Frameworks for Large-scale parallel simulations

Jun Makino
RIKEN Advanced Institute for Computational Science (AICS)

Talk Overview

- Who am I and what am I doing here?
- Framework for particle-based simulations
- Framework for grid-based simulations
- Summary

Team leader, Co-design team, Flagship 2020 project.

Official words: *The mission of the Co-design team is to organize the "co-design" between the hardware and software of the exascale system. It is unpractical to design the many-core complex processor of today without taking into account the requirement of applications. At the same time, it is also unrealistic to develop applications without taking into account the characteristics of the processors on which it will run. The term "Co-design" means we will modify both hardware and software to resolve bottlenecks and achieve best performance.*

In a bit more simpler words...

- Today's microprocessors have become very complex.
- As a result, to develop applications which run efficiently on today's processor has become almost impossible.
- To make the impossible somewhat less impossible, in the early phase of the microprocessor design, predict what problems will occur and fix them if at all possible.

Team leader, particle simulator research team.

This one I have been involved since 2012.

Simulation methods for hydrodynamics and structural analysis can be divided into grid-based and particle-based methods. In the latter case, physical values are assigned to particles, while the partial differential equation is approximated by the interactions between particles. Particle-based methods have been used to study phenomena ranging in scale from the molecular to the entire Universe. Historically, software programs for these applications have been developed independently, even though they share many attributes. We are currently developing a "universal" software application that can be applied to problems encompassing molecules to the Universe, and yet runs efficiently on highly parallel computers such as the K computer.

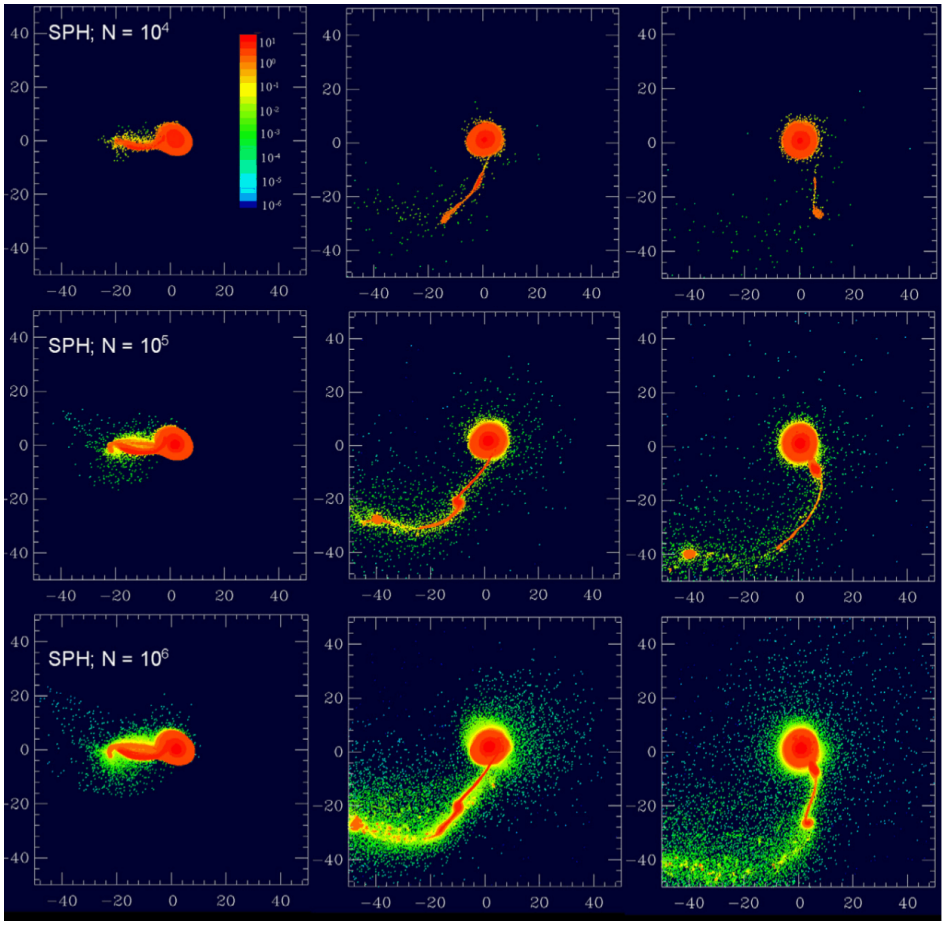
What do we actually do?

