;
                F64 dr2 = dr * dr + ep2;
                F64 dri = 1.0 / sqrt(dr2);
                ai -= (dri * dri * dri
                       * mj) * dr;
            }
            force[i].acc += ai;
        }
    }
};

template<class Tpsys>
void predict(Tpsys &p,
             const F64 dt) {
    S32 n = p.getNumberOfParticleLocal();
    for(S32 i = 0; i < n; i++)
        p[i].predict(dt);
}

template<class Tpsys>
void correct(Tpsys &p,
             const F64 dt) {
    S32 n = p.getNumberOfParticleLocal();
    for(S32 i = 0; i < n; i++)
        p[i].correct(dt);
}

template <class TDI, class TPS, class TTFF>
void calcGravAllAndWriteBack(TDI &dinfo,
                             TPS &ptcl,
                             TTFF &tree) {
    dinfo.decomposeDomainAll(ptcl);
    ptcl.exchangeParticle(dinfo);
    tree.calcForceAllAndWriteBack
        (CalcGrav<Nbody>(),
         CalcGrav<SPJMonopole>(),
         ptcl, dinfo);
}

int main(int argc, char *argv[]) {
    F32 time  = 0.0;
    const F32 tend  = 10.0;
    const F32 dtime = 1.0 / 128.0;
    PS::Initialize(argc, argv);
    PS::DomainInfo dinfo;
    dinfo.initialize();
    PS::ParticleSystem<Nbody> ptcl;
    ptcl.initialize();
    PS::TreeForForceLong<Nbody, Nbody,
        Nbody>::Monopole grav;
    grav.initialize(0);
    ptcl.readParticleAscii(argv[1]);
    calcGravAllAndWriteBack(dinfo,
                            ptcl,
                            grav);
    while(time < tend) {
        predict(ptcl, dtime);
        calcGravAllAndWriteBack(dinfo,
                                ptcl,
                                grav);
        correct(ptcl, dtime);
        time += dtime;
    }
    PS::Finalize();
    return 0;
}
```

Listing 1 shows the complete code which can be actually

compiled and run, not only on a single-core machine but also massively-parallel, distributed-memory machines such as the full-node configuration of the K computer. The total number of lines is only 117.

Now let us explain how this sample code works. This code consists of four parts: The declaration to use FDPS (lines 1 and 2), the definition of the particle (the vector $\vec{u}_i$) (lines 4 to 35), the definition of the gravitational force (the functions $\vec{f}$ and $\vec{f}'$) (lines 37 to 61), and the actual user program, comprising a user-defined main routine and user-defined functions from which library functions of FDPS are called (lines 63 to line 117). In the following, we explain them step by step.

In order to declare to use FDPS, the only thing the user program need to do is to include the header file "particle_simulator.hpp". This file and other source library files of FDPS should be in the include path of the compiler. Everything in the standard FDPS library is provided as the header source library, since they are implemented as template libraries which need to receive particle class and interaction functions. Everything in FDPS is provided in the namespace "PS". Therefore in this sample program, we declare it as the default namespace to simplify the code. (For simplicity's sake, we do not omit the namespace "PS" of FDPS functions and class templates in the main routine.)

Before going to the 2nd parts, let us list the data types and classes defined in FDPS. `F32/F64` are data types of 32-bit and 64-bit floating points. `S32` is a data type of 32-bit signed integer. `F64vec` is a class of a vector consisting of three 64-bit floating points. This class provides several operators, such as the addition, subtraction and the inner product indicated by "*". Users need not use these data types in their own program, but some of the functions which users should define should return the values in these data types.

In the 2nd part, we define the particle, i.e. the vector $\vec{u}_i$, as a class `Nbody`. This class has member variables: `mass` (m_i), `eps` (ϵ_i), `pos` ($\vec{r}_i$), `vel` ($\vec{v}_i$), and `acc` ($d\vec{v}_i/dt$). Although the member variable `acc` does not appear in equation (4) – (7), we need this variable to store the result of the gravitational force calculation. A particle class for FDPS must provide public member functions `getPos`, `getCharge`, `copyFromFP`, `copyFromForce`, `clear`, and `readAscii`, in these names, so that the internal functions of FDPS can access the data within the particle class. For the name of the particle class itself and the names of the member variables, a user can use whatever names allowed by the C++ syntax. The member functions `predict` and `correct` are used in the user-defined part of the code to integrate the orbits of particles. Note that since the interaction used here is of $1/r$ type, the definition and construction method of the superparticle are given as the default in FDPS and not shown here.

