Toward a Science of HPC software (and hardware)

Jun Makino
FS2020 Project, RIKEN-CCS
Department of Planetology, Kobe University

R-CCS Cafe, Oct 5, 2018

Talk Overview

- Brief introduction of our team: Co-design team and Particle simulator research team.
- FDPS
  - What don't want to do
  - What to do
  - Current status and future plan
- How we can make R&D in HPC "Scientific"?
- An example: "The Streamline Aeroplane"
- Another example: The Carnot Cycle
- What is "ideal" in HPC?
- 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.

It turned out that there had been really many problems...

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.

Current Member

- **JM**
- **Keigo Nitadori, Masaki Iwasawa, Yutaka Maruyama, Daisuke Namekata, Kentaro Nomura (R-CCS researchers)**
- **Yutaka Hirai (RIKEN SPD)**
- **Youhei Ishihara (Ph. Student, Kyoto U.)**

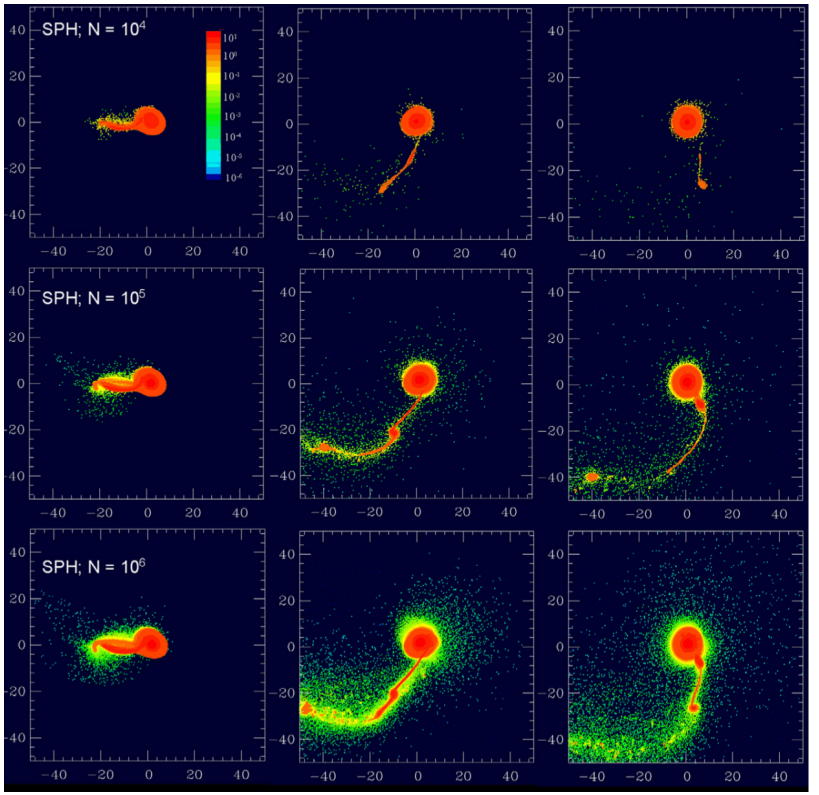
What do we actually do?

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

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 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.
- Largest simulations still employ 1M 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.**

In other words

- Modern high-end HPC systems have become too complex and too difficult to program
  - Wide SIMD instruction set
  - Many-core architecture with hierarchical cache
  - Decreasing memory bandwidth
  - Distributed memory parallel computer without global address space
  - Accelerators

But what we can do?

Traditional ideas

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

But...

- These hopes have never been realized.
- Reason: low performance. Only the approach which achieves 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. (Also, your target machine will disappear before...)
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".

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 could be part of 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 (FDPS), or regular-grid calculation (Formura).**
- **Achieve the above by "metaprogramming"**

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

Therefore, if we can develop a program which generates a highly optimized MPI program to do

- **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.
- **Users write their program using these library functions.**

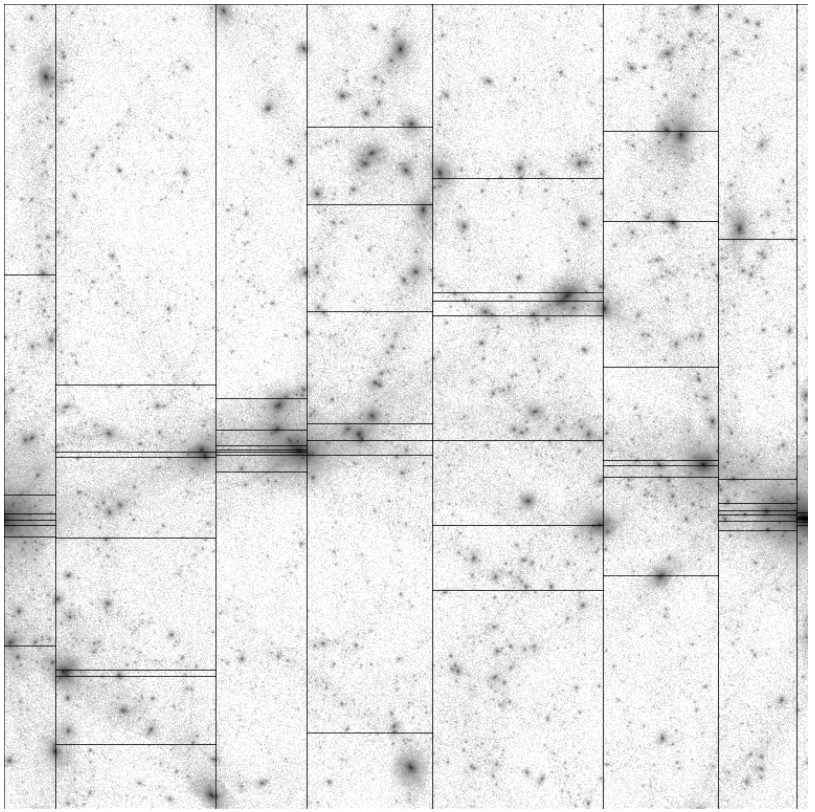
Actual "generation" is done using C++ templates.