We have developed and are maintaining FDPS (Framework for Developing Particle Simulator).

1. **What we (don't) want to do when writing parallel particle-based simulation codes.**
2. **What should be done?**
3. **Design of FDPS**
4. **Current status and future plan**

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 10M or less particles....



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 (Cuda, OpenCL, ...).**

But what can we 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...

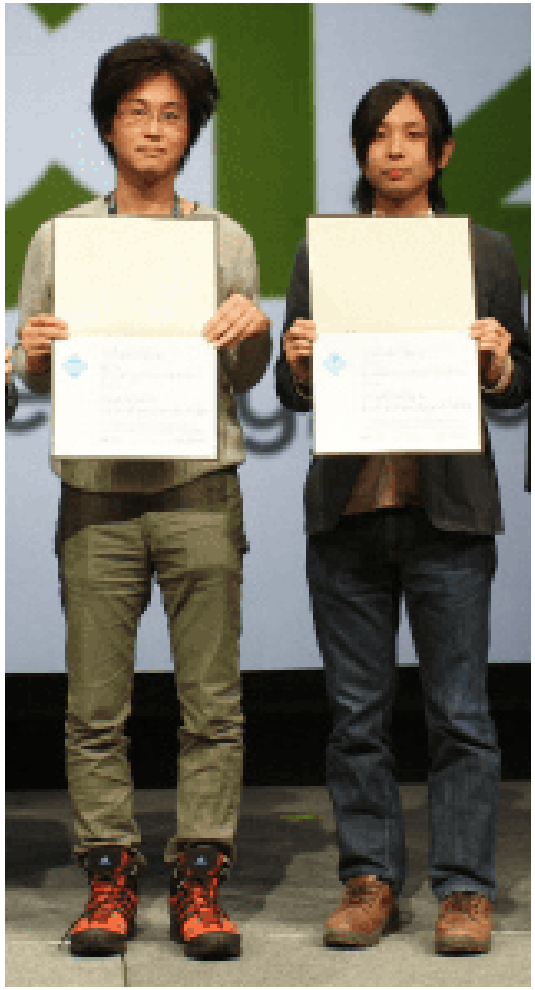
- These hopes have all been betrayed.
- Reason: low performance. Only the approach which achieve the best performance on the most inexpensive systems survives.

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. (unless your speciality is CS)
2. Let someone else do the hard 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".

Examples of "non-ordinary people"



2011/2012 Gordon Bell Prize winners

Myself excluded from the photo

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)

Our solution

If we can develop a program which generates highly optimized MPI programs to do

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

for given particle-particle interactions, 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.
- Users write their program using these library functions.

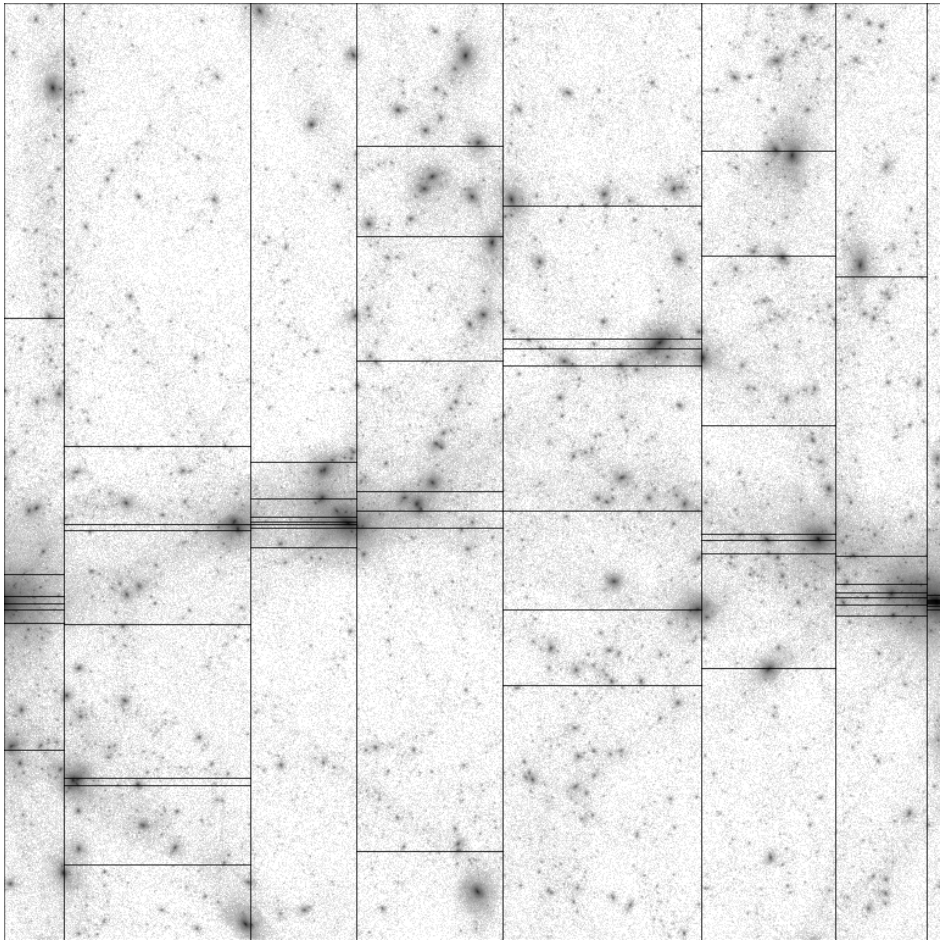
Actual "generation" is done using C++ templates.

