FDPS: Framework for Developing Particle Simulator

Jun Makino
Kobe University
RIKEN Advanced Institute for Computational Science (AICS)
Talk plan

1. What we (don’t) want to do when writing particle-based simulation codes.

2. What should be done?

3. Design of FDPS

4. Current status

5. Issues on heterogeneous many-core systems

6. Performance of FDPS on TaihuLight (and PEZY-SC2)
What we want to do

- We want to try large simulations.
- Computers (or the network of computers...) are fast enough to handle hundreds of millions of particles, for many problems.
- In many fields, largest simulations still employ 1M or less particles....

(example: Canup+ 2013)
What we want to do

More precisely, what we do not want to do

- We do not want to write parallel programs using MPI.
- We do not want to modify data structure and loop structure to make use of data caches.
- We do not want to do complicated optimizations to hide interprocessor communications.
- We do not want to write tricky codes to let compilers make use of SIMD instruction sets.
- We do not want to do machine-specific optimizations or write codes using machine-specific languages (C* d*).
But what we can do?

Traditional ideas:

- Hope that parallelizing compilers will solve all problems.
- Hope that big shared memory machines will solve all problems.
- Hope that parallel languages (with some help of compilers) will solve all problems.

But...

- These hopes have never been......
- Reason: low performance. Only approaches which achieve the best performance on the most inexpensive systems have survived.
Then what can we really do?

1. Accept the reality and write MPI programs and do optimization
   Limitation: If you are an ordinary person the achieved performance will be low, and yet it will take more than infinite time to develop and debug programs. Your researcher life is likely to finish before you finish programming.

2. Let someone else do the work
   Limitation: If that someone else is an ordinary person the achieved performance will be low, and yet it will take more than infinite time and money.

- Neither is ideal
- We do need “non-ordinary people”.
Products of “non-ordinary people”
Within Astrophysics

- pkdgrav (Quinn et al. 1997)
- Gadget (Springel et al. 2001)
- GreeM (Ishiyama et al. 2009)
- REBOUND (Rein and Liu 2012)
Problems with “non-ordinary people”

- If you can secure non-ordinary people there might be some hope.
- But they are very limited resource.

If we can apply “non-ordinary people” to many different problems, it will be the solution.
How can we apply “non-ordinary people” to many different problems?

Our approach:

- Formulate an abstract description of the approach of “non-ordinary people”, and apply it to many different problem.
- “Many different” means particle-based simulations in general.
- Achieve the above by “metaprogramming”
- DRY (Don’t Repeat Yourself) principle.
To be more specific:

Particle-based simulations includes:

- Gravitational many-body simulations
- Molecular-dynamics simulations
- CFD using particle methods (SPH, MPS, MLS etc)
- Meshless methods in structure analysis etc (EFGM etc)

Almost all calculation cost is spent in the evaluation of interaction between particles and their neighbors (long-range force can be done using tree, FMM, PME etc)
To be more specific:

Particle-based simulations includes:

- Gravitational many-body simulations
- Molecular-dynamics simulations
- CFD using particle methods (SPH, MPS, MLS etc)
- Meshless methods in structure analysis etc (EFGM etc)

Almost all calculation cost is spent in the evaluation of interaction between particles and their neighbors (long-range force can be done using tree, FMM, PME etc)

However, calculation time on accelerator-based or heterogenous systems show different behaviours. I’ll come back to this issue later.
Our solution

If we can develop a program which can generate a highly optimized MPI program for

- domain decomposition (with load balance)
- particle migration
- interaction calculation (and necessary communication)

for a given particle-particle interaction, that will be the solution.
Design decisions

- API defined in C++
- Users provide
  - Particle data class
  - Function to calculate particle-particle interaction

Our program generates necessary library functions. Interaction calculation is done using parallel Barnes-Hut tree algorithm.

- Users write their program using these library functions.

Actual “generation” is done using C++ templates.
Status of the code
Iwasawa+2016 (PASJ 2016, 68, 54+arxive 1601.03138)

- Publicly available
- A single user program can be compiled to single-core, OpenMP parallel or MPI parallel programs.
- Parallel efficiency is very high
- As of version 3.0 (released 2016) GPUs can be used and user programs can be in Fortran

Tutorial
FDPS Github: https://github.com/FDPS/FDPS
Getting FDPS and run samples

> git clone git://github.com/FDPS/FDPS.git
> cd FDPS/sample/c++/nbody
> make
> ./nbody.out

To use OpenMP and/or MPI, change a few lines of Makefile
Domain decomposition

Each computing node (MPI process) takes care of one domain. Recursive Multisection (JM 2004)

Size of each domain are adjusted so that the calculation time will be balanced (Ishiyama et al. 2009, 2012)

Works reasonable well for up to 160k processes (so far the max number of processes we tried)
Sample code with FDPS