In the 3rd part, the interaction functions $\vec{f}$ and $\vec{f}'$ are defined. Since the shapes of the functions $\vec{f}$ and $\vec{f}'$ are the same, we give one as a template function. The interaction function used in FDPS should have the following five arguments. The first argument `ip` is the pointer to the array of variables of particle class `Nbody`. This argument specifies i-particles which receive the interaction. The second argument `ni` is the number of i-particles. The third argument `jp` is the pointer to the array of variable of a template data type `TPJ`. This argument specifies j-particles or superparticles which exert the interaction. The fourth argument `nj` is the number of j-particles or super-particles. The fifth argument `force` is the pointer to the array of a variable of a user-defined class to which the calculated interaction on an i-particle can be stored. In this example, we used the particle class itself, but this can be another class or a simple array.

The interaction function should be defined as a function object, so that it can be passed to other functions as argument. Thus, it is declared as a `struct`, with the only member function `operator ()`. In this example, the interaction is calculated through a simple double loop. In order to make full advantage of the SIMD unit in modern processors, architecture-dependent tuning may be necessary, but only to this single function.

In the 4th part, we give the main routine and functions called from the main routine. In the main routine, we first initialize FDPS by calling `PS::Initialize` (line 95). Next, we create and initialize objects of FDPS classes (lines 96 to 102). In line 103, we read particle data from a file by calling FDPS function `PS::readParticleAscii`. Then, we calculate gravity of all the particles at the initial time by calling a function `calcGravAllAndWriteBack` (lines 104 to 106). In this function, we decompose the computational domain by calling `decomposeDomainAll` (line 83), relocate particle data by calling `exchangeParticle` (line 84), and calculate gravity by calling `calcForceAllAndWriteBack` (lines 85 to 88). From line 107 to line 114, we integrate the orbits of all the particles with Leap-Frog method. Finally, we finalize FDPS by calling `PS::Finalize` (line 115).

3. IMPLEMENTATION

In this section, we describe how the operations discussed in the previous section are implemented in FDPS. In section 3.1 we describe the domain decomposition and particle exchange, and in section 3.2, the calculation of interactions.

3.1 Domain decomposition and particle exchange

In this section, we describe how the domain decomposition and the exchange of particles are implemented in FDPS. We use the three-dimensional multi-section decomposition [24] as the method for the domain decomposition in FDPS. In this method, first we construct a cuboid which covers all particles. We call this cuboid the computational domain. When the periodic boundary condition is used, the dimensions of this cuboid is given from the user program. In the case of an open boundary, FDPS constructs the dimensions from positions of particles. This computational domain is then divided into n_x domains by planes perpendicular to the x-axis. Each of these n_x domains is then further subdivided into n_y domains by planes perpendicular to the y-axis, and finally the same thing is done for the z-axis. In each step, the division is made so that the computational load is equally distributed to all subdomains. We will discuss this point in more details below.

Figure 3 illustrates the result of the multi-section decomposition with $(n_x, n_y, n_z) = (7, 6, 1)$. We can see that the size

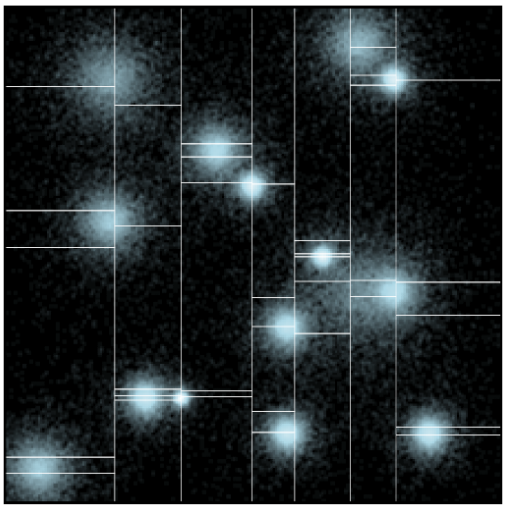

Figure 3: Example of the domain decomposition. The division is 7×6 in 2-dimension.

