Journal: Publications of the Astronomical Society of Japan
Article doi: 10.1093/pasj/psy062
Article title: Fortran interface layer of the framework for developing particle simulator FDPS
First Author: Daisuke NAMEKATA
Corr. Author: Daisuke NAMEKATA

INSTRUCTIONS

We encourage you to use Adobe's editing tools (please see the next page for instructions). If this is not possible, please list clearly in an e-mail. Please do not send corrections as track changed Word documents.

Changes should be corrections of typographical errors only. Changes that contradict journal style will not be made.

These proofs are for checking purposes only. They should not be considered as final publication format. The proof must not be used for any other purpose. In particular we request that you: do not post them on your personal/institutional web site, and do not print and distribute multiple copies. Neither excerpts nor all of the article should be included in other publications written or edited by yourself until the final version has been published and the full citation details are available. You will be sent these when the article is published.

1. **Licence to Publish:** Oxford Journals requires your agreement before publishing your article. If you haven't already completed this, please sign in with your My Account information and complete the online licence form. Details on how to do this can be found in the Welcome to Oxford Journals email.
2. **Permissions: Permission to reproduce any third party material in your paper should have been obtained prior to acceptance. If your paper contains figures or text that require permission to reproduce, please inform me immediately by email.**
3. **Author groups:** Please check that all names have been spelled correctly and appear in the correct order. Please also check that all initials are present. Please check that the author surnames (family name) have been correctly identified by a pink background. If this is incorrect, please identify the full surname of the relevant authors. Occasionally, the distinction between surnames and forenames can be ambiguous, and this is to ensure that the authors' full surnames and forenames are tagged correctly, for accurate indexing online.
4. **Figures:** If applicable, figures have been placed as close as possible to their first citation. Please check that they are complete and that the correct figure legend is present. Figures in the proof are low resolution versions that will be replaced with high resolution versions when the journal is printed.
5. **Missing elements:** Please check that the text is complete and that all figures, tables and their legends are included.
6. **Special characters and equations:** Please check that special characters, equations and units have been reproduced accurately.
7. **URLs:** Please check that all web addresses cited in the text, footnotes and reference list are up-to-date.
8. **Funding:** If applicable, any funding used while completing this work should be highlighted in the Acknowledgements section. Please ensure that you use the full official name of the funding body.

AUTHOR QUERIES - TO BE ANSWERED BY THE CORRESPONDING AUTHOR

The following queries have arisen during the typesetting of your manuscript. Please answer these queries by marking the required corrections at the appropriate point in the text.

Query No.	Nature of Query	Author's Response
Q1	Author: Please note, color figures in print will be charged £250/€325/$400/¥25,000 per page. Color figures online will not be charged. Please confirm which figures should be printed in color and reword the legend/text to avoid using reference to color if the figures are to be printed in black and white.	We hope all the figures to be printed in color.
Q2	Author: To check that we have your surnames correctly identified and tagged (e.g. for indexing), we have coloured pink the names that we have assumed are surnames. If any of these are wrong, please let us know so that we can amend the tagging.	No correction is needed.
Q3	Author: Please check that three to five key words have been selected from the PASJ list of approved key words which is appended to these proofs.	We chose all the key words from the PASJ list of approved key words.
Q4	Author: please clarify what is meant by "without taking care of" in this context. Thank you.	We mean that researchers do not need to parallelize their simulation codes by themselves.
Q5	Author: please expand the acronym "MPI" here at its first use. Thank you.	MPI is the acronym for Message Passing Interface.
Q6	Author: please check the appearance of this and all the other images in the article proofs.	We confirmed that the appearances of all the figures are OK.
Q7	Author: the meaning of the wording "... corresponds to define specific..." is not clear—please could you check it and change the wording to improve the clarity? Thank you.	We'd like to replace the original sentence as follows (we hope this improves the clarity). This procedure corresponds to the one defining public member functions with special names in a particle class when using FDPS from C++.
Q8	Author: journal style rules state that only vectors should be styled in bold italic, e.g. $\boldsymbol{r}$. Please check that the correct style is used throughout this paper.	We confirmed that the bold italic styles are used for vectors only.
Q9	Author: please clarify what is meant by "does not take care about" in this context. Thank you.	We mean by this sentence that FDPS does not optimize interaction functions. Use of "take care of" may be more appropriate.
Q10	Author: the meaning of the wording "to be" in the context of "have member procedures/functions to be interoperable..." does not make sense. Please could you check this setence and reword it? Thank you.	In this sentence, we just describe the condition in which derived data type in Fortran is interoperable with C. The condition is that derived data type does not has any member procedures/functions. Inserting "for them" just before "to be" may make the sentence clearer.
Q11	Author: Please provide up-to-date publication details (e.g. journal, volume, page number) for this reference, if possible. If no new details are available, the "???," marks must be deleted. Thank you.	The up-to-data publication details is ApJ, 858, 26. Please see also the following link: http://adsabs.harvard.edu/abs/2018ApJ...858...26T

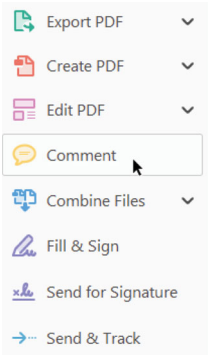
MAKING CORRECTIONS TO YOUR PROOF

These instructions show you how to mark changes or add notes to your proofs using Adobe Acrobat Professional versions 7 and onwards, or Adobe Reader DC. To check what version you are using go to **Help** then **About**. The latest version of Adobe Reader is available for free from get.adobe.com/reader.

DISPLAYING THE TOOLBARS

Adobe Reader DC

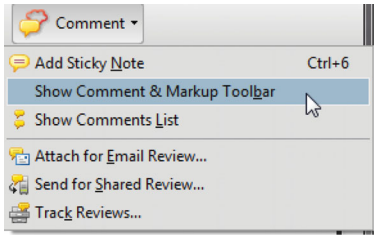
In Adobe Reader DC, the Comment toolbar can be found by clicking 'Comment' in the menu on the right-hand side of the page (shown below).



The toolbar shown below will then display along the top.

Acrobat Professional 7, 8, and 9

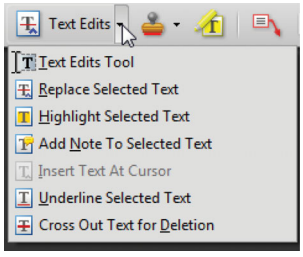
In Adobe Professional, the Comment toolbar can be found by clicking 'Comment(s)' in the top toolbar, and then clicking 'Show Comment & Markup Toolbar' (shown below).



The toolbar shown below will then be displayed along the top.

USING TEXT EDITS AND COMMENTS IN ACROBAT

This is the quickest, simplest and easiest method both to make corrections, and for your corrections to be transferred and checked.



1. Click **Text Edits**
2. Select the text to be annotated or place your cursor at the insertion point and start typing.
3. Click the **Text Edits** drop down arrow and select the required action.

You can also right click on selected text for a range of commenting options, or add sticky notes.

SAVING COMMENTS

In order to save your comments and notes, you need to save the file (**File, Save**) when you close the document.

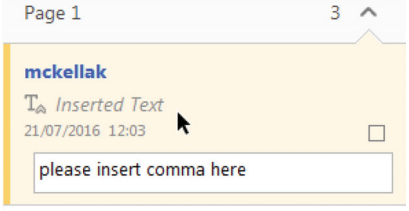
USING COMMENTING TOOLS IN ADOBE READER

All commenting tools are displayed in the toolbar. You cannot use text edits, however you can still use highlighter, sticky notes, and a variety of insert/replace text options.

POP-UP NOTES

In both Reader and Acrobat, when you insert or edit text a pop-up box will appear. In **Acrobat** it looks like this:

In **Reader** it looks like this, and will appear in the right-hand pane:



DO NOT MAKE ANY EDITS DIRECTLY INTO THE TEXT, USE COMMENTING TOOLS ONLY.

Publ. Astron. Soc. Japan (2018) 00 (0), 1–17
doi: 10.1093/pasj/psy062
Advance Access Publication Date: 2018 0

Fortran interface layer of the framework for developing particle simulator FDPS

Daisuke Namekata,[1,*] Masaki Iwasawa,[1,*] Keigo Nitadori,[1,*] Ataru Tanikawa,[1,2,*] Takayuki Muranushi,[1,*,†] Long Wang,[1,3,*] Natsuki Hosono,[1,4,*] Kentaro Nomura,[1,*] and Junichiro Makino[1,5,6,*]

[1]RIKEN Advanced Institute for Computational Science, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo 650-0047, Japan
[2]Department of Earth and Astronomy, College of Arts and Science, The University of Tokyo, 3-8-1 Komaba, Meguro-ku, Tokyo 153-8902, Japan
[3]Argelander Institut Für Astronomie, Auf Dem Hügel 71, 53121 Bonn, Germany
[4]Yokohama Institute for Earth Sciences, Japan Agency for Marine-Earth Science and Technology, 3173-25, Showa-machi, Kanazawa-ku, Yokohama-city, Kanagawa, 236-0001, Japan
[5]Department of Planetology, Graduate School of Science, Kobe University, 1-1 Rokkodai-cho, Nada-ku, Kobe 657-8501, Hyogo, Japan
[6]Earth-Life Science Institute, Tokyo Institute of Technology, 2-12-1 Ookayama, Meguro-ku, Tokyo 152-8551, Japan

*Email: daisuke.namekata@riken.jp (DN), masaki.iwasawa@riken.jp (MI), keigo@riken.jp (KN), tanikawa@ea.c.u-tokyo.ac.jp (AT), takayuki.muranushi@riken.jp (TM), long.wang@riken.jp (LW), natsuki.hosono@riken.jp (NH), kentaro.nomura@riken.jp (KN), makino@mail.jmlab.jp (JM)

Received 2018 February 16; Accepted 2018 May 1

Abstract

Numerical simulations based on particle methods have been widely used in various fields including astrophysics. To date, various versions of simulation software have been developed by individual researchers or research groups in each field, through a huge amount of time and effort, even though the numerical algorithms used are very similar. To improve the situation, we have developed a framework, called FDPS (Framework for Developing Particle Simulators), which enables researchers to develop massively parallel particle simulation codes for arbitrary particle methods easily. Until version 3.0, FDPS provided an API (application programming interface) for the C++ programming language only. This limitation comes from the fact that FDPS is developed using the template feature in C++, which is essential to support arbitrary data types of particle. However, there are many researchers who use Fortran to develop their codes. Thus, the previous versions of FDPS require such people to invest much time to learn C++. This is inefficient. To cope with this problem, we developed a Fortran interface layer in FDPS, which provides API for Fortran. In order to support arbitrary data types of particle in Fortran, we design the Fortran interface layer as follows. Based on a given derived data type in Fortran representing particle, a Python script provided by us automatically generates a library that manipulates the C++ core part of FDPS. This library is seen as a Fortran module providing an API of FDPS from the Fortran side and uses C programs internally to interoperate Fortran with C++. In this way, we have overcome several technical issues when emulating

a 'template' in Fortran. Using the Fortran interface, users can develop all parts of their codes in Fortran. We show that the overhead of the Fortran interface part is sufficiently small and a code written in Fortran shows a performance practically identical to the one written in C++.