1. Particle Class

```cpp
#include <particle_simulator.hpp> //required
using namespace PS;
class Nbody{ //arbitrary name
public:
    F64 mass, eps; //arbitrary name
    F64vec pos, vel, acc; //arbitrary name
    F64vec getPos() const {return pos;} //required
    F64 getCharge() const {return mass;} //required
    void copyFromFP(const Nbody &in){ //required
        mass = in.mass;
        pos = in.pos;
        eps = in.eps;
    }
    void copyFromForce(const Nbody &out) { //required
        acc = out.acc;
    }
};
```
Particle class (2)

```c
void clear() { //required
    acc = 0.0;
}

void readAscii(FILE *fp) {//to use FDPS IO
    fscanf(fp,
           "%lf%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) { //used in user code
    vel += (0.5 * dt) * acc;
    pos += dt * vel;
}

void correct(F64 dt) { //used in user code
    vel += (0.5 * dt) * acc;
}
};
```
Interaction function

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;
    for(PS::S32 i = 0; i < n_ip; i++){
        PS::F64vec xi = ep_i[i].getPos();
        PS::F64vec ai = 0.0;
        PS::F64 poti = 0.0;
Interaction function

```cpp
for(PS::S32 j = 0; j < n_jp; j++){
    PS::F64vec rij   = xi - ep_j[j].getPos();
    PS::F64   r3_inv = rij * rij + eps2;
    PS::F64   r_inv  = 1.0/sqrt(r3_inv);
    r3_inv    = r_inv * r_inv;
    r_inv     *= ep_j[j].getCharge();
    r3_inv    *= r_inv;
    ai        -= r3_inv * rij;
    poti      -= r_inv;
}
force[i].acc += ai;
force[i].pot += poti;
```
Time integration (user code)

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);
}
Calling interaction function through FDPS