and shape of subdomains shows large variation. By allowing this variation, FDPS achieves quite good load balance and high scalability. Note that $n = n_x n_y n_z$ is the number of MPI processes. By default, values of n_x, n_y, and n_z are chosen so that they are integers close to $n^{1/3}$. For figure 3, we force the numbers used to make two-dimensional decomposition.

We adopts the "sampling method" [9] for the calculation of the geometry of subdomains (we denote this part as "domain decomposition"). The detail algorithm is the same as [21]. The advantage of the sampling method is in its low calculation and communication cost. They are $\mathcal{O}(N_\mathrm{s} \log N_\mathrm{s})$ and $\mathcal{O}(N_\mathrm{s})$, where N_s is the total number of sampled particles. Thus, as far as N_s is not much larger than N/n, the average number of particles per process, the cost of domain decomposition is negligible.

After the domain decomposition is done and the result is broadcasted to all processes, they exchange particles so that each of them has the particles in its domain. Since each process has the complete information of the domain decomposition, this part is pretty straightforward to implement. Each process looks at each of its particles, and determines if that particle is still in its domain. If not, the process determines to which process that particle should be sent. After the destinations of all particles are determined, each process sends them out, using `MPI_Alltoallv` function.

3.2 Interaction calculation

In this section, we describe the implementation of the calculation of interactions in FDPS. Conceptually, it consists of the following two steps. In the first step, each process determines, for each of other processes, which of its particles and superparticles are required by that process for interaction calculation, and sends them to it. In the second step, each process calculates the interactions onto i-particles by calling user-defined function objects.

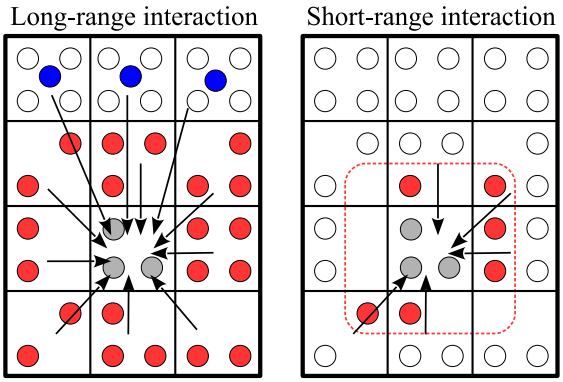

Figure 4: Illustration of communication among processes during the interaction calculation.

For both steps, the octree structure is used, both for long- and short-range interactions. In the first step, each process constructs the tree structure for its local particles, and uses it to determine what data should be sent to other processes. For the long-range interactions, this part is done through the usual tree traversal [4, 5]. For the short-range interactions, tree traversal is also used. A cube in a tree need to be subdivided if it is within the cutoff length from anywhere in the domain of the process to which the data will be sent. The current implementation of FDPS can handle four different types of the cutoff length for the "short-range" interaction: fixed, j-dependent, i-dependent and symmetric. For i-dependent and symmetric cutoffs, FDPS does the tree traversal twice. Figure 4 illustrates what data are sent, for both long- and short-range interactions.

After a process receives all data it requires, it reconstructs the tree structure which contains all information necessary to calculate interactions on its particles.

The interaction calculation is performed using this new tree. The procedure is the same as described in detail in the literature [5, 23], except for the following two differences. First, this part is fully multithreaded using OpenMP, to achieve very good parallel performance. Second, for the interaction calculation the user-provided functions are used, to achieve the flexibility and high performance at the same time.

4. PERFORMANCE

In this section, we report the performance of two astrophysical applications of FDPS: a gravitational N-body simulation code in section 4.1 and an SPH simulation code with self-gravity in 4.2.