Key words: dark matter — galaxies: evolution — methods: numerical — planets and satellites: formation

1 Introduction

Numerical simulations based on particle methods have been widely used in many fields of science and engineering. For example, in astrophysics, gravitational N-body and smoothed particle hydrodynamics (SPH) simulations are commonly used to study dynamics of celestial bodies. In the context of engineering and biology, moving particle semi-implicit (MPS) method, molecular dynamics (MD), power dynamics, and distinct element method (DEM) simulations are utilized for a variety of purposes such as disaster prevention and drug discovery. The reason for the frequent use of particle-based methods may be attributed to the fact that various kinds of physical systems can be well modeled by collections of particles, and particle-based methods have several advantages (e.g., adaptivity in space and time, Galilean invariance). The numbers of particles used in simulations go on increasing because there has been a rise in the need for refining results drawn from numerical simulations. For instance, cosmological N-body simulations can be used to study the evolution of rare objects such as quasars and active galactic nuclei. In order to make a prediction for such objects with a good statistical accuracy, we need to simulate a large volume of the universe (Ishiyama et al. 2015). Large-scale simulations are also required in MD simulations of vapor-to-liquid nucleation for a direct comparison of predicted nucleation rates with those measured by laboratory experiments (Diemand et al. 2013). This trend requires researchers to develop simulation codes that run efficiently on modern distributed-memory parallel supercomputers.

However, developing such a code is quite difficult, time-consuming task, because in order to achieve high efficiency we need to implement efficient algorithms, such as the Barnes–Hut tree algorithm (Barnes & Hut 1986), as well as some load-balancing mechanism into the code. To date, simulation codes have been developed by individual researchers or research groups in each field of science, costing a huge amount of time and effort, even though the numerical algorithms used are very similar to each other. Besides, in many cases, simulation codes are being developed for each specific application of particle methods, making it difficult for researchers to try a new method or an alternative method. This is not efficient.

In order to improve this situation, we have developed a framework called FDPS (Framework for Developing Particle Simulators; Iwasawa et al. 2015, 2016)[1] that enables researchers to develop codes for massively parallel particle simulations for arbitrary particle methods easily, without taking care of the parallelization of their codes. More specifically, FDPS provides a set of functions that (1) divide a computational domain into subdomains based on a given distribution of particles and assigns each subdomain to one MPI process, (2) exchange the information of particles among processes so that each process owns particles in its subdomain, (3) collect the information of particles necessary to perform interaction calculations in each process, and (4) perform interaction calculations using the Barnes–Hut tree algorithm for long-range force, or a fast tree-based neighbor search for short-range force. All of these functions [hereafter called FDPS API (application programming interface)] are highly optimized, and codes developed using FDPS show good scalings for a large number of processes (up to $\sim 10^5$ processes; Iwasawa et al. 2016). Thus, FDPS allows researchers to concentrate on their studies without spending time on the parallelization and complex optimization of the codes. FDPS has already been used to develop various applications (e.g., Hosono et al. 2016a,b, 2017; Michikoshi & Kokubo 2017; Tanikawa et al. 2017; Iwasawa et al. 2017; Tanikawa 2018b, 2018a).

One issue affecting the previous versions of FDPS (version 2.0 or ealier) is that it requires researchers to develop codes in the C++ programming language. This limitation comes from the fact that FDPS is written in C++, in order to use the template feature in C++ to support arbitrary data types of particles. However, there are many supercomputer users who mainly use Fortran language to develop codes. In order for such people to use the functionalities of FDPS, they must first spend time learning the C++ language. In addition, programs which they have written in Fortran in the past can no longer be used in any newly-developed codes. To cope with this problem, we have redesigned FDPS to have a Fortran interface layer (hereinafter, we call it "Fortran interface"). This layer provides an API for Fortran, i.e., a set of subroutines/functions that enables the use of the

[1] ⟨https://github.com/FDPS/FDPS⟩.

functionalities of FDPS from Fortran. Thus, in FDPS version 3.0 or later, researchers can develop their simulation codes in Fortran. In this paper, we present the basic design and implementation of Fortran interface and compare the performance of a simple application written using Fortran interface with that written in C++ using FDPS. We show that the overhead of Fortran interface is sufficiently small and the performances of both codes are almost identical.

This paper is organized as follows: In section 2, we briefly review the C++ core part of FDPS necessary to explain the design and implementation of Fortran interface, which are themselves described in section 3. The performance of the application developed using Fortran interface is shown in section 4. In section 5, we present one example of practical application. Finally, in section 6, we summarize this study.

2 Overview of FDPS

In this section, we provide a brief overview of the previous versions (version 2.0 or earlier) of FDPS. In subsection 2.1, we describe what FDPS does and what FDPS looks like for users. In subsection 2.2, we explain the implementation details of the previous versions of FDPS.

2.1 What FDPS does and its user interface design

FDPS provides C++ library functions that perform tasks required for parallel particle simulations. In distributed-memory parallel supercomputers, particle simulations are generally performed through the following procedure.

(1) The computational domain is divided into subdomains based on a given distribution of particles and each subdomain is assigned to one MPI process. We call this task "domain decomposition".
(2) Particles are exchanged among processes so that each process owns particles in its subdomain. We call this task "particle exchange".
(3) Each process collects the information necessary to perform interaction calculation for the particles belonging to this process.
(4) Each process performs interaction calculation.
(5) The information about particles is updated using the result of the interaction calculation in each process.

Among the procedures above, procedures (1)–(3) are specific for parallel computation and the implementation of them into a code is a daunting task. As described in section 1, FDPS provides APIs, a suite of functions that perform these procedures efficiently. Therefore, using FDPS, researchers can avoid the implementation of these complex procedures. In addition, FDPS also provide APIs that

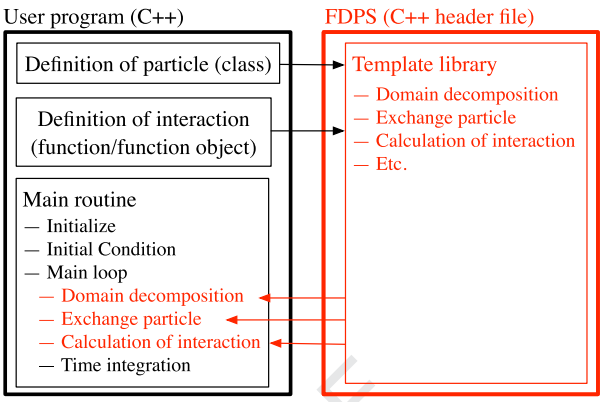


Fig. 1. Schematic illustration of the user interface design of the previous version of FDPS. Note that this figure is the same as figure 1 in Iwasawa et al. (2016).

perform procedure (4), i.e., interaction calculation, using fast algorithms for both long- and short-range forces. Thus, users of FDPS can perform these procedures just by calling FDPS APIs. The other parts of users' codes including procedure (5) are implemented by the users. These parts do not involve parallelization and thus users of FDPS do not need to consider the parallelization of the codes explicitly.

Figure 1 schematically illustrates a typical structure of a parallel particle simulation code developed using FDPS and how a user uses FDPS. The user program first defines particle data set and interaction using class and function in the C++ programming language, respectively. Thus, the user program must be written in C++. This is because, as shown in the right-hand portion of figure 1, FDPS is written in C++, for the reason given in subsection 2.2. FDPS provides APIs to perform domain decomposition, particle exchange, and calculation of interaction. They are defined as function templates in C++ (see subsection 2.2). These function templates are combined with the definitions of particle and interaction in order to instantiate functions that perform domain decomposition, particle exchange, and calculation of interaction in the user program. The instantiated functions are used in the main loop of the user program, as shown in the lower left-hand portion of figure 1. FDPS accepts arbitrary types of particle and interaction. Thus, we can use FDPS to develop any kinds of particle simulation codes.

In the next section, we will explain how the user interface design described here is realized using features in the C++ programming language.

2.2 Implementation details of FDPS

One development goal of FDPS is to make it possible for FDPS to support arbitrary types of particle and interaction. In order to achieve this without a decrease in performance of simulation codes, FDPS is implemented using

the template feature in the C++ programming language. Roughly speaking, this feature allows functions and classes to be described by the use of general data types (for detail, see descriptions in the ISO C++ standard [ISO/IEC 14882:2014]). Functions and classes described using the template feature are called function templates and class templates, respectively. All general data types used in a function template or class template are replaced by specific data types given through special arguments, called template arguments, at the compile-time of a program. Hence, there are no unknown data types at the compile-time, allowing compilers to optimize function templates aggressively. Thus, using the template feature, we can make a high-performance library for general particle types. These are the reasons why we adopted the C++ programming language to develop FDPS.

In order to make it possible for FDPS to provide the APIs that perform domain decomposition, particle exchange, and interaction calculation for arbitrary types of particle data and interaction, we adopt an internal structure for FDPS as described below. First, all functions in FDPS API are implemented as function templates to manipulate general particle types. As a result, users of FDPS must implement particle data as C++ classes and pass them to FDPS API through template arguments when using the API. Secondly, we require that particle classes defined by FDPS users must have some public member functions that have specific names and specific functionalities. This is because FDPS, for example, must know which member variable represents the position of a particle to perform domain decomposition and particle exchange. We solve this by requiring that a particle class has a public member function, named `getPos`, that returns the position of a particle. For similar reasons, there are other public member functions that a user-defined particle class should have. For a complete list, one can refer to the specification document for FDPS. Thirdly, we require that particle–particle interactions must be implemented as functions or functors that have a specific interface. With this, we can implement FDPS without knowing the contents of interaction functions. This requirement does not restrict types of interactions; we can still implement arbitrary types of interaction functions as long as they have the specified interface.