Status of the code

- 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 2.0 (released last month) GPUs can be used.

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

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 80k nodes (so far the max number of nodes we can try)

Sample code with FDPS

1. 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 TPJ>
struct CalcGrav{
    void operator () (const Nbody * ip,
                      const S32 ni,
                      const TPJ * jp,
                      const S32 nj,
                      Nbody * force) {
        for(S32 i=0; i<ni; i++){
            F64vec xi  = ip[i].pos;
            F64    ep2 = ip[i].eps
                * ip[i].eps;
            F64vec ai = 0.0;
```

Interaction function

```
            for(S32 j=0; j<nj;j++){
                F64vec xj = jp[j].pos;
                F64vec dr = xi - xj;
                F64 mj  = jp[j].mass;
                F64 dr2 = dr * dr + ep2;
                F64 dri = 1.0 / sqrt(dr2);
                ai -= (dri * dri * dri
                       * mj) * dr;
            }
            force[i].acc += ai;
        }
    }
};
```

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
        (CalcGrav<Nbody>(),
         CalcGrav<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]);
```

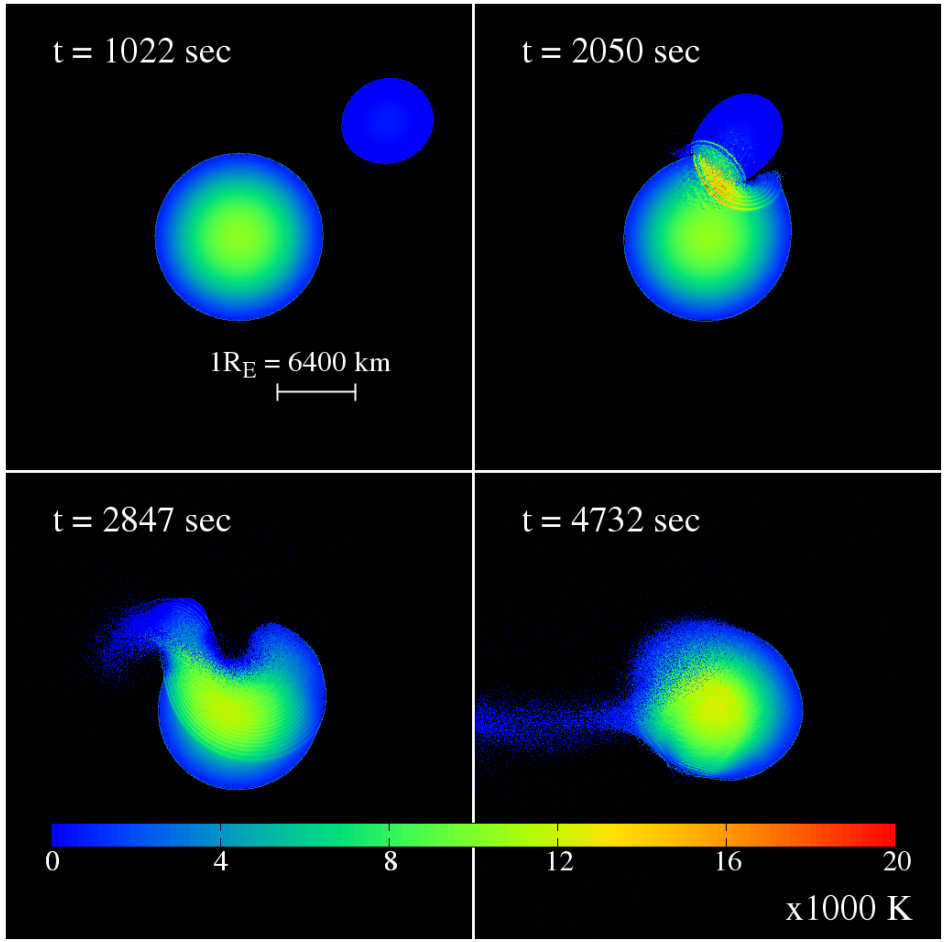
Main function

```
    // 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

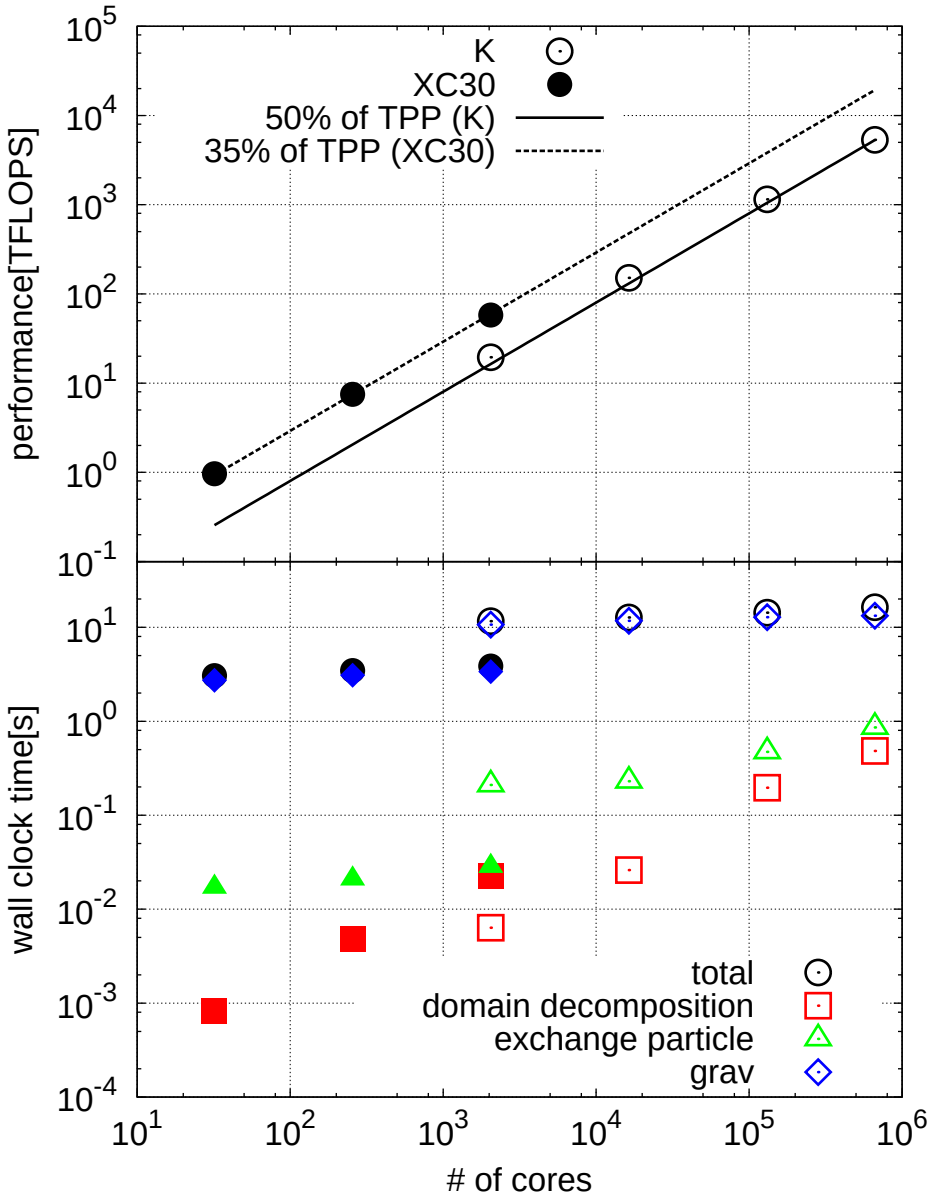
- **Multiple particles can be defined (such as dark matter + gas)**
- **User-defined interaction function should be optimized to the given architecture for the best performance (for now)**
- **This program runs fully parallelized with OpenMP + MPI.**
- **Total number of lines: 117**

Example of calculation



Giant Impact calculation (Hosono et al. in prep...)
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.

Measured performance

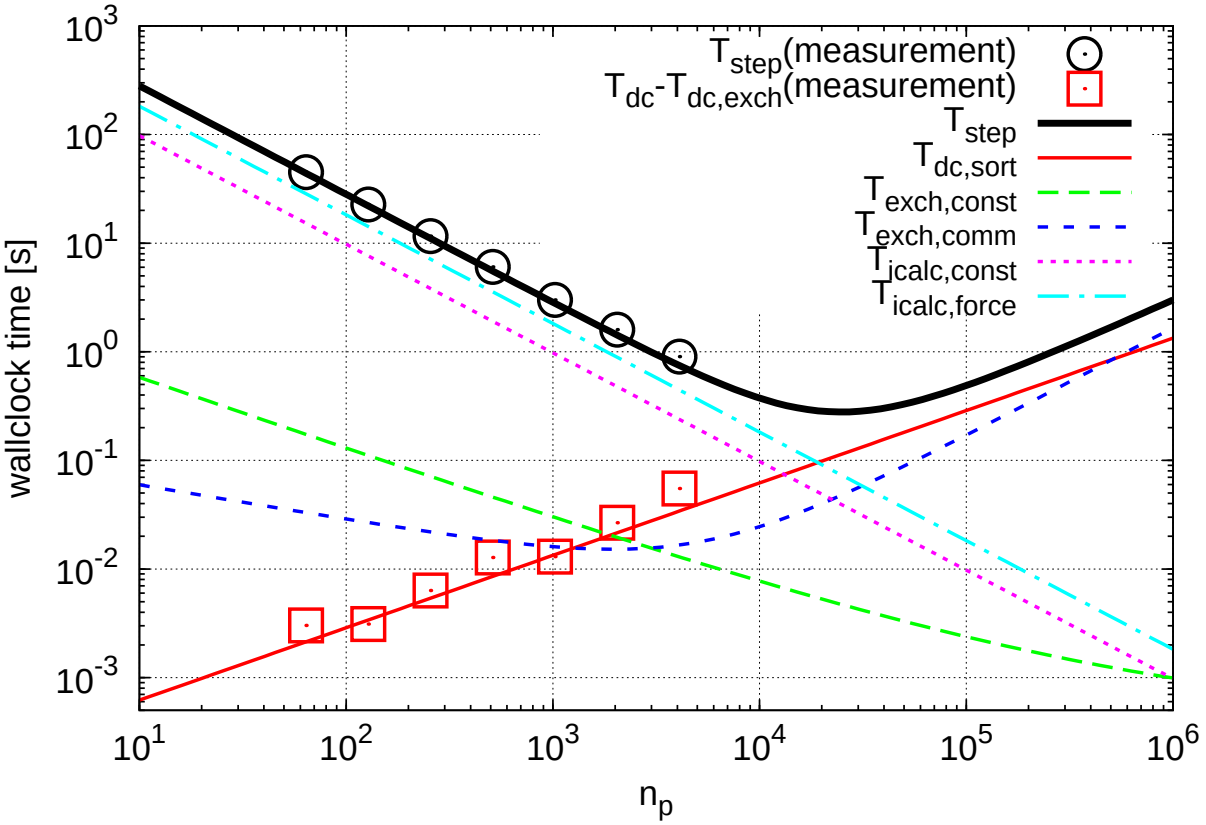


Measured performance on K computer and Cray XC30 (Haswell Xeon)

Gravitational N-body simulation

Weak scaling with 550M particles/process

Strong scaling result and model



Strong scaling result and model
550M particles on K computer

p processes:

Domain decomposition cost: $\sim O(p^{2/3})$
Communication cost: $\sim O(p^{-1/3}) + O(p)$
We are working on reducing these terms.

FDPS summary

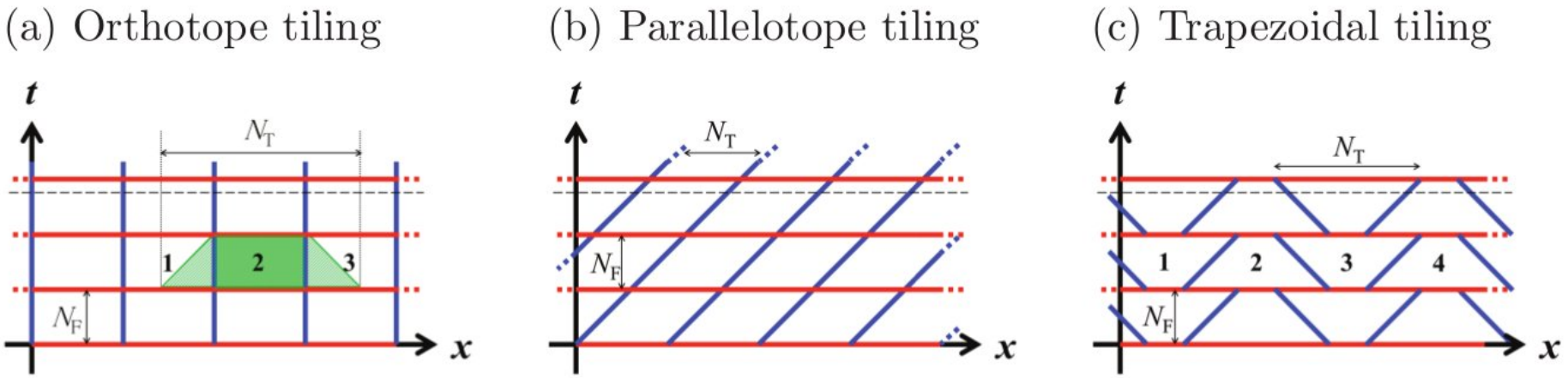
Iwasawa+2016 (arxive 1601.03138)

- **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 simulation with two-body interactions.**
- **Uses essentially the same algorithm as used in our treecode implementation on K computer (GreeM, Ishiyama, Nitadori and JM 2012).**
- **Runs efficiently on K, Xeon clusters or GPU clusters**

How about stencil calculation?

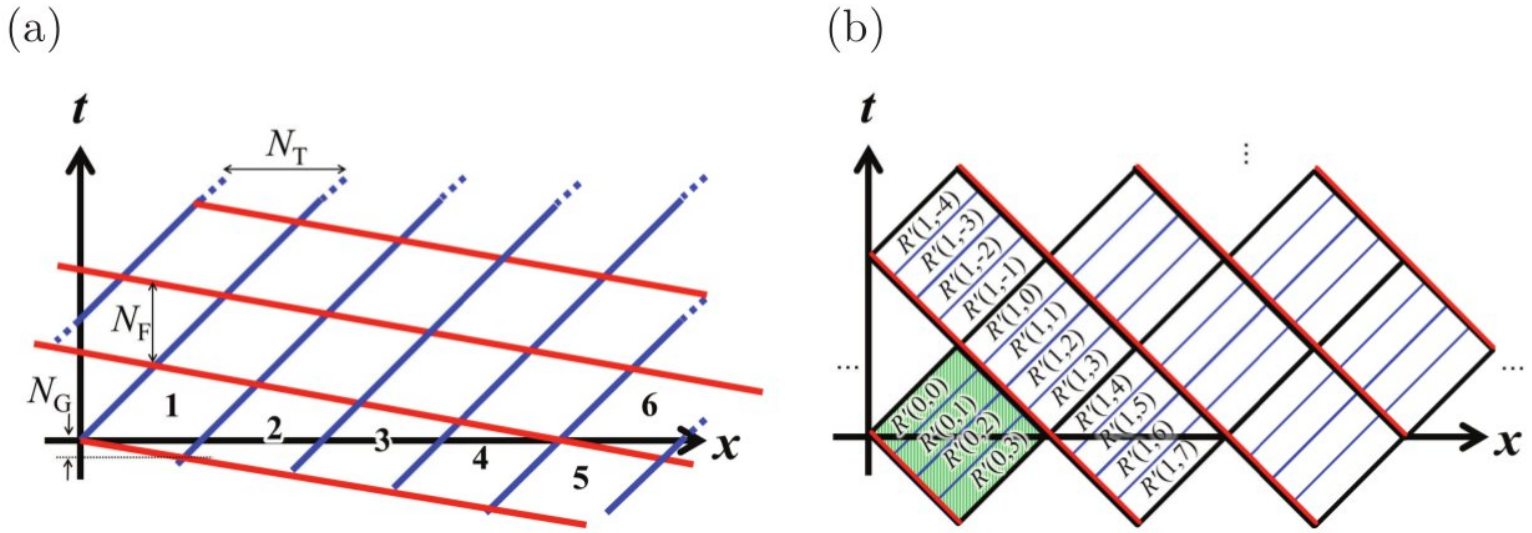
- For regular grid calculations, parallelization is not so difficult.
- Performance is low and degrading for the last three decades, due to the steady decrease of the memory bandwidth (in terms of B/F).
- Some new approach seems necessary.

Temporal Blocking



- Read in a small localized region to on-chip memory (cache), and update that region for multiple time steps.
- Can reduce the required bandwidth to the main memory.
- Algorithm which allows parallelization in 3D calculation and the theoretical limit in the bandwidth reduction were not well understood.

Optimal Temporal Blocking



Muranushi and Makino 2015

- **Formulated parallel algorithm in 3D.**
- **Obtained the theoretical limit.**

However...

(Ordinary) people cannot write a program with such a complicated algorithm. Optimization and parallelization would be even more difficult.

Let a program write programs?

- (Ordinary) people specifies the finite difference scheme and a bit more, such as the boundary and initial conditions.
- Then some fancy program generates a highly optimized and highly scalable program.

You would say: There's no such thing as a free lunch.

- Many years ago there had been... (Example: DEQSOL on Hitachi vector supercomputers)
- DEQSOL disappeared with vector machines...

Reinventing Wheels?

Formura `https://github.com/nushio3/formura`

- **We are working on this problem.**
- **Language specification is ready. The temporal blocking is not, but executables can be generated.**
- **Temporal blocking and parallelization will be available "real soon". (April-May timeframe this year)**

Example of Formura