4.1 *N*-body simulation

In this section, we discuss the performance and scalability of a gravitational N-body simulation code implemented using FDPS. The code is essentially the same as the sample code described in section 2.2, except for the following two differences in the user code for the calculation of the interaction. First, here, we used the expansion up to quadrupole moment, instead of monopole-only one used in the sample code, to improve the accuracy. Second, we used highly-optimized

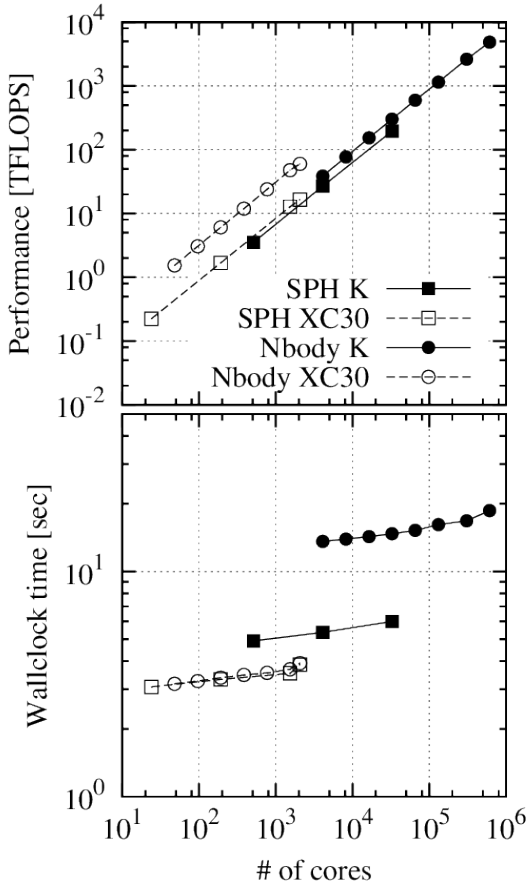

Figure 5: Performance measured in the speed of the floating-point operation (top) and wallclock time per one timestep (bottom) plotted as functions of the number of cores. Filled and open points indicate the performance on K computer and Cray XC30, respectively. Square and circle points show the performance of our SPH simulation code and N-body simulation code, respectively.

kernel developed using SIMD builtin functions, instead of the simple one in the sample code.

We use the Plummer model [28] to realize particle distribution at the initial time. We adopt $\theta = 0.4$ for the opening angle for the tree algorithm. In this paper, we present the weak-scaling performance of the code with FDPS. Therefore we fixed the number of particles per core to 250 thousands.

We run our N-body simulation code on two platforms: K computer at RIKEN Advanced Institute for Computational Science, and Cray XC30 (with Haswell processors) at National Astronomical Observatory of Japan. The maximum numbers of cores we used are 612352 cores (76544 nodes) on K computer, and 2064 cores (86 nodes) on Cray XC30. The run using 612352 cores on K computer is almost a full-system one, where the full-system run operates 663552 cores. Precision of floating point for interaction calculations is double on K computer, and single on Cray XC30. The single floating point precision is sufficient for the scientific calculation on this field. Since the performance of the single floating point precision is the same as that of the double one on K computer, we adopt the double one for the calculation on K computer.

Filled and open circles in Figure 5 show the measured performance of our N-body simulation code. We can see the measured efficiency and scalability are both very good on both platforms. On K computer, efficiency is very close to 50% for the entire range of nodes. Wallclock time shows slight increase for larger number of nodes, but this is due to the increase of the calculation cost and not due to the degradation of the efficiency. The performance on Cray XC30 is about 4 times higher than that on K computer, since the peak performance per core in single floating point precision on Cary XC30 is about 5 times higher than that in double floating point precision on K computer

We compare the performance of our N-body simulation code with that of GreeM code [21, 22]. GreeM code treats N-body simulation with TreePM algorithm, and was awarded the 2012 ACM Gordon Bell Prize. The wallclock times of GreeM and our codes are 2.80×10^{-11} and 1.16×10^{-10} sec/step/particle. Our code using FDPS seems to be 4 times slower than GreeM. However, this is due to the difference of the algorithms used. Since TreePM algorithm calculates forces from distant particles by Fast Fourier Transform, the number of particle-particle interactions in TreePM algorithm is much smaller than that in the tree algorithm. Therefore, comparison of the wallclock time per interaction is more fair. The wallclock time is 1.14×10^{-14} (GreeM) and 1.71×10^{-14} (ours) sec/step/interaction. The performance is comparable, even though the number of particles per node is smaller by a factor of 6. The performance of GreeM was measured for 1.6 million particles per core, while we used 250 thousand particles per core.