The FDPS API is actually provided as the public member functions of two class templates, `ParticleSystem` and `TreeForForce`, and one (non-template) class, `DomainInfo`. Hereinafter, they are collectively called FDPS classes. Each of them is used to make FDPS do a certain task. To be more precise, instances of FDPS classes `ParticleSystem`, `DomainInfo`, and `TreeForForce` are used, respectively, for particle exchange, domain decomposition, and interaction calculation.

```
1  #include <particle_simulator.hpp>
2
3  // Define particles and interactions [omitted]
4
5  int main(int argc, char *argv[]) {
6
7     // Initialize FDPS [omitted]
8
9     PS::ParticleSystem<Tfp> psys;
10    PS::DomainInfo dinfo;
11    PS::TreeForForceLong<Tforce,Tepi,Tepj>::Monopole tree;
12
13    // Set initial condition [omitted]
14
15    dinfo.decomposeDomainAll(psys);
16    psys.exchangeParticle(dinfo);
17    tree.calcForceAllAndWriteBack(interact_func_ep_ep,
18                                  interact_func_ep_sp,
19                                  psys,
20                                  dinfo);
21
22 }
```

Fig. 2. Example of a C++ code which uses FDPS.

To illustrate the usage of FDPS API, we show in figure 2 an example of a C++ code that uses FDPS. At the first line of the example, the file `particle_simulator.hpp` is included. This is the file where all FDPS classes are implemented. Hence, all user programs include this file. As explained above, in order to use the functionalities of FDPS, we first create instances of FDPS classes. This is done at lines 9–10 in the example, where "`PS`" is the name space in which FDPS classes are defined and the words separated by commnas in the angle brackets associated with FDPS classes `ParticleSystem` and `TreeForForce` are template arguments. `ParticleSystem` class takes `FullParticle` (`FP`) class as a template argument, where `FP` class is a class that contains all the information about a particle necessary to perform simulations as member variables. `Tfp` in the example is the user-defined `FP` class. In the example, `TreeForForceLong` class is used. It is a `TreeForForce` class specifically for long-range force and takes the following three particle classes as template arguments—the first one is `Force` class which contains the quantities of an *i*-particle used to store the results of the calculations of interactions as member variables; the second and third ones are `EssentialParticleI` (`EPI`) and `EssentialParticleJ` (`EPJ`) classes which contain the minimum sets of the quantities of *i*- and *j*-particles necessary to calculate interactions as member variables, where an *i*-particle is a particle that receives a force and a *j*-particle is a particle that exerts a force. In the example, `Tforce`, `Tepi`, and `Tepj` are the user definitions of these three classes. Hereinafter, we call the four kinds of particle classes described above user-defined types.[2] After creating the instances of FDPS classes, we can use the FDPS API by calling public member functions of these instances as in lines 15–20.

[2] The user-defined types `EPI` and `EPJ` are introduced in order to reduce the requirements for the communication bandwidth and the memory bandwidth when performing interaction calculation.

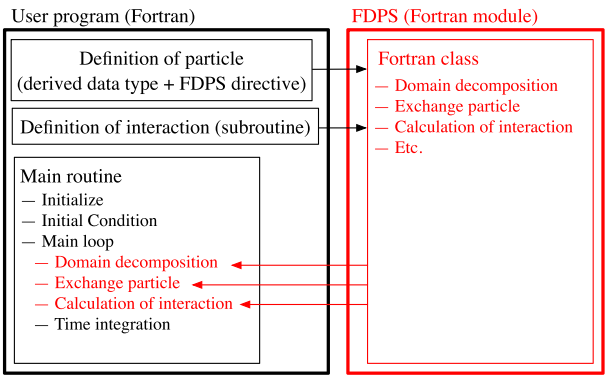


Fig. 3. Schematic illustration of the relation between a user's Fortran program and the FDPS Fortran interface.

More details about the previous version of FDPS can be found in Iwasawa et al. (2016).

3 Design, usage, and implementation of Fortran interface

In this section, we describe the details of our Fortran interface. In subsection 3.1, we describe the user interface design of Fortran interface. In subsection 3.2, we explain the usage of Fortran interface using a specific example of a Fortran program. In subsection 3.3, we describe the implementation details of Fortran interface. In the following, we call the part of FDPS corresponding to the previous version of FDPS the core part.

3.1 User interface design of Fortran interface

The Fortran interface to FDPS developed in this study wraps the core part of FDPS and hence has essentially the same user interface design as shown in figure 3. The user program first defines particle and interaction using derived data type and subroutine in Fortran 2003, respectively. Thus, the user program must be written in Fortran 2003. This is because, as will be explained in subsection 3.3, the Fortran interface uses features in Fortran 2003 in order to make Fortran interoperate with C++. In the definition of particles, the user must describe complementary information about particles using FDPS directives, which are Fortran comments having special format and are introduced by us (for details, see subsection 3.2). This procedure corresponds to define specific public member functions in particle class when using FDPS from C++ (see subsection 2.2). The Fortran interface to FDPS is written in a Fortran module, as shown in the right-hand portion of figure 3, and all of the FDPS API are defined as public member functions of a Fortran class for manipulating the core part of FDPS. Corresponding with the APIs in the core part of FDPS, there are APIs for domain decomposition, particle exchange, and calculation of interaction in Fortran interface. The user program calls these APIs in the main loop, as shown in the lower left-hand portion of figure 3.

The Fortran interface is also designed to accept arbitrary types of particle and interaction as with the core part of FDPS. In the core part of FDPS, this is realized using the template feature in C++ as described in subsection 2.2. Fortran does not have such a feature. In the Fortran interface developed in this study, we realize it via an automatic generation of source programs of Fortran interface specifically for particle types given by users based on the information given through FDPS directives (the reason why this kind of mechanism is needed will be described in subsection 3.3). The generation of source programs of Fortran interface is done by a PYTHON script provided by us. The users of Fortran interface must run this script to use Fortran interface.

In the following subsections, we first explain the usage of Fortran interface in subsection 3.2, and then describe the implementation details of Fortran interface in subsection 3.3.

3.2 Usage of Fortran interface

In this section, we explain using a sample code how a user can write an application program using Fortran interface. The basic procedures to develop codes using Fortran interface are as follows.

I. Define a particle using a Fortran-derived data type.
II. Generate Fortran interface programs.
III. Define an interaction using a Fortran subroutine.
IV. Develop the main part of a code using the FDPS API given through Fortran interface.

Thus, the development procedures are similar to the case in which we develop codes in C++ using FDPS (see section 2). The only difference is the existence of Procedure II. There is no difficulty in this procedure because a user only has to execute the auto-generation script provided by us as stated earlier.

The sample code used is a gravitational N-body simulation code and is essentially the same as the one shown in subsection 2.2 in Iwasawa et al. (2016), except that the code shown here is written in Fortran. The behavior of the code is very simple: it sets the initial condition by reading particle data from a binary file, and then it follows the time evolution of the system by integrating Newton's equations of motion of particles with the leap-frog method. The gravitational forces acting on particles are calculated using FDPS with the tree algorithm in which the gravitational forces from distant particles are approximated as those from a superparticle with multipole moments. For simplicity, we

use the center-of-mass approximation. In this case, the gravitational acceleration of particle i is evaluated as the sum of contributions from other particles and superparticles:

$$\frac{d^2\boldsymbol{r}_i}{dt^2} = \sum_{\substack{j=1 \\ (j\neq i)}} \frac{Gm_j(\boldsymbol{r}_j - \boldsymbol{r}_i)}{\left(|\boldsymbol{r}_j - \boldsymbol{r}_i|^2 + \epsilon_i^2\right)^{3/2}} + \sum_{j=1} \frac{Gm'_j(\boldsymbol{r}'_j - \boldsymbol{r}_i)}{\left(|\boldsymbol{r}'_j - \boldsymbol{r}_i|^2 + \epsilon_i^2\right)^{3/2}}, \quad (1)$$

where $\boldsymbol{r}_i$, $\boldsymbol{r}_j$, $\boldsymbol{r}'_j$ are the positions of particle i, particle j, and superparticle j, respectively. m_j and m'_j are the masses of particle j and superparticle j, respectively. ϵ_i the gravitational softening of particle i, and G is the gravitational constant.

Figure 4 shows the sample code, which runs either in a single-process execution case or an MPI parallel environment. The code consists of three parts: the definition of the particle (lines 7–18), the definition of the interaction functions (lines 22–88), and the other parts including numerical integration and I/O (lines 92–205). These three parts correspond to Procedures I, III, IV described at the beginning of this section. In the following, we explain each procedure in detail.

3.2.1 Defining particle types

Users of FDPS must first define particles as derived data types in Fortran (Procedure I). In the sample code, the particle data is defined as `full_ptcl` type in lines 7–18. It has the following member variables: `id` (particle identification number), `mass` (m_i), `eps` (ϵ_i), `pos` ($\boldsymbol{r}_i$), `vel` ($\boldsymbol{v}_i$; the velocity of particle i), `pot` (ϕ_i; the gravitational potential at $\boldsymbol{r}_i$), and `acc` ($d\boldsymbol{v}_i/dt$).

As we will describe in subsection 3.3, the particle data type must be interoperable with C because Fortran interface uses the C interoperability feature in Fortran 2003 to exchange data between Fortran programs and the core part of FDPS. In order to be interoperable with C, a derived data type must satisfy the following conditions.

- It has the `bind(c)` attribute.
- The data types of all member variables must be interoperable with C.

The first condition is satisfied if a derived data type is declared with the `bind(c)` keyword (line 7). As for the second condition, Fortran 2003 or later offers data types corresponding to primitive data types in the C language such as `int`, `float`, `double`, etc., through the `iso_c_binding` module. The second condition is fulfilled if we define member variables using these data types. `integer(kind=c_long_long)` is one such example. `type(fdps_f64vec)` type used at lines 14, 15, and 17, which is defined in module `fdps_vector` and represents a space vector, is also interoperable with C because their member variables are all interoperable with C.