```
dimension :: 3
axes :: x, y, z
ddx = fun(a) (a[i+1/2,j,k] - a[i-1/2,j,k])/2
ddy = fun(a) (a[i,j+1/2,k] - a[i,j-1/2,k])/2
ddz = fun(a) (a[i,j,k+1/2] - a[i,j,k-1/2])/2

∂  = (ddx,ddy,ddz)
Σ = fun (e) e(0) + e(1) + e(2)

begin function init() returns dens_init
      float [] :: dens_init = 0
end function

begin function dens_next = step(dens)
  float :: Dx, Dt
  Dx = 4.2
  Dt = 0.1
  dens_next = dens + Dt / Dx**2 * Σ fun(i) (∂  i . ∂  i) dens
end function
```

Example of Formura (2)

Previous one demonstrate how fancy you can be, using macros for finite difference operators.

I believe you can write:

```
dimension :: 3
axes :: x, y, z
begin function init() returns dens_init
      float [] :: dens_init = 0
end function
begin function dens_next = step(dens)
  float :: Dx, Dt
  Dx = 4.2
  Dt = 0.1
  dens_next=dens+Dt/(Dx*Dx)*(dens[i,j+1,k]+dens[i,j-1,k]
       +dens[i-1,j,k]+dens[i+1,j,k]
       +dens[i,j,k-1]+dens[i,j,k+1]-6*dens[i,j])
end function
```

Summary

- It has become very difficult to develop large-scale parallel programs, and to achieve high efficiency.
- We believe the frameworks can be the solution.
- The point is to separate
  - physics and numerical schemes
  - parallelization and optimization

  Framework provides the latter for a specific domain of problems.
- For particle-based simulations, we believe our FDPS will be useful for many researchers.
- For grid-based calculations, we are working on Formura.