4.2 SPH simulation

In this section, we discuss the performance of an SPH simulation code with self-gravity implemented using FDPS. The test problem used is the simulation of Giant Impact (GI). The giant impact hypothesis [19, 11] is one of the most popular scenarios for the formation of the Moon. The hypothesis is as follows. About 5 billion years ago, a Mars-sized object (hereafter, the impactor) collided with the proto-Earth (hereafter, the target). The collision scattered a large amount of debris, which first formed the debris disk and eventually the Moon. Many researchers have performed simulations of GI, using the SPH method. [8, 12, 1].

For the gravity, we used monopole-only kernel with $\theta = 0.5$. We adopt the standard SPH scheme [25, 29, 32] for the hydro part. Artificial viscosity was used to handle shock waves [26], and the standard Balsara switch was used to reduce the shear viscosity [3]. Assuming that the target and impactor consist of granite, we adopt the equation of state of granite [8] for the particles. The initial conditions, such as the orbital parameters of the two objects, are the same as those in [8]. In this paper, we report the weak scaling performance with about 250k particles per node. For the largest calculation, we used 1.0 billion particles and 4096 nodes.

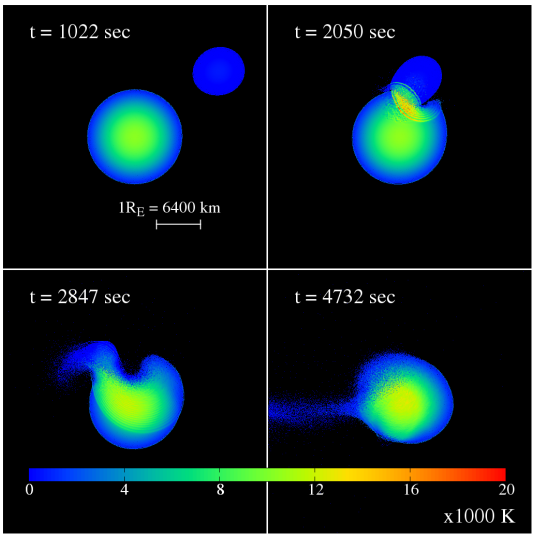

Figure 6: Temperature maps of the target and impactor in the run of 9.9 million particles at four different epochs.

Similarly to our N-body simulation code, we run our SPH simulation code on K computer and Cray XC30. The maximum numbers of cores we used are 32768 cores (4096 nodes) on K computer, and 2064 cores (86 nodes) on Cray XC30.

For gravity calculations, we used highly-optimized kernel developed using SIMD builtin functions on both platforms. However, for fluid calculations, we used the optimized kernel on the K computer, but do not on Cray XC30.

Figure 6 shows the time evolution of the target and impactor for a run with 9.9 million particles. We can see that the shock waves are formed just after the moment of impact in both the target and impactor (t=2050sec). The shock propagates in the target, while the impactor is completely disrupted (t=2847sec) and debris is ejected. Part of the debris falls back to the target, while the rest will eventually form the disk and the Moon. So far, the resolution used in the published papers have been much lower. We plan to use this code to improve the accuracy of the GI simulations.

Filled and open squares in Figure 5 show the measured performance of our SPH simulation code. We can see the measured efficiency and scalability are both very good on both platforms, similarly to our N-body simulation code in the previous section. On the K computer, efficiency is very close to 40% for the entire range of the number of nodes. The difference of SPH performance between K computer and Cray XC30 is smaller than that of N-body performance. This is because our SPH simulation code on Cray XC30 does not efficiently utilize SIMD instructions for fluid calculations. Although the SIMD instructions are used for gravity calculations, the calculation cost is dominated by fluid calculations.

The largest number of particles used for GI simulations so far reported is 100 million [35]. Unfortunately, performance numbers are not given. After we replace the interaction kernels with SIMD-optimized ones for hydrodynamics part, we believe we can achieve the performance not so much lower than that we achieved for pure gravity calculation.