Users of FDPS must write FDPS directives within the definition part of the particles. One can see a variety of Fortran comments which begin with `!$fdps` in the sample code. They are FDPS directives. FDPS directives are used to pass complementary information about particles, e.g., which member variable represents a physical quantity that FDPS must know, to FDPS. FDPS directives can be classified into the following three types.

(a) Directive specifying the type of a particle.
(b) Directive specifying which member variable represents which physical quantity.
(c) Directive specifying the way the data operation is performed in FDPS.

In the following, we explain each type of directive.

Directive (a) is used to specify which user-defined type a derived data type corresponds to out of `FP`, `EPI`, `EPJ`, and `Force` described in subsection 2.2. A derived data type can serve multiple user-defined types. Directive (a) in this sample code is written at line 7:

```
!$fdps FP,EPI,EPJ,Force
```

With this, `full_ptcl` type can be used as any of user-defined types in the sample code.

Directive (b) is used to specify which member variable represents the physical quantity required by FDPS. In the present case, we need to specify the mass and the position of a particle (the mass is required to calculate the monopole information of superparticles). Therefore, the (b) directives are written as follows (see lines 12 and 14):

```
  real(kind=c_double) mass !$fdps charge
type(fdps_f64vec) :: pos !$fdps position
```

With these, FDPS recognizes that the variables `mass` and `pos` represent the mass and the position of a particle.

In the sample code, three directives of type (c) are written in lines 8–10 as follows.

```
!$fdps copyFromForce full_ptcl (pot,pot)
  (acc,acc)
!$fdps copyFromFP full_ptcl (mass,mass)
   (eps,eps) \ (pos,pos)
!$fdps clear id=keep, mass=keep, eps=keep, \
  pos=keep, vel=keep
```

where the symbol \ is used to represent that the current line continues to the next line because of space limitations; it cannot be used in an actual code (FDPS directive must be written within a line). The first two directives containing keywords `copyFromForce` and `copyFromFP` specify how data copy is performed between different user-defined types. The former specifies the method of data copy from `Force` type to `FP` type and the latter specifies that from `FP` type to `EPI` type or `EPJ` type. The directive including the

COLOUR ONLINE, B&W IN PRINT

```fortran
module user_defined_types
   use, intrinsic :: iso_c_binding
   use fdps_vector
   use fdps_super_particle
   implicit none

   type, public, bind(c) :: full_ptcl !$fdps FP,EPI,EPJ,Force
      !$fdps copyFromForce full_ptcl (pot,pot) (acc,acc)
      !$fdps copyFromFP full_ptcl (mass,mass) (eps,eps) (pos,pos)
      !$fdps clear id=keep, mass=keep, eps=keep, pos=keep, vel=keep
      integer(kind=c_long_long) :: id
      real(kind=c_double)  mass !$fdps charge
      real(kind=c_double) :: eps
      type(fdps_f64vec) :: pos !$fdps position
      type(fdps_f64vec) :: vel
      real(kind=c_double) :: pot
      type(fdps_f64vec) :: acc
   end type full_ptcl

   contains

   subroutine calc_gravity_pp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
      integer(c_int), intent(in), value :: n_ip,n_jp
      type(full_ptcl), dimension(n_ip), intent(in) :: ep_i
      type(full_ptcl), dimension(n_jp), intent(in) :: ep_j
      type(full_ptcl), dimension(n_ip), intent(inout) :: f
      integer(c_int) :: i,j
      real(c_double) :: eps2,poti,r3_inv,r_inv
      type(fdps_f64vec) :: xi,ai,rij

      do i=1,n_ip
         eps2 = ep_i(i)%eps * ep_i(i)%eps
         xi = ep_i(i)%pos
         ai = 0.0d0; poti = 0.0d0
         do j=1,n_jp
            rij%x  = xi%x - ep_j(j)%pos%x
            rij%y  = xi%y - ep_j(j)%pos%y
            rij%z  = xi%z - ep_j(j)%pos%z
            r3_inv = rij%x*rij%x + rij%y*rij%y + rij%z*rij%z &
                   + eps2
            r_inv  = 1.0d0/sqrt(r3_inv)
            r3_inv = r_inv * r_inv
            r_inv  = r_inv * ep_j(j)%mass
            r3_inv = r3_inv * r_inv
            ai%x   = ai%x - r3_inv * rij%x
            ai%y   = ai%y - r3_inv * rij%y
            ai%z   = ai%z - r3_inv * rij%z
            poti   = poti - r_inv
         end do
         f(i)%pot = f(i)%pot + poti
         f(i)%acc = f(i)%acc + ai
      end do

   end subroutine calc_gravity_pp

   subroutine calc_gravity_psp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
      integer(c_int), intent(in), value :: n_ip,n_jp
      type(full_ptcl), dimension(n_ip), intent(in) :: ep_i
      type(fdps_spj_monopole), dimension(n_jp), intent(in) :: ep_j
      type(full_ptcl), dimension(n_ip), intent(inout) :: f
      integer(c_int) :: i,j
      real(c_double) :: eps2,poti,r3_inv,r_inv
      type(fdps_f64vec) :: xi,ai,rij

      do i=1,n_ip
         eps2 = ep_i(i)%eps * ep_i(i)%eps
         xi = ep_i(i)%pos
         ai = 0.0d0; poti = 0.0d0
         do j=1,n_jp
            rij%x  = xi%x - ep_j(j)%pos%x
            rij%y  = xi%y - ep_j(j)%pos%y
            rij%z  = xi%z - ep_j(j)%pos%z
            r3_inv = rij%x*rij%x + rij%y*rij%y + rij%z*rij%z &
                   + eps2
            r_inv  = 1.0d0/sqrt(r3_inv)
            r3_inv = r_inv * r_inv
            r_inv  = r_inv * ep_j(j)%mass
            r3_inv = r3_inv * r_inv
            ai%x   = ai%x - r3_inv * rij%x
            ai%y   = ai%y - r3_inv * rij%y
            ai%z   = ai%z - r3_inv * rij%z
            poti   = poti - r_inv
         end do
         f(i)%pot = f(i)%pot + poti
         f(i)%acc = f(i)%acc + ai
      end do

   end subroutine calc_gravity_psp

end module user_defined_types

subroutine f_main()
   use fdps_module
   use user_defined_types
   implicit none
   double precision, parameter :: time_end=10.0d0
   double precision, parameter :: dt=1.0d0/128.0d0
   integer :: i,j,k,ierr
   integer :: psys_num,dinfo_num,tree_num
   character(len=64) :: tree_type
   double precision :: time_sys=0.0d0
   type(fdps_controller) :: fdps_ctrl
   call fdps_ctrl%PS_Initialize()
   call fdps_ctrl%create_dinfo(dinfo_num)
   call fdps_ctrl%init_dinfo(dinfo_num)
   call fdps_ctrl%create_psys(psys_num,'full_ptcl')
   call fdps_ctrl%init_psys(psys_num)
   tree_type="Long,full_ptcl,full_ptcl,full_ptcl,Monopole"
   call fdps_ctrl%create_tree(tree_num,tree_type)
   call fdps_ctrl%init_tree(tree_num,0)
   call read_IC(fdps_ctrl,psys_num)
   call calc_gravity(fdps_ctrl,psys_num,dinfo_num,tree_num)
   do
      call kick(fdps_ctrl,psys_num,0.5d0*dt)
      time_sys=time_sys+dt
      call drift(fdps_ctrl,psys_num,dt)
      call calc_gravity(fdps_ctrl,psys_num,dinfo_num,tree_num)
      call kick(fdps_ctrl,psys_num,0.5d0*dt)
      if(time_sys >= time_end) exit
   end do
   call fdps_ctrl%PS_Finalize()
end subroutine f_main

subroutine calc_gravity(fdps_ctrl,psys_num,dinfo_num,tree_num)
   use fdps_module
   use user_defined_types
   implicit none
   type(fdps_controller), intent(IN) :: fdps_ctrl
   integer, intent(IN) :: psys_num,dinfo_num,tree_num
   type(c_funptr) :: pfunc_ep_ep,pfunc_ep_sp
   call fdps_ctrl%decompose_domain_all(dinfo_num,psys_num)
   call fdps_ctrl%exchange_particle(psys_num,dinfo_num)
   pfunc_ep_ep=c_funloc(calc_gravity_pp)
   pfunc_ep_sp=c_funloc(calc_gravity_psp)
   call fdps_ctrl%calc_force_all_and_write_back(tree_num,&
                                                pfunc_ep_ep,&
                                                pfunc_ep_sp,&
                                                psys_num,&
                                                dinfo_num)
end subroutine calc_gravity

subroutine kick(fdps_ctrl,psys_num,dt)
   use fdps_vector
   use fdps_module
   use user_defined_types
   implicit none
   type(fdps_controller), intent(IN) :: fdps_ctrl
   integer,intent(IN) :: psys_num
   double precision,intent(IN) :: dt
   integer :: i,nptcl_loc
   type(full_ptcl), dimension(:), pointer :: ptcl
   nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
   call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
   do i=1,nptcl_loc
      ptcl(i)%vel = ptcl(i)%vel + ptcl(i)%acc*dt
   end do
   nullify(ptcl)
end subroutine kick

subroutine drift(fdps_ctrl,psys_num,dt)
   use fdps_vector
   use fdps_module
   use user_defined_types
   implicit none
   type(fdps_controller), intent(IN) :: fdps_ctrl
   integer, intent(IN) :: psys_num
   double precision, intent(IN) :: dt
   integer :: i,nptcl_loc
   type(full_ptcl), dimension(:), pointer :: ptcl
   nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
   call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
   do i=1,nptcl_loc
      ptcl(i)%pos = ptcl(i)%pos + ptcl(i)%vel*dt
   end do
   nullify(ptcl)
end subroutine drift

subroutine read_IC(fdps_ctrl,psys_num)
   use fdps_module
   use user_defined_types
   implicit none
   type(fdps_controller), intent(IN) :: fdps_ctrl
   integer, intent(IN) :: psys_num
   character(len=16), parameter :: root_dir="input_data"
   character(len=16), parameter :: file_prefix="proc"
   integer :: i,myrank,nptcl_loc
   character(len=64) :: fname,proc_num
   type(full_ptcl), dimension(:), pointer::ptcl
   myrank = fdps_ctrl%get_rank()
   write(proc_num,"(i5.5)")myrank
   fname = trim(root_dir)//"/" &
         //trim(file_prefix)//proc_num//".dat"
   open(unit=9,file=trim(fname),action='read',form='unformatted',&
        access='stream',status='old')
      read(9)nptcl_loc
      call fdps_ctrl%set_nptcl_loc(psys_num,nptcl_loc)
      call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
      do i=1,nptcl_loc
      read(9)ptcl(i)%id,ptcl(i)%mass,ptcl(i)%eps, &
             ptcl(i)%pos%x,ptcl(i)%pos%y,ptcl(i)%pos%z, &
             ptcl(i)%vel%x,ptcl(i)%vel%y,ptcl(i)%vel%z
   end do
   close(unit=9)
   nullify(ptcl)
end subroutine read_IC
```

Fig. 4. Sample code for *N*-body simulation developed by Fortran interface to FDPS.

keyword `clear` is used to specify how to initialize `Force` type before starting the calculations of interactions. Q9

The details of FDPS directives can be found in the specification document for Fortran interface in the distributed FDPS package.