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

```c
int main(int argc, char *argv[]) {
  F32 time   = 0.0;
  const F32 tend  = 10.0;
  const F32 dtime = 1.0 / 128.0;
  // FDPS initialization
  PS::Initialize(argc, argv);
  PS::DomainInfo dinfo;
  dinfo.initialize();
  PS::ParticleSystem<Nbody> ptcl;
  ptcl.initialize();
  // pass in interaction function to FDPS
  PS::TreeForForceLong<Nbody, Nbody, Nbody>::Monopole grav;
  grav.initialize(0);
  // read snapshot
  ptcl.readParticleAscii(argv[1]);
}```
Main function

// interaction calculation
calcGravAllAndWriteBack(dinfo,
    ptcl,
    grav);

while(time < tend) {
    predict(ptcl, dtime);
    calcGravAllAndWriteBack(dinfo,
        ptcl,
        grav);

    correct(ptcl, dtime);
    time += dtime;
}
PS::Finalize();
return 0;
Remarks

- Multiple particles can be defined (such as dark matter + gas)
- User-defined interaction function should be optimized to the given architecture for the best performance (for now)
- This program runs fully parallelized with OpenMP + MPI.
Example of calculation

Giant Impact calculation (Hosono et al. 2017, PASJ 69, 26+)

Figure: 9.9M particles
Up to 2.6B particles tried on K computer
We need more machine time to finish large calculation...
Moving to PEZY systems.
Performance examples

Strong scaling with 550M particles
Measured on both K computer and Cray XC30 at NAOJ
Gravity only, isolated spiral galaxy
scales up to 100k cores
30-50% of the theoretical peak performance
Version 2.0

GPGPU and other accelerators

- FDPS Version 1.0: interaction function calculates forces from one group of particles to one group of particles (single “interaction list”)

- Version 2.0: interaction function should handle multiple interaction lists in a single call (to hide large startup overhead of GPGPUs)
Version 3.0

API to user programs written in Fortran

- "C++ is very difficult to learn/write"
- There are still many Fortran users
Fortran API

- Particle data in Fortran structured data type
- Using Fortran iso_c_binding functions, make Fortran-defined class and functions visible from C++ library functions
- Generate C++ class and member functions from Fortran source and directives
module user_defined_types
    use, intrinsic :: iso_c_binding
    use fdps_vector
    use fdps_super_particle
    !**** Full particle type
    type, public, bind(c) :: full_particle !$fdps FP,EPI,EPJ,Force
        !$fdps copyFromForce full_particle (pot,pot) (acc,acc)
        !$fdps copyFromFP full_particle (id,id) (mass,mass) (eps,eps) (pos,pos)
        !$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 !$fdps velocity
        real(kind=c_double) :: pot
        type(fdps_f64vec) :: acc
    end type full_particle
Version 4.0

- Released on Nov 8, 2017
- Mainly performance enhancement from our experience on Sunway TaihuLight
- Barnes-Hut tree in cylindrical coordinates for narrow ring calculation
- Many other improvements on scalability and/or efficiency
Performance (and tuning) of FDPS on TaihuLight (and PEZY-SC2)

• Why did we do that?
• TaihuLight and SC2 — view from the application side
• FDPS (or $N$-body simulation) on TaihuLight — New algorithms introduced
• Achieved performance
Why did we do that?

- Why not?
- TaihuLight and PEZY-SC2 have many things in common, and they are among the fastest machines. Their architecture might be tomorrow’s standard.
- We can learn how we should improve FDPS so that it will be useful on future HPC platforms.
TaihuLight and SC2 — view from the application side

Good news:

- They are fast — the fastest in the world and in Japan
TaihuLight and SC2 — view from the application side

Good news:

- They are fast — the fastest in the world and in Japan

Not-so-good news:

- Extreme performance ratio between general-purpose and “special-purpose” cores (effectively much more than a factor of 100)
- Extremely limited main memory bandwidth (BF ~ 0.0x)
- Even more limited network bandwidth
Good news:

- They are fast — the fastest in the world and in Japan

Not-so-good news:

- Extreme performance ratio between general-purpose and “special-purpose” cores (effectively much more than a factor of 100)
- Extremely limited main memory bandwidth (BF ~ 0.0x)
- Even more limited network bandwidth

Good for HPL, but...
HPL has a Number of Problems

- HPL performance of computer systems are no longer so strongly correlated to real application performance, especially for the broad set of HPC applications governed by partial differential equations.

- Designing a system for good HPL performance can actually lead to design choices that are wrong for the real application mix, or add unnecessary components or complexity to the system.
Well, then, what’s RIGHT?

Should HPC machines be designed to run “real application mix”?

What should be the goal of “hardware-software code-sign”? 
The ultimate goal

- Both the hardware and software (or, their combination) must be “optimal”
- Probably the only meaningful “optimality” in the post-Moore era is of energy efficiency (energy-for-the-solution)
Cost of DP mult and off-chip data move

- PEZY-SC2 achieve 17 Gflops/W. FMA operation itself would be around 50-60Gflops/W = 20pJ/flop
- Data move with HBM2 would be around 20 GB/s/W = 400pJ/word
- Even with HBM2, a machine with B/F=0.3 would spend equal power for computing and DRAM access (and similar or more for on-chip data move)
- Relative cost of data movement will be higher in the future
Implication on algorithm/software development

- We need to “minimize” data move
- In other words, we need to understand the theoretical lower limit of data movement necessary to solve a given problem with a given numerical scheme.
- However, at present we have no clue on it. We do not know what is the lower limit. We do not know how to get there either.

We can learn a lot by trying to use machines with low B/F and low network bandwidth
What we did on TaihuLigt

Standard Parallel Barnes-Hut tree algorithm on accelerator

- construct load-balanced domain decomposition
- move particles to new home
- construct local tree
- exchange “local essential tree”
- construct global tree
- traverse tree for a group of particles, construct an “interaction list” and let the accelerator do the actual interaction calculation. Do this for all groups
Problem with the standard algorithm

- On TaihuLight, all steps other than interaction calculation are slow
- They are extremely slow on MPE, but even when moved to CPEs, they are slow due to the limited memory bandwidth
- There are a number of other issues...
Our current implementation

- Use the “interaction list” for multiple timesteps (similar to “bookkeeping” or “neighbour list” method)
- “semi-dynamic” load balance between CPEs
- manual tuning (in assembly language) of the interaction kernels
- Elimination of all-to-all communications through the introduction of multi-process “superdomains”
- Problem-specific optimizations for planetary ring calculations
Amount of memory access and calculation

Memory access:

- Tree physical quantity update: $\sim N$
- Force calculation: $\sim 10N$
- Time integration (merged with force calculation)

around 300 bytes/particle/timestep

Force calculation:
around $3\times 10^4$ operations/particle/timestep

B/F 0.01 can be achieved. To use machines with B/F $< 0.01$, we need new ideas.
Achieved performance

Nearly 4PF on 1/10 of TaihuLight
## Calculation time breakdown

<table>
<thead>
<tr>
<th>wall clock time [s]</th>
<th># of processes</th>
<th>total</th>
<th>interaction</th>
<th>communication</th>
<th>others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-2}$</td>
<td>$10^{-1}$</td>
<td>$10^0$</td>
<td>$10^2$</td>
<td>$10^3$</td>
<td>$10^4$</td>
</tr>
</tbody>
</table>

![Graph showing calculation time breakdown](image.png)

The graph above illustrates the breakdown of calculation time across different numbers of processes. The y-axis represents the wall clock time in seconds, and the x-axis represents the number of processes. The graph includes lines for total, interaction, communication, and others, each represented by different markers and colors.
Summary

- Please visit: https://github.com/FDPS/FDPS
- A Framework for Developing parallel Particle Simulation code
- FDPS offers library functions for domain decomposition, particle exchange, interaction calculation using tree.
- Can be used to implement pure Nbody, SPH, or any particle simulations with two-body interactions.
- Uses essentially the same algorithm as used in our treecode implementation on K computer (GreeM, Ishiyama, Nitadori and JM 2012).
- Improvements for heterogenous manycore systems are ready
- Good weak scaling and performance not far from theoretical limit on TaihuLight. Hopefully also on PEZY-SC2.