5. CONCLUSION

We have developed a novel framework for particle-based simulations, FDPS. Users of FDPS need not care about complex implementations of domain decomposition, exchange of particles, communication of data for the interaction calculation, or optimization for multi-core processors. Using FDPS, particle simulation codes which achieve high performance and high scalability on massively parallel distributed-memory machines can be easily developed for a variety of problems. As we have shown in section 2.2, a parallel N-body simulation code can be written in less than 120 lines. Example implementations of gravitational N-body simulation and SPH simulation showed excellent scalability and performance. We hope FDPS will help researchers to concentrate on their research, by removing the burden of complex code development for parallization and architecture-dependent tuning.

6. ACKNOWLEDGMENTS

We thank M. Fujii for providing initial conditions of spiral simulations, T. Ishiyama for providing his Particle Mesh code, and Y. Maruyama for being the first user of FDPS. We are grateful to M. Tsubouchi for her help in managing the FDPS development team. This research used computational resources of the K computer provided by the RIKEN Advanced Institute for Computational Science through the HPCI System Research project (Project ID:ra000008). Numerical computations were in part carried out on Cray XC30 at Center for Computational Astrophysics, National Astronomical Observatory of Japan.

7. REFERENCES

[1] E. Asphaug and A. Reufer. Mercury and other iron-rich planetary bodies as relics of inefficient accretion. *Nature Geoscience*, 7:564–568, Aug. 2014.

[2] J. S. Bagla. TreePM: A Code for Cosmological N-Body Simulations. *Journal of Astrophysics and Astronomy*, 23:185–196, Dec. 2002.

[3] D. S. Balsara. von Neumann stability analysis of smooth particle hydrodynamics–suggestions for optimal algorithms. *Journal of Computational Physics*, 121:357–372, 1995.

[4] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, Dec. 1986.

[5] J. E. Barnes. A modified tree code: Don't laugh; It runs. *Journal of Computational Physics*, 87:161–170, Mar. 1990.

[6] J. Bédorf, E. Gaburov, M. S. Fujii, K. Nitadori, T. Ishiyama, and S. P. Zwart. 24.77 pflops on a gravitational tree-code to simulate the milky way galaxy with 18600 gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 54–65, Piscataway, NJ, USA, 2014. IEEE Press.

[7] J. Bédorf, E. Gaburov, and S. Portegies Zwart. A sparse octree gravitational N-body code that runs

entirely on the GPU processor. *Journal of Computational Physics*, 231:2825–2839, Apr. 2012.

[8] W. Benz, W. L. Slattery, and A. G. W. Cameron. The origin of the moon and the single-impact hypothesis. I. *Icarus*, 66:515–535, June 1986.

[9] D. Blackston and T. Suel. Highly portable and efficient implementations of parallel adaptive n-body methods. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, SC '97, pages 1–20, New York, NY, USA, 1997. ACM.

[10] P. Bode, J. P. Ostriker, and G. Xu. The Tree Particle-Mesh N-Body Gravity Solver. *Astrophysical Journal Supplement*, 128:561–569, June 2000.

[11] A. G. W. Cameron and W. R. Ward. The Origin of the Moon. In *Lunar and Planetary Science Conference*, volume 7 of *Lunar and Planetary Science Conference*, page 120, Mar. 1976.

[12] R. M. Canup, A. C. Barr, and D. A. Crawford. Lunar-forming impacts: High-resolution SPH and AMR-CTH simulations. *Icarus*, 222:200–219, Jan. 2013.

[13] J. Dubinski. A parallel tree code. *New Astronomy*, 1:133–147, Oct. 1996.

[14] J. Dubinski, J. Kim, C. Park, and R. Humble. GOTPM: a parallel hybrid particle-mesh treecode. *New Astronomy*, 9:111–126, Feb. 2004.

[15] E. Gaburov, S. Harfst, and S. Portegies Zwart. SAPPORO: A way to turn your graphics cards into a GRAPE-6. *New Astronomy*, 14:630–637, Oct. 2009.

[16] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 62:1–62:12, New York, NY, USA, 2009. ACM.

[17] T. Hamada and K. Nitadori. 190 tflops astrophysical n-body simulation on a cluster of gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–9, Washington, DC, USA, 2010. IEEE Computer Society.

[18] T. Hamada, K. Nitadori, K. Benkrid, Y. Ohno, G. Morimoto, T. Masada, Y. Shibata, K. Oguri, and M. Taiji. A novel multiple-walk parallel algorithm for the barnes-hut treecode on gpus-towards cost effective, high performance n-body simulation. *Computer Science-Research and Development*, 24(1):21–31, 2009.

[19] W. K. Hartmann and D. R. Davis. Satellite-sized planetesimals and lunar origin. *Icarus*, 24:504–514, Apr. 1975.

[20] R. Hockney and J. Eastwood. *Computer Simulation Using Particles*. CRC Press, 1988.

[21] T. Ishiyama, T. Fukushige, and J. makino. Greem: Massively parallel treepm code for large cosmological n -body simulations. *Publications of the Astronomical Society of Japan*, 61(6):1319–1330, December 2009.

[22] T. Ishiyama, K. Nitadori, and J. makino. 4.45 pflops astrophysical n-body simulation on k computer – the gravitational trillion-body problem. In *SC '12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Salt Lake City, UT, USA, Nov 11 - 15, 2012*, 2012.

[23] J. Makino. A Modified Aarseth Code for GRAPE and Vector Processors. *Publications of the Astronomical Society of Japan*, 43:859–876, Dec. 1991.

[24] J. Makino. A Fast Parallel Treecode with GRAPE. *Publications of Astronomical Society of Japan*, 56:521–531, June 2004.

[25] J. J. Monaghan. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30:543–574, 1992.

[26] J. J. Monaghan. SPH and Riemann Solvers. *Journal of Computational Physics*, 136:298–307, Sept. 1997.

[27] K. Nitadori, J. Makino, and P. Hut. Performance tuning of N-body codes on modern microprocessors: I. Direct integration with a hermite scheme on x86_64 architecture. *New Astronomy*, 12:169–181, Dec. 2006.

[28] H. C. Plummer. On the problem of distribution in globular star clusters. *Monthly Notices of the Royal Astronomical Society*, 71:460–470, Mar. 1911.

[29] S. Rosswog. Astrophysical smooth particle hydrodynamics. *New Astronomy Reviews*, 53:78–104, Apr. 2009.

[30] J. K. Salmon and M. S. Warren. Skeletons from the treecode closet. *Journal of Computational Physics*, 111:136–155, Mar. 1994.

[31] V. Springel. The cosmological simulation code gadget-2. volume 364, pages 1105–1134, December 2005.

[32] V. Springel. Smoothed Particle Hydrodynamics in Astrophysics. *Annual Review of Astronomy and Astrophysics*, 48:391–430, Sept. 2010.

[33] A. Tanikawa, K. Yoshikawa, K. Nitadori, and T. Okamoto. Phantom-GRAPE: Numerical software library to accelerate collisionless N-body simulation with SIMD instruction set on x86 architecture. *New Astronomy*, 19:74–88, Feb. 2013.

[34] A. Tanikawa, K. Yoshikawa, T. Okamoto, and K. Nitadori. N-body simulation for self-gravitating collisional systems with a new SIMD instruction set extension to the x86 architecture, Advanced Vector eXtensions. *New Astronomy*, 17:82–92, Feb. 2012.

[35] L. F. A. Teodoro, M. S. Warren, C. Fryer, V. Eke, and K. Zahnle. A One Hundred Million SPH Particle Simulation of the Moon Forming Impact. In *Lunar and Planetary Science Conference*, volume 45 of *Lunar and Planetary Science Conference*, page 2703, Mar. 2014.

[36] M. S. Warren and J. K. Salmon. A portable parallel particle program. *Computer Physics Communications*, 87:266–290, May 1995.

[37] M. S. Warren, J. K. Salmon, D. J. Becker, M. P. Goda, T. L. Sterling, and W. Winckelmans. Pentium pro inside: I. a treecode at 430 gigaflops on asci red, ii. price/performance of $50/mflop on loki and hyglac. In *SC*, page 61. IEEE, 1997.

[38] G. Xu. A New Parallel N-Body Gravity Solver: TPM. *Astrophysical Journal Supplement*, 98:355, May 1995.

[39] K. Yoshikawa and T. Fukushige. PPPM and TreePM Methods on GRAPE Systems for Cosmological N-Body Simulations. *Publications of Astronomical Society of Japan*, 57:849–860, Dec. 2005.