3.2.2 Generating Fortran interface

The next step is the generation of Fortran interface programs (Procedure II), which can be done simply by executing the auto-generation script `gen_ftn_if.py` provided by us in the command line as follows.

```
./$(FDPS_LOC)/scripts/gen_ftn_if.py sample_
   code.F90
```

where `$(FDPS_LOC)` represents the PATH of the top directory of the FDPS library. If the auto-generation succeeds, all the files described in sub-subsection 3.3.2 are created at the current directory.

3.2.3 Defining interaction functions

Users of FDPS must define interactions as subroutines in Fortran (Procedure III). In the sample code, two subroutines are defined—one for the gravitational interaction between particles (lines 22–54) and the other for the interaction between particles and superparticles (lines 56–88).

The interaction functions must also be interoperable with C because their function pointers are passed to the core part of FDPS using the C interoperability feature in Fortran 2003. To be interoperable with C, a subroutine must satisfy the following conditions.

- It has the `bind(c)` attribute.
- The data types of all variables used in the subroutine must be interoperable with C.

These conditions are the same as those for derived data types which are interoperable with C (see sub-subsection 3.2.1). Hence, we can clear these requirements in the same way: first add the `bind(c)` keyword after the argument list of the subroutine (see lines 22 and 56), and define variables using data types which are interoperable with C.

The interaction functions must have the following arguments: (from the beginning) the array of *i* particles (`ep_i`), the number of *i* particles (`n_ip`), the array of *j* particles or superparticles (`ep_j`), the number of *j* particles or superparticles (`n_jp`), and the array of variables that stores the calculated interaction on *i* particles (`f`). Due to the specification of the core part of FDPS, the numbers of *j* particles and superparticles must be pass-by-value arguments. Hence, the `value` keyword is needed in the type declaration statements for these arguments (see lines 21 and 57).

FDPS does not take care about the optimizations of the interaction functions. Thus, users of FDPS must optimize the interaction functions by themselves in order to achieve high efficiency. For simplicity, the interaction functions in this sample code are implemented without any optimization techniques. Some typical optimization techniques are presented in section 4.

3.2.4 Developing the main part of a user code

Users of FDPS must implement the main routine of a user code within a subroutine named `f_main()` for the reason explained in subsection 3.3 (Procedure IV). `f_main()` of the sample code is implemented in lines 92–122 and it consists of the following steps.

(1) Initialize FDPS (line 103).
(2) Create and initialize instances of FDPS classes (lines 104–110).
(3) Read particle data from a file (line 111).
(4) Calculate the gravitational forces on all the particles at the initial time (line 112).
(5) Integrate the orbits of all the particles with the leap-frog method (lines 113–120).
(6) Finalize FDPS (line 121).

In the following, we first explain the variable declaration part of `f_main()`, then we explain each step in detail.

In the Fortran interface, FDPS API is provided as type-bound procedures of Fortran 2003 class `fdps_controller` defined in the module `fdps_module`. This module is defined in `FDPS_module.F90`, one of the source programs of Fortran interface. In order to use FDPS API, we first need to make this module accessible from `f_main()`, and then we should create an instance of this class. These things are done at lines 93 and 102, respectively. Thus, in this sample code, FDPS API is used by calling type-bound procedures of class instance `fdps_ctrl`.

In step 1, API `ps_initialize` is called. This procedure initializes MPI and OpenMP libraries if they are used. If not, it does nothing.

In step 2, we create and initialize three instances of FDPS classes `ParticleSystem`, `DomainInfo`, and `TreeForForce` by calling type-bound procedures whose names contain '`create`' or '`init`', which, as the names suggest, create and initialize instances, respectively. Each of these procedures takes an integer argument (`psys_num`, `dinfo_num`, and `tree_num` in the sample code) because all instances of FDPS classes are identified by descriptors in integer variables in the Fortran interface. API `create_psys` receives the name of a derived data type representing `FP` as the second argument, and it creates an instance of `ParticleSystem<Tfp>`, where `Tfp` is the name of the C++ class corresponding

to the derived data type having the specified name. In the sample code, `full_ptcl` is specified. Note that this API accepts only the names of derived data types qualified by FDPS directive of type (a) explained in sub-subsection 3.2.1. API `create_tree` receives the type of an instance of class `TreeForForce` as the second argument. The type information must be given by a single string of characters that consists of five strings of characters delimited by commas. The first field represents the type of force (long-range or short-range), which is followed by the three names of derived data types corresponding to `Force`, `EPI`, and `EPJ` types, and the final field shows the subtype of tree for long-/short-range force. In this example, we calculate the gravitational forces as the long-range force and use the monopole approximation. In addition, `full_ptcl` type is used as `Force`, `EPI`, and `EPJ` types. Therefore, "`Long,full_ptcl,full_ptcl,full_ptcl,Monopole`" is specified.

In step 3, we read particle data from a binary file, using subroutine `read_IC` defined in lines 178–205. In the framework of the Fortran interface, all the particle data is stored in a C++ program, which is one of the Fortran interface programs (see subsection 2.2). In order to set up the particle data from a Fortran program, we should take the following steps.

(i) Create and initialize an instance of `ParticleSystem` class (this is already done at step 2).
(ii) Allocate memory for this instance. In the sample code, this is done by calling API `set_nptcl_loc`. This API allocates a memory for the instance enough to store an array of the particles of size specified by the second argument (i.e., `nptcl_loc` in this sample).
(iii) Get the pointer to this instance. This is done by calling API `get_psys_fptr`. Because this API returns the pointer corresponding to the instance specified by its first argument (i.e., `psys_num` in this sample), we have to prepare the pointer of the same type. In the present case, the pointer for an array of `full_type` type should be prepared. After the pointer is set, we can use this pointer like an array.
(iv) Set the particle data using the pointer. This is done in lines 199–201.

In step 4, the interaction calculation is performed through the subroutine `calc_gravity`, which is defined in lines 124–140. This subroutine consists of the following three operations.

(i) Perform domain decomposition. This is done by calling API `domain_decomposition` (line 131). The first argument of this API is an integer variable corresponding to an instance of `DomainInfo` class. The second one is an integer variable corresponding to an instance of `ParticleSystem` class, which is needed because the positional information of the particles is required to perform domain decomposition.
(ii) Perform particle exchange. This is done by calling API `exchange_particle` (line 132). This API takes an integer variable corresponding to an instance of `DomainInfo` class as the first argument. The second argument is an integer variable corresponding to an instance of `ParticleSystem` class. This is required because the information of subdomains is necessary to perform particle exchange.
(iii) Perform the calculation of interactions. This is done by calling API `call_force_all_and_write_back` (line 135). The number of arguments of this API depends on the type of force. In the present case, it takes five arguments because of the long-range interaction. The definitions of the first, fourth, and fifth arguments should be obvious, and hence we do not explain them here. The second and third arguments are the function pointers for the subroutines that calculate particle–particle interaction and particle–superparticle interaction. The function pointers must be obtained by the intrinsic function `c_funloc`, and be stored in variables of `c_funptr` type. In the sample code, `calc_gravity_pp` and `calc_gravity_psp` are used for the calculation of interactions.

In step 5, the time integration is performed through the subroutines `kick` and `drift`, which are defined, respectively, in lines 142–158 and 160–176. Both subroutines have the same structure: first get the pointer to the particle data stored in the C++ side of Fortran interface programs using API `get_psys_fptr`, and then update the particle data using this pointer.

In step 6, API `ps_finalize` is called. It calls the `MPI_Finalize` function if MPI is used.

In this section, we have described how an application program can be written using Fortran interface and have shown that researchers can develop particle-based simulation codes by writing Fortran codes only. Thus, learning the C++ programming language is totally unnecessary in order to use FDPS.

3.3 Implementation details of Fortran interface

In this section, we describe the implementation details of Fortran interface. In sub-subsection 3.3.1, we describe why we need a generation of Fortran interface. In sub-subsection 3.3.2, we describe the file structure of Fortran interface programs and the role of each file.

3.3.1 Necessity of generation of Fortran interface

One difficulty in developing a Fortran interface to FDPS is that the template feature in the C++ programming language cannot be used directly from Fortran (Fortran is not designed to interoperate with the C++ programming language). As described earlier, the template feature is essential for FDPS to support arbitrary types of particle and interaction. Hence, we need to work around this problem in some way in order to make Fortran interface support arbitrary types of particle and interaction.

In this study, we solve this problem by making use of (i) the fact that Fortran can interoperate with the C programming language through the use of the `iso_c_binding` module introduced in Fortran 2003, (ii) the fact that C++ functions can be called from C programs using `extern "C"` modifier, and (iii) auto generation of source codes. The outline of our solution is as follows. As a result of (i) and (ii), we can make Fortran interoperate with C++ via C. Therefore, it is also possible to use an FDPS library (a set of functions to manipulate FDPS) made for a *specific* particle class from a Fortran code through a C interface to the library. However, what we want to realize is to enable researchers to use an FDPS library for an *arbitrary* particle class from a Fortran code. The only way to realize this would be to generate an FDPS library for a given particle class and use it from a Fortran code. Another requirement in designing a Fortran interface is that researchers must be able to develop codes via Fortran only so that they will not need to invest much time in learning C++. To make this possible, we must generate an FDPS library based on some Fortran data structures representing particles, instead of based on C++ classes. It is natural to use Fortran's derived data type, which is similar to structure in C, to define a particle. Thus, we adopted the following solution: first define a particle as a derived data type in Fortran. Then, generate a C++ class corresponding to this derived data type. Finally, generate an FDPS library for this particle class. The generation of required C++ classes and a library is automatically done by a script provided by us. Hence, researchers do not have to be concerned about it.

The one remaining problem is how to generate a C++ class from a given Fortran derived data type. In FDPS, C++ class representing particles must have specific public member functions such as `getPos` as described in section 2. Fortran 2003 supports object-oriented programming and offers classes for it, which are derived data types having type-bound procedures and are essentially the same as classes in C++. Thus, a simple way to generate a C++ class is to define a particle as a class in Fortran 2003 that has public member functions required by FDPS and to convert it to a C++ class. However, we cannot take this approach because Fortran 2003 Standard specifies that derived data types must not have member procedures/functions to be interoperable with C. In order to solve this problem, we introduce FDPS directives. They are used by users of FDPS to pass all information necessary to generate public member functions of the corresponding C++ classes to the auto-generation script described above. For example, the auto-generation script needs to know which member variable of a given Fortran derived data type represents the position of a particle in order to generate a public member function `getPos` in the corresponding C++ class. This information is passed to the script by adding FDPS directive `!$fdps position` to the corresponding member variable of the Fortran derived data type. In other words, our auto-generation script is designed to generate `getPos` in the corresponding C++ class using a member variable of a given Fortran derived data type indicated by FDPS directive `!$fdps position`. In this way, we can pass all of the necessary information to the auto-generation script. In subsection 3.2, we have given a rough explanation of the FDPS directive for simplicity, but this kind of thing is indeed done.

The generation of a Fortran interface can be summarized schematically as shown in figure 5: First, researchers define particles as derived data types in Fortran with FDPS directives (Step ①in figure 5). Next, the auto-generation script generates C++ classes corresponding to the given Fortran derived data types (Step ②). Using the information given by FDPS directives, the script can generate public member functions required by FDPS. Then, the script generates an FDPS library specifically for these C++ classes (Step ③). The generated library is not a template library, but a normal library that can be called in the manner of C. Finally, the script also generates a Fortran module to manipulate the generated library (Step ④). Users of FDPS can use FDPS through this module (Step ⑤).

3.3.2 Structure of Fortran interface programs

In this section, we explain the file structure of the generated programs and their internal structures.

Figure 6 is a brief summary of the file structure of the Fortran interface programs and their roles. Four files enclosed by the dashed line (`FDPS_module.F90`, `FDPS_ftn_if.cpp`, `FDPS_Manipulators.cpp`, `main.cpp`) are the files to be generated by the script, and the file `f_main.F90` corresponds to user programs. In the files enclosed by the dotted line (`FDPS_vector.F90`, `FDPS_matrix.F90`, `FDPS_super_particle.F90`, etc.), several derived data types are defined, which are needed to define user-defined types and interaction functions in Fortran (see also subsection 3.2). In the following, we explain the role of each interface program.

At first, we explain the roles of `FDPS_Manipulators.cpp` and `main.cpp`. Because the core part of FDPS is

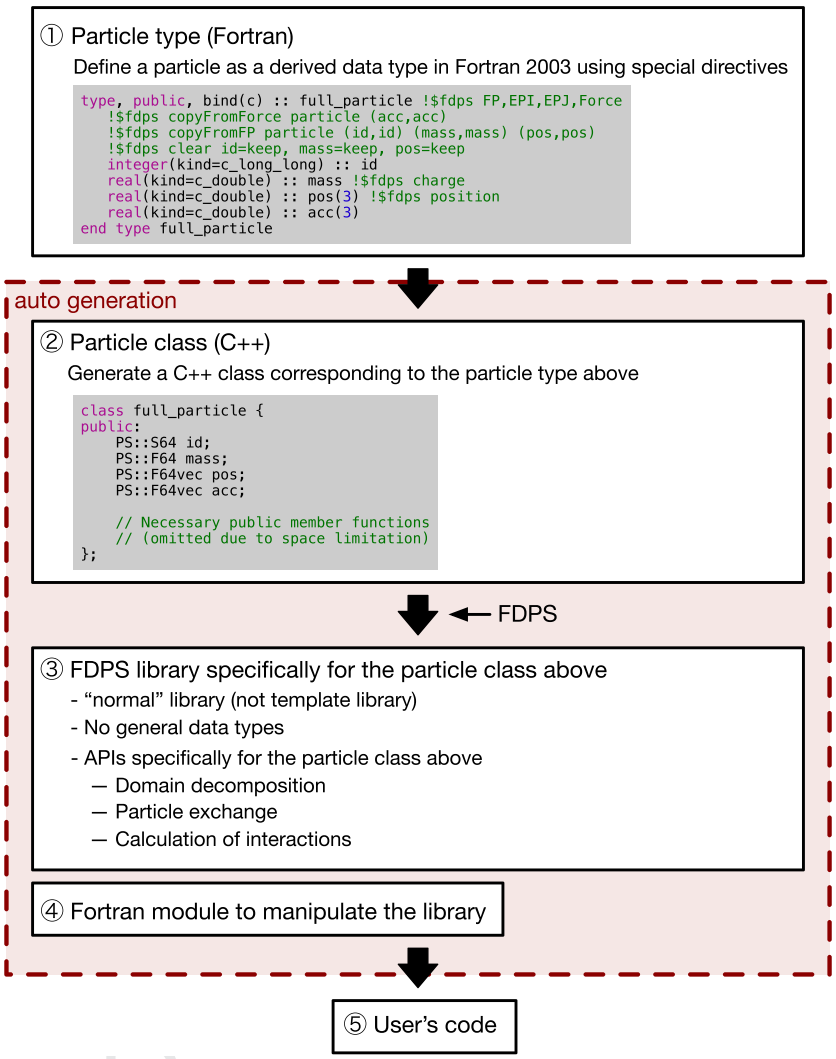


Fig. 5. Schematic illustration of our solution to enable the use of an arbitrary type of particle in a Fortran interface.

written in C++, all the C++ instances of FDPS classes described in section 2 (i.e., `DomainInfo`, `ParticleSystem`, and `TreeForForce` classes) must be created and be managed in C++ codes. This task is performed by `FDPS_Manipulators.cpp`, which corresponds to a library shown in Step ③ in figure 5. For the same reason, we must place the main function of the user program in a C++ file. Thus, `main.cpp` is generated. It calls a Fortran subroutine named `f_main()`. Hence, users should prepare a Fortran subroutine `f_main()` and must implement all parts of the simulation code inside `f_main()`. In the Fortran interface, all the C++ instances created in `FDPS_Manipulators.cpp` are assigned to Fortran's integer variables. Hence, users also need to manage these instances using integer variables.

The file `FDPS_ftn_if.cpp` provides C interfaces to the functions defined in `FDPS_Manipulators.cpp`. These C functions can be called from a Fortran program using the functionalities provided by module `iso_c_binding` in Fortran 2003, as described in sub-subsection 3.3.1. The file `FDPS_module.F90` provides a class in Fortran 2003, named `fdps_controller`, for users, which is used to call the C interface functions described above. All of the FDPS API for Fortran are provided as public type-bound procedures of this class. Therefore, in order to use the FDPS API, we first need to create an instance of this class, and then call its type-bound procedures.

As described above, the particle data is stored in `FDPS_Manipulators.cpp` as explained above. It is actually an array of C structures representing particles (cf. Step ② in figure 5). Fortran module `iso_c_binding` enables us to access it if a derived data type in Fortran corresponding to this C structure is interoperable with C. The Fortran interface makes use of this functionality, and therefore we require that all derived data types that will be used in FDPS

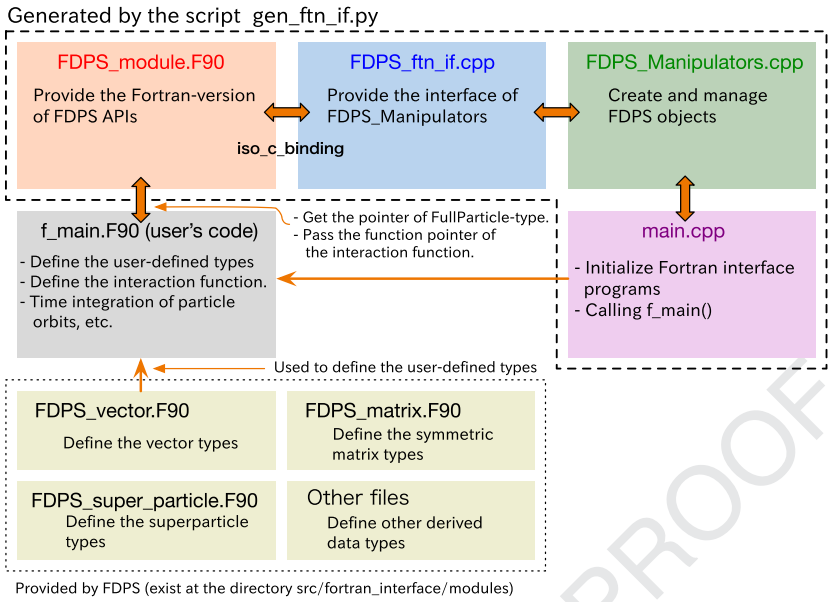


Fig. 6. File structure of Fortran interface programs and their relations to the user's code.

must be interoperable with C. We also use the functionalities of module `iso_c_binding` to pass interaction functions to `FDPS_Manipulators.cpp`. To take this approach, interaction functions must be implemented as subroutines in Fortran and they must be interoperable with C. This is because subroutines are passed in the form of the C addresses (i.e., function pointers in C), which can be obtained only if the subroutines are interoperable with C. These are the reasons why we require C interoperability in subsection 3.2.

4 Performance of application developed by Fortran interface

In this section, we present and discuss the performance of an application developed by Fortran interface to FDPS. Here, we mainly focus on the overhead of the FDPS API-call from a Fortran application because all the tasks of the API are processed in its C++ core and it is expected that the times required to process these tasks do not depend on development language. To this end, we compare the difference between the performances of Fortran and C++ codes for N-body simulations. The performance measurements are carried out in different computer systems to check the dependencies of the performance on CPU architecture and compilers. Table 1 lists the computer systems and the compiler information used in the measurements.

The Fortran code used in the performance measurement is essentially same as the sample code described in subsection 3.2 (figure 4), except for the following two differences. First, we apply standard optimization techniques to interaction functions. Figure 7 shows a Fortran subroutine for the gravitational interaction between particles. In the subroutine, the information of j-particles are stored in local arrays (lines 11–16), and all the calculations in the innermost loop are described using arrays (lines 25–39). These modifications are aiming to facilitate compilers to optimize the code more easily. We use almost the same subroutine for the interaction between particles and superparticles. For a fair comparison, in the C++ version of the

Table 1. Computer systems and compiler information.

System name	K Computer	Oakforest-PACS	Cray XC30
CPU	Fujitsu SPARC 64 VIIIfx	Intel Xeon Phi 7250	Intel Xeon CPU E5-2650 v3
Compiler	Fujitsu compilers (ver. 1.2.0 P-id: L30000-15)	Intel compilers (ver. 17.0.4 20170411)	Intel compilers (ver. 16.0.4 20160811)
Compile options (C++)	`-Kfast -Ksimd=2 -Krestp=all`	`-O3 -ipo -xMIC-AVX512 -no-prec-div`	`-fast -ipo -xCORE-AVX2 -no-prec-div`
Compile options (F)	`-X 03 -Kfast -Ksimd=2 -Free -fs`	`-O3 -ipo -xMIC-AVX512 -no-prec-div`	`-fast -ipo -xCORE-AVX2 -no-prec-div`

```
subroutine calc_gravity_pp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
   integer(c_int), intent(in), value :: n_ip,n_jp
   type(full_particle), dimension(n_ip), intent(in) :: ep_i
   type(full_particle), dimension(n_jp), intent(in) :: ep_j
   type(full_particle), dimension(n_ip), intent(inout) :: f
   integer(c_int) :: i,j
   real(c_double) :: eps2,poti,r3_inv,r_inv
   real(c_double), dimension(3) :: xi,ai,rij
   real(c_double), dimension(n_jp) :: mj
   real(c_double), dimension(3,n_jp) :: xj
   do j=1,n_jp
      mj(j)   = ep_j(j)%mass
      xj(1,j) = ep_j(j)%pos%x
      xj(2,j) = ep_j(j)%pos%y
      xj(3,j) = ep_j(j)%pos%z
   end do
   do i=1,n_ip
      eps2 = ep_i(i)%eps * ep_i(i)%eps
      xi(1) = ep_i(i)%pos%x
      xi(2) = ep_i(i)%pos%y
      xi(3) = ep_i(i)%pos%z
      ai = 0.0d0
      poti = 0.0d0
      do j=1,n_jp
         rij(1) = xi(1) - xj(1,j)
         rij(2) = xi(2) - xj(2,j)
         rij(3) = xi(3) - xj(3,j)
         r3_inv = rij(1)*rij(1) &
                + rij(2)*rij(2) &
                + rij(3)*rij(3) &
                + eps2
         r_inv  = 1.0d0/sqrt(r3_inv)
         r3_inv = r_inv * r_inv
         r_inv  = r_inv * mj(j)
         r3_inv = r3_inv * r_inv
         ai(1)  = ai(1) - r3_inv * rij(1)
         ai(2)  = ai(2) - r3_inv * rij(2)
         ai(3)  = ai(3) - r3_inv * rij(3)
         poti   = poti - r_inv
      end do
      f(i)%pot = f(i)%pot + poti
      f(i)%acc%x = f(i)%acc%x + ai(1)
      f(i)%acc%y = f(i)%acc%y + ai(2)
      f(i)%acc%z = f(i)%acc%z + ai(3)
   end do

end subroutine calc_gravity_pp
```

Fig. 7. Interaction function in the *N*-body simulation code written in Fortran.

N-body simulation code, we use a function with the same structure, which is shown in figure 8.

Secondly, we rewrite all the operations between derived data types using their member variables explicitly. This is because in the combination of Intel Xeon Phi and Intel compilers the executable file which is generated with the highest optimization compile flags does not perform some numerical operations correctly. We believe that this is a problem on the compiler side and this situation will be improved by future upgrade of the compiler. For now, however, we recommend users of Fortran interface to follow this prescription.

Both of the (Fortran and C++) codes solve the cold collapsed problem, which is one of the standard test problems in *N*-body simulation codes. Most of the calculation times are spent on the interaction calculation part, and the overhead due to Fortran interface may be hidden by this part. In order to clarify the overhead cost, we measure the performances for the following two cases: One is the case when the interaction functions are empty (empty cases) and the other is the case when the interaction functions shown in figures 7 and 8 are used (normal cases). Both codes are executed with a single core. In this case, the APIs for domain decomposition and particle exchange do nothing and just increase the overhead. The upper panels of figure 9 show the wall-clock time per step for different numbers of particles per processes for each computer system for the former case, while the lower panels show those for the latter case. As shown in the upper panels, the calculation times in both codes are almost the same for all computer systems we used. Relative differences of the wall-clock times between both codes are $\approx 5\%$. Thus, the overhead of Fortran interface is sufficiently small. The overall performances also show a similar behavior, as shown in the lower panels of figure 9.

```
template <class TParticleJ>
void CalcGravity(const FPGrav * ep_i,
                 const PS::S32 n_ip,
                 const TParticleJ * ep_j,
                 const PS::S32 n_jp,
                 FPGrav * force) {
    PS::F64 eps2 = FPGrav::eps * FPGrav::eps;
    PS::F64  xi[3], ai[3];
    PS::F64  *mj     = (PS::F64  *    )malloc(sizeof(PS::F64) *
     ↪    n_jp);
    PS::F64 (*xj)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * n_jp
     ↪    * 3);
    for (PS::S32 j=0; j<n_jp; j++) {
        mj[j]    = ep_j[j].getCharge();
        xj[j][0] = ep_j[j].getPos()[0];
        xj[j][1] = ep_j[j].getPos()[1];
        xj[j][2] = ep_j[j].getPos()[2];
    }
    for(PS::S32 i = 0; i < n_ip; i++){
        xi[0] = ep_i[i].getPos()[0];
        xi[1] = ep_i[i].getPos()[1];
        xi[2] = ep_i[i].getPos()[2];
        ai[0] = 0.0;
        ai[1] = 0.0;
        ai[2] = 0.0;
        PS::F64 poti = 0.0;
        for(PS::S32 j = 0; j < n_jp; j++){
            PS::F64 rij[3];
            rij[0] = xi[0] - xj[j][0];
            rij[1] = xi[1] - xj[j][1];
            rij[2] = xi[2] - xj[j][2];
            PS::F64 r3_inv = (rij[0] * rij[0]
                             +rij[1] * rij[1]
                             +rij[2] * rij[2])
                           + eps2;
            PS::F64 r_inv  = 1.0/sqrt(r3_inv);
            r3_inv  = r_inv * r_inv;
            r_inv  *= mj[j];
            r3_inv *= r_inv;
            ai[0]  -= r3_inv * rij[0];
            ai[1]  -= r3_inv * rij[1];
            ai[2]  -= r3_inv * rij[2];
            poti   -= r_inv;
        }
        force[i].acc.x += ai[0];
        force[i].acc.y += ai[1];
        force[i].acc.z += ai[2];
        force[i].pot += poti;
    }
    free(mj);
    free(xj);
}
```

Fig. 8. Interaction function in the *N*-body simulation code written in C++, where the symbol ↪ means that the current line is a continuation line to the previous line.

Compared to the case of empty interaction function, the relative differences become small in each computer system. Thus, a code written by Fortran interface shows a nearly identical performance to that written by C++.

5 Example of practical application

In order to demonstrate its ability, we perform a large-scale global planetary ring simulation using a code developed by Fortran interface. Ring particles around a planet interact with each other through mutual gravity and inelastic collision. In the code, we calculate the gravity using the Tree method with an opening angle criterion of $\theta = 0.5$. The inelastic collision is modeled using the soft sphere model following Salo (1995) and Michikoshi and Kokubo (2017).

The initial condition is taken from Michikoshi and Kokubo (2017), who investigated the dynamics of two narrow rings around Centaur Chariklo through changing the radius and mass of ring particles. Among their models, we consider the case of $r_p = 5\,\mathrm{m}$ and $\rho_p/\rho_C = 0.5$, where r_p and ρ_p are the radius and density of ring particles, respectively. The density of Chariklo is $\rho_C = 1.0\,\mathrm{g\,cm^{-3}}$. For simplicity, we consider the inner ring only, which is located at a distance of $a = 390.6\,\mathrm{km}$ from the center of Chariklo. The radial width and optical depth of the ring are 6.7 km and 0.38, respectively. We model the ring using 79557408 particles. With these parameters, it is expected that self-gravitational wakes form in the ring in a dynamical time (Toomre 1964). The simulation is performed using 1088 MPI processes and four OpenMP threads on Oakforest-PACS.

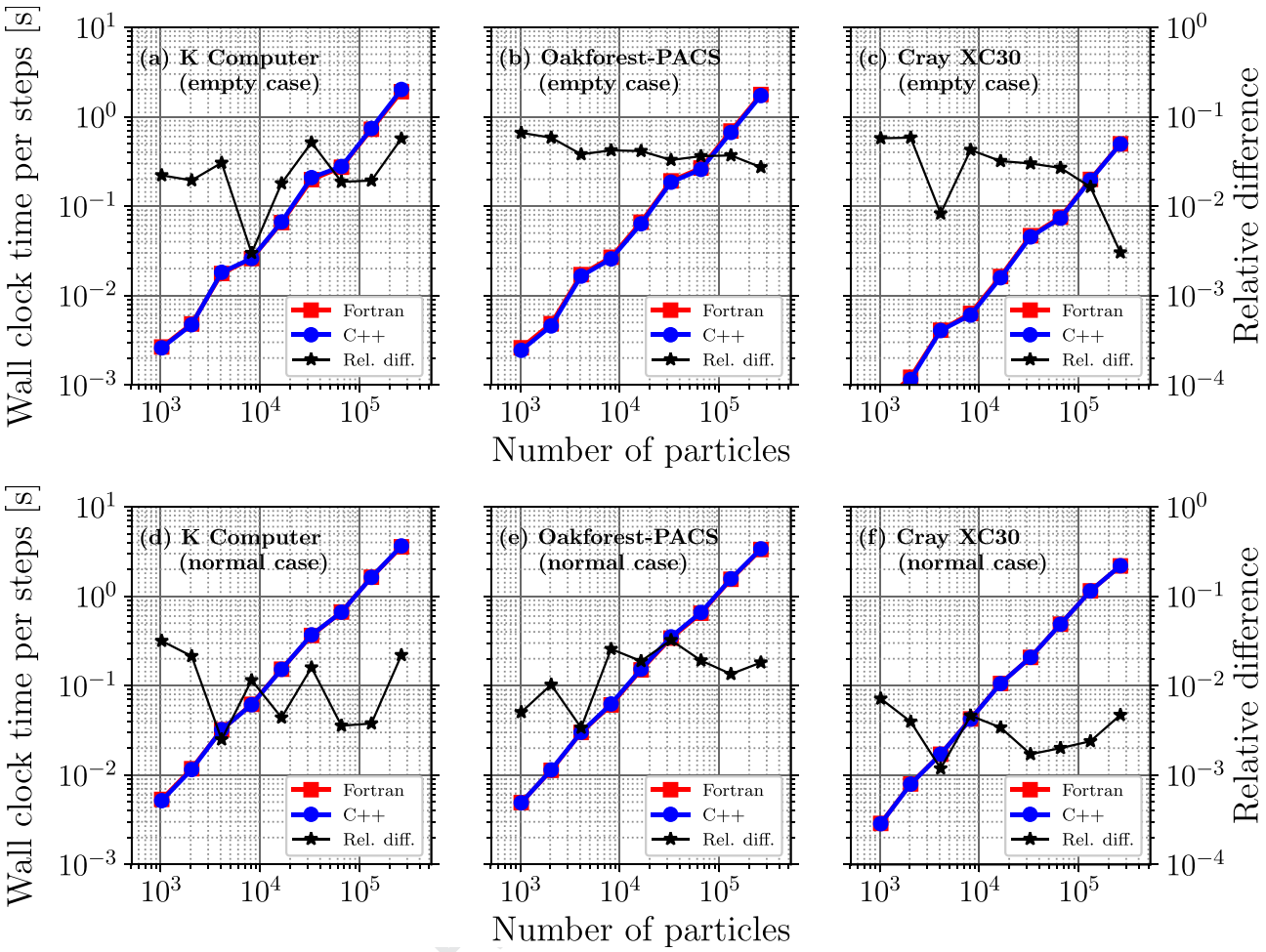


Fig. 9. Wall-clock time per steps for different numbers of particles for each computer system (upper row: the case when the interaction functions are empty, lower row: the case when the interaction functions shown in figures 7 and 8 are used; left: K-Computer, middle: Oakforest-PACS, right: Cray XC30). The relative difference of the wall-clock times between Fortran code and C++ code is also shown by stars.

Figure 10 show the distribution of ring particles at $t = 10\,t_{\rm Kep}$, where $t_{\rm Kep}$ is the orbital period at the distance of a. The particle distribution closely resembles that of Michikoshi and Kokubo (2017) as expected (see their figure 1).

Based on our experience, a great deal of time was not needed to complete this application. Thus, using Fortran interface, researchers can concentrate on their study without spending a lot of time developing simulation codes.

6 Summary

In this paper, we have presented the basic design and the implementation of the Fortran interface layer of FDPS, which is newly introduced in version 3.0. This layer provides API for Fortran and supports arbitrary types of particles and interactions as with the C++ core part of FDPS. This is realized by means of both a feature introduced in Fortran 2003 and auto-generation of interface programs. The Fortran interface also supports almost all the standard features of FDPS. Using Fortran interface, researchers can develop massively-parallel particle simulation codes in Fortran.

We also have presented the performance of an application developed by Fortran interface. By comparing it with the performance of a C++ version of the application, we have shown that the overhead cost due to Fortran interface is sufficiently small and the overall performances of both codes are very similar to each other.

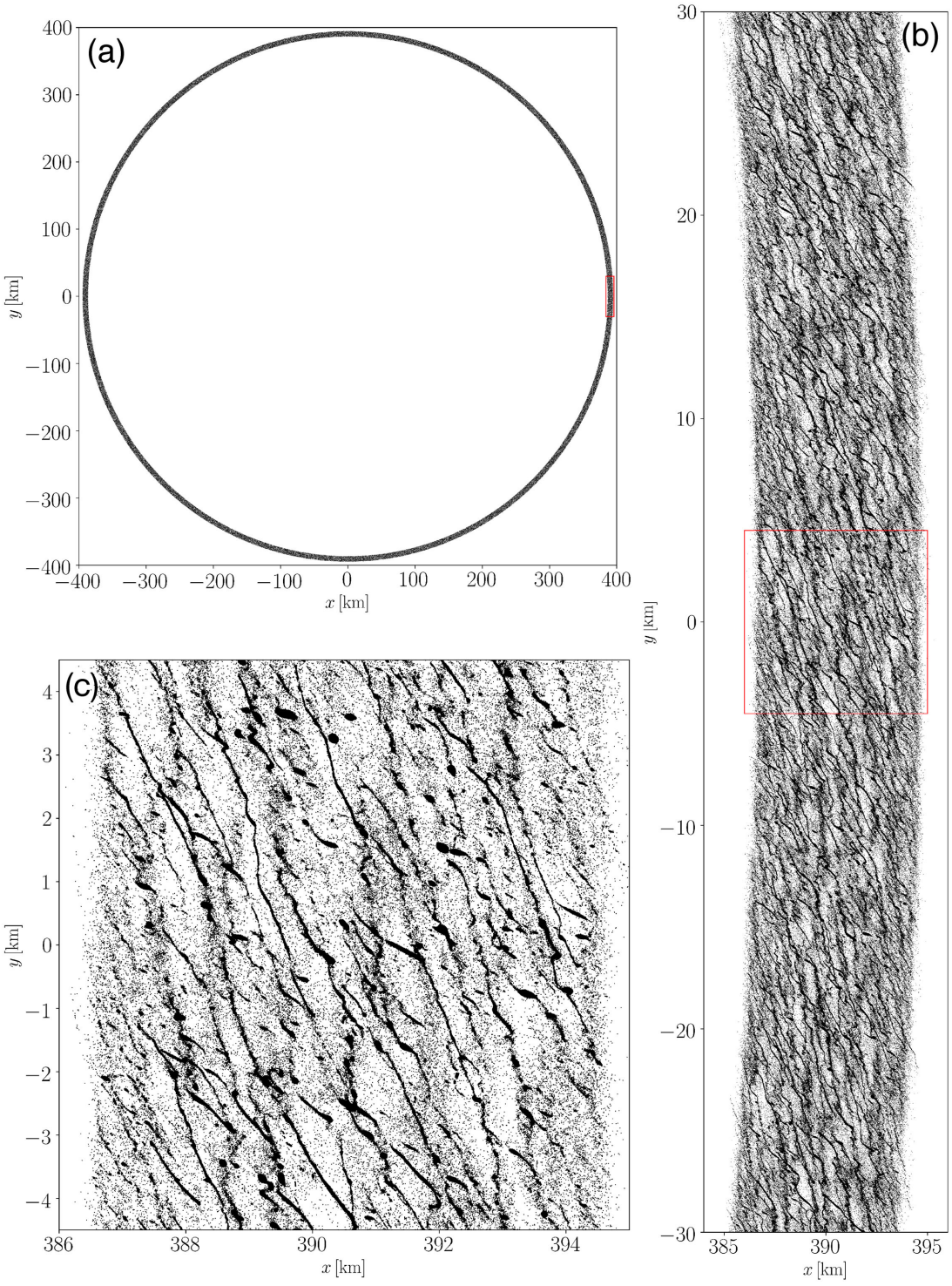


Fig. 10. Distribution of ring particles at $t = 10\ t_{\rm Kep}$ in the planetary ring simulation. Panels (b) and (c) are enlarged views of regions surrounded by the red rectangles in panels (a) and (b), respectively. In panel (a), we plot only a 10th of particles in order to keep the file size of the figure down.

Acknowledgments

We are grateful to M. Tsubouchi, Y. Wakamatsu, and Y. Yamaguchi for their 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). Part of the research covered in this paper research was funded by MEXT's program for the Development and Improvement for the Next Generation Ultra High-Speed Computer System, under its Subsidies for Operating the Specific Advanced Large Research Facilities. Numerical computations were in part carried out on Cray XC30 at Center

for Computational Astrophysics, National Astronomical Observatory of Japan, and on Oakforest-PACS at Supercomputing Division, Information Technology Center, the University of Tokyo. L.W. in this work is supported by the funding from Alexander von Humboldt Foundation.

References

Barnes, J., & Hut, P. 1986, Nature, 324, 446
Diemand, J., Angélil, R., Tanaka, K. K., & Tanaka, H. 2013, J. Chem. Phys., 139, 074309
Hosono, N., Iwasawa, M., Tanikawa, A., Nitadori, K., Muranushi, T., & Makino, J. 2017, PASJ, 69, 26
Hosono, N., Saitoh, T. R., & Makino, J. 2016a, ApJS, 224, 32
Hosono, N., Saitoh, T. R., Makino, J., Genda, H., & Ida, S. 2016b, Icarus, 271, 131
Ishiyama, T., Enoki, M., Kobayashi, M. A. R., Makiya, R., Nagashima, M., & Oogi, T. 2015, PASJ, 67, 61
Iwasawa, M., Oshino, S., Fujii, M. S., & Hori, Y. 2017, PASJ, 69, 81
Iwasawa, M., Tanikawa, A., Hosono, N., Nitadori, K., Muranushi, T., & Makino, J. 2015, in Proc. 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (New York: ACM), 1
Iwasawa, M., Tanikawa, A., Hosono, N., Nitadori, K., Muranushi, T., & Makino, J. 2016, PASJ, 68, 54
Michikoshi, S., & Kokubo, E. 2017, ApJ, 837, L13
Salo, H. 1995, Icarus, 117, 287
Tanikawa, A. 2018a, ???, in press (arXiv:1711.05451)
Tanikawa, A. 2018, MNRAS, 475, L67
Tanikawa, A., Sato, Y., Nomoto, K., Maeda, K., Nakasato, N., & Hachisu, I. 2017, ApJ, 839, 81
Toomre, A. 1964, ApJ, 139, 1217