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 4.0, the users can use GPUs, and can write their program in Fortran. (Other languages with version 5.0)

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 reasonably well for up to 160k nodes (so far the max number of processes we could 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 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
        (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]);
```

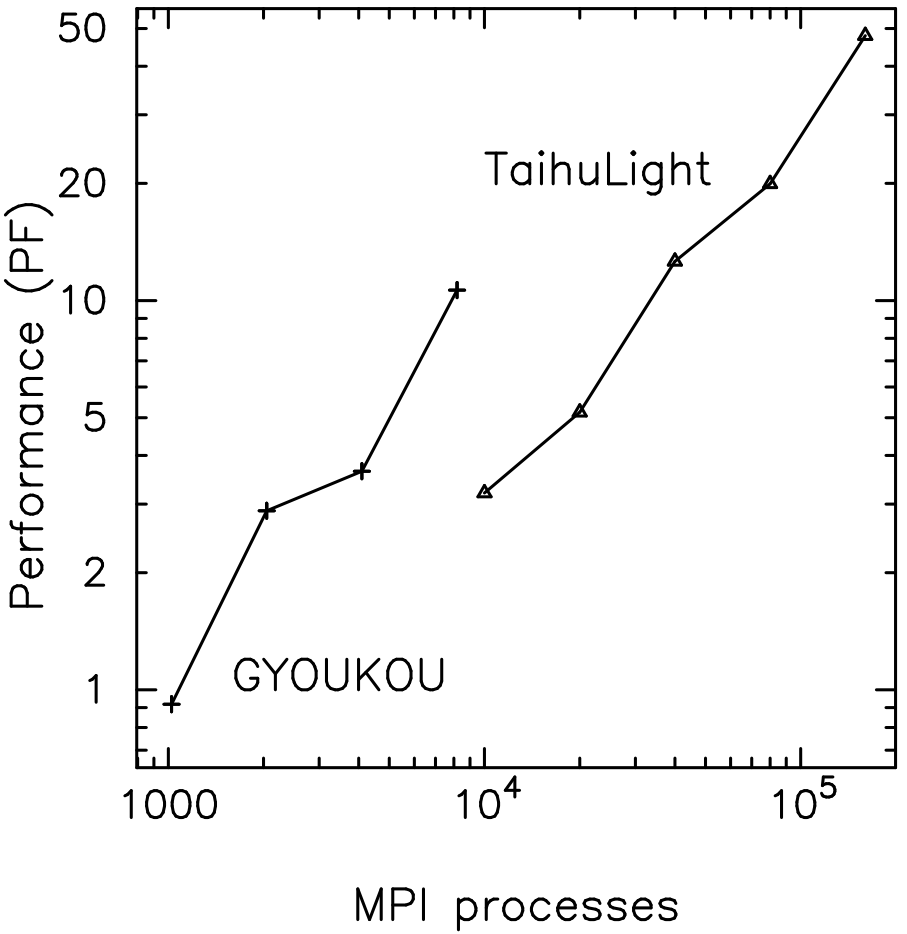
Main function

```
    // 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

- User-defined particle class should have several "required" member functions
- Multiple particles can be defined (such as dark matter + gas)
- User-defined interaction function should be optimized to a given architecture for the best performance (for now)
- This program runs fully parallelized with OpenMP + MPI(taken care within FDPS)
- SIMD should be taken care in interaction function. Accelerator should be too.

Performance example



- **Weak scaling with 10M particles/process**
- **Simulation of planetary rings**
- **Optimized version for PEZY-SC2 and Sunway TaihuLight**
- **40% and 23.5% of the theoretical peak performance.**

Users of FDPS

- So far, $\sim$ 20 scientific papers have been published
- Astrophysics, Planetary Science, Material Science, and more
- There are many more users in various fields

FDPS summary

Iwasawa+2016 (Publ. Astron. Soc. J. 2016, 68, 54/arXive 1601.03138, https://github.com/FDPS/FDPS)

- **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.**
- **Use 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**

What we learned from FDPS development

- Frameworks like FDPS can be used to make good use of a wide variety of processor architectures
- "easy to use" does not necessarily mean "easy to achieve high efficiency"
- "High efficiency" does not necessarily mean high energy efficiency

The goal of HPC R&D: Provide tools to solve scientific problems "efficiently"

Question:

What is the meaning of "efficiency"? Is there any scientific definition?

How we can make the R&D of HPC software and hardware "Scientific"?

What do I mean by scientific?

- Our approach for HPC application development is rather "problem-driven". Try an existing code on a new architecture, see what happens, and fix the problems,
- Our approach for HPC architecture is, well, "evolutionary" at best.
- Scientific approach should be driven by the first principle, whatever it is.

Let's look at examples in other fields.

Disciplines based on scientific approach

- **An example: "The Streamline Aeroplane"**
- **Another example: The Carnot Cycle**
- **The meaning of "Scientific" approach for HPC**

The Streamline Aeroplane

B. Melville Jones, The Streamline Aeroplane, Journal of the Royal Aeronautical Society, 33(1929)

THE STREAMLINE AEROPLANE

BY B. MELVILL JONES, A.F.C., M.A., F.R.A$_E$.S.

Ever since I first began to study Aeronautics I have been annoyed by the vast gap which has existed between the power actually expended on mechanical flight and the power ultimately necessary for flight in a correctly shaped aeroplane. Every year, during my summer holiday, this annoyance is aggravