# Frameworks for large-scale astrophysical simulations

**Jun Makino**
**Kobe University/ RIKEN Center for Computational Science**

# Talk plan

1. The evolution of the number of particles in astrophysical simulations
2. What should be done?
3. Design of FDPS
4. Current status
5. Who are using FDPS?
6. Summary

# The evolution of the number of particles in astrophysical simulations

| | early times | present | remark |
|---|---|---|---|
| Cosmological $N$-body | $4000$(1979) | $5.5 \times 10^{11}$(2015) | Ishiyama et al. |
| Galaxy formation | $8192$(1991) | $5 \times 10^{6}$(2017) | Auriga, DM particles |
| Star clusters | $250$(1974) | $10^{6}$(2016) | DRAGON |
| Planetary formation | $200$(1982) | $80000$(2017) | Kominami et al. |
| Giant Impact | $2000$(1986) | $6 \times 10^{6}$(2017) | Reinhardt and Stadel 2017 |

- Except for cosmological $N$-body simulations, the increase in $N$ in the last 3-4 decades is 3-4 orders of magnitudes
- $N$ for cosmological $N$-body calculation increased by eighth order of magnitude, consistent with the increase of the computer power

"$N$ gap" between cosmology and everything else.

# Why such a large gap

- Cosmological $N$-body is "easy". We need gravity and shared-timestep leapfrog integrator, nothing else. Codes for other fields are not so well optimized.
- Direct $N^2$ algorithm is still used for star clusters and planetary formation
- Star cluster simulations need to cover relaxation timescale: one more $N$.
- Tree+individual timestep for galaxy formation does not scale well (Saitoh and JM 2010).

# What do we need?

- **We need new algorithms (calculation cost per dynamical time less than $N^2$) for star clusters and planetary formation**
  - **Tree/Direct Hybrid integrator (Oshino+ 2011)**
- **For each problem (and for each machine architecture...) we need to develop a highly efficient and scalable code.**

# What do we need?

- We need new algorithms (calculation cost per dynamical time less than $N^2$) for star clusters and planetary formation
  - Tree/Direct Hybrid integrator (Oshino et al. 2011)
- For each problem (and for each machine architecture...) we need to develop a highly efficient and scalable code.

What we have learned (in the last three decades)

- Developing highly efficient and scalable code for anything more complicated than cosmological $N$-body simulation seems practically impossible

# There are so many obstacles

- **We need to write parallel programs using MPI. This is a very time-consuming and error-prone process. We also have to do a lot of work to optimize the communication and load balancing.**
- **To achieve reasonable efficiency on modern machines, we have to do lots of "tuning" including the modification of the data structure and loop structure to make better use of Cache memories.**
- **We also have to find some way to use the SIMD instruction set of modern processors.**
- **If we are to use GPGPUs, we need to modify the algorithm, data structure, and rewrite the code using Cuda or OpenCL or ...**

# Lost Hope

**Thirty years ago we hoped**

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

**But...**

- **These hopes have never been fulfilled.**
- **Reason: low performance. Only approaches which achieve the best performance on the most inexpensive systems have survived.**

# What we are doing now

1. A small group of astrophysicists (actually in most cases a single person) develops complex MPI codes (Gadget, pkd-grav, Gasoline, GreeM, AREPO, REBOUND, ....)
2. If we get large funding, or if the application is selected for petascale or exascale project, we pass our code to tuning specialists from HPC companies or big national labs.

Now we know that these approaches are not enough to develop an efficient and scalable code (except for cosmological $N$-body codes).

# Framework — a different approach

**Basic idea:**

- **Formulate an abstract description of efficient and scalable parallel particle simulation code 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)

This is of course why GRAPEs were useful when we could develop them.

# 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, Namekata+2018, Iwasawa+ 2020a, b**

- **Publicly available**
- **A single user program can be compiled to single-core, OpenMP parallel or MPI parallel programs.**
- **Parallel efficiency is very high**
- **User program can be written in standard C/Fortran (since C API is there, in principle from any other language with standard C interface).**
- **GPUs can be used**
- **High-level compiler for interaction kernel.**

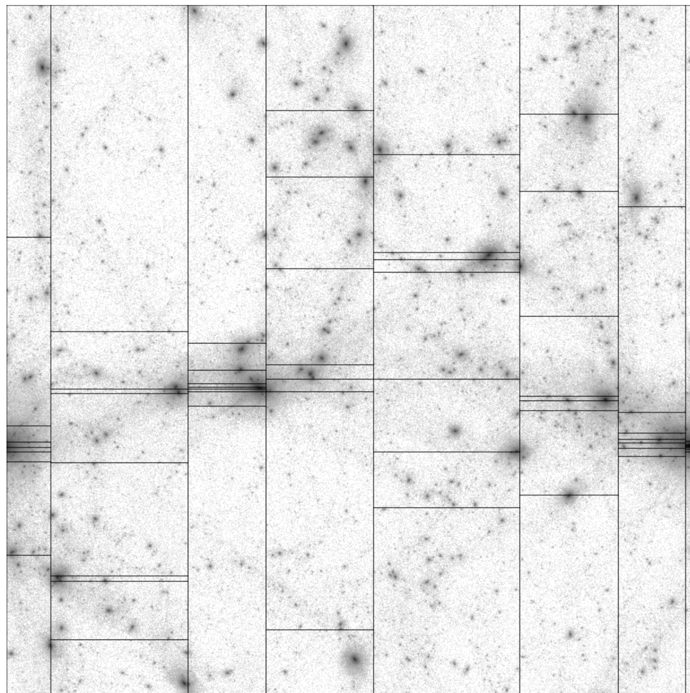
**Tutorial**
**FDPS Github: https://github.com/FDPS/FDPS**

# 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 — Particle class

```
#include <particle_simulator.hpp> //required
using namespace PS;
class Nbody{                //arbitorary name
public:
    F64    mass, eps;   //arbitorary name
    F64vec pos, vel, acc; //arbitorary 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)

```
    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

```
        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

```
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 initeraction 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)**
- **This program runs fully parallelized with OpenMP + MPI.**

# Interaction Kernel

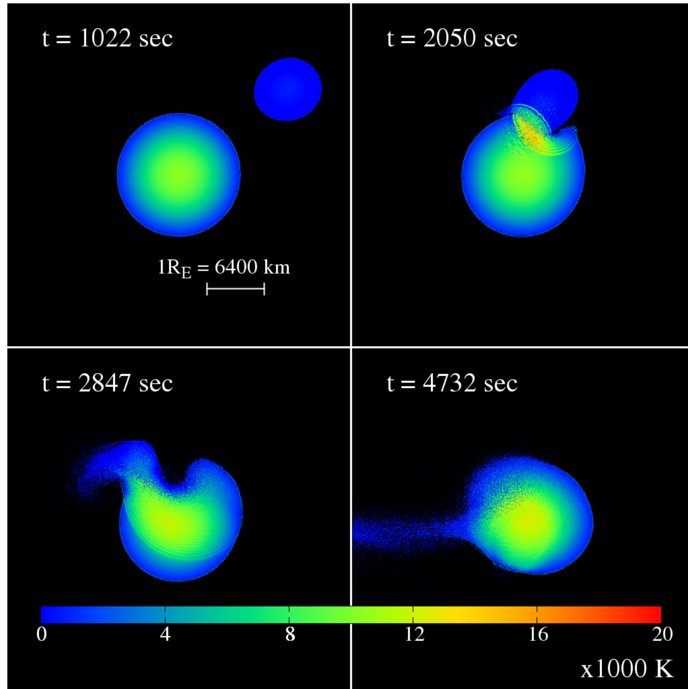
**PIKG (Particle-particle Interaction Kernel Generator) Nomura+ (in prep.)**
**From simple and high-level description like:**

```
F64 eps2
rij = EPI.pos - EPJ.pos
r2 = rij * rij + eps2
r_inv  = rsqrt(r2)
r2_inv = r_inv * r_inv
mr_inv  = EPJ.mass * r_inv
mr3_inv = r2_inv * mr_inv
FORCE.acc -= mr3_inv * rij
FORCE.pot -= mr_inv
```

**Optimized code to use SIMD units/GPGPU is generated. Currently supports:**
**AVX2, AVX512, Arm SVE (for supercomputer Fugaku), and Cuda (coming soon)**

**Users of FDPS can get full advantage of new SIMD instructions without writing machine-specific code.**
**PIKG has been used to implement SPH on Fugaku.**

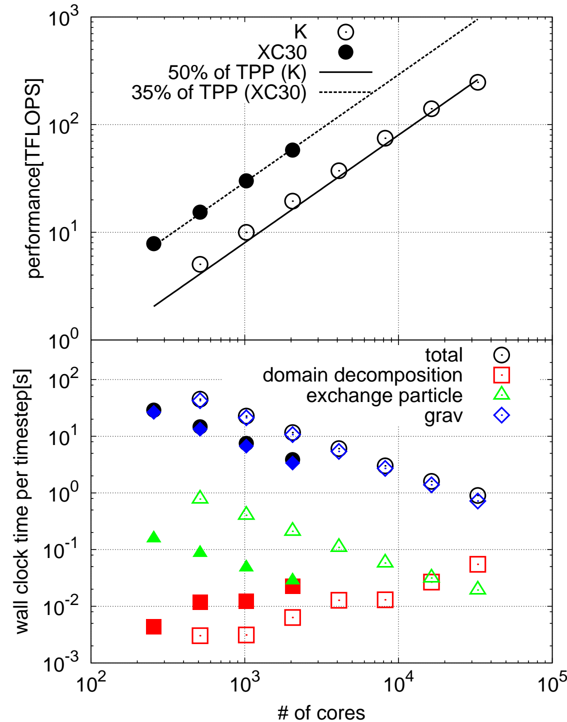
# Example of calculation




Giant Impact calculation (Hosono+ 2017, PASJ 69, 26+)
Figure: 9.9M particles
Up to 2.6B particles tried on K computer

Also used for our moon-from-magma-ocean paper (Hosono+ 2019, Nature Geoscience 12, 418)

# 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**

# New public codes developed using FDPS

- **PENTACLE https://github.com/PENTACLE-Team/PENTACLE Planetary formation code**
- **GPLUM https://github.com/YotaIshigaki/GPLUM Yet another planetary formation code**
- **PeTar https://github.com/lwang-astro/PeTar Globular Cluster**
- **FDPS_SPH https://github.com/NatsukiHosono/FDPS_SPH SPH for Giant Impact etc.**

**We are also working on a galaxy formation code with new algorithm (ASURA/FDPS)**

# Papers published using FDPS

- 60 (Google Scholar)/32 (ADS)
- Roughly half in astrophysics and half in other fields
- SPH (Tanikawa, Sugiura, Hosono, Washizu, Sugimura ...)
- Molecular Dynamics (Ayuba, ...)
- Nbody (Michikoshi, Nakajima, ...)

# Summary

- 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.
- Achieves very good scalability and efficiency on modern HPC platforms, including K, Fugaku, and GPGPUs.
- If you are considering to write your own program for some particle-based simulation, please take a look at FDPS at https://github.com/FDPS/FDPS
- If you are looking for a high-performance code, one of new codes written using FDPS may be it.
- We are also working for grid-based simulation (https://github.com/formura), but today I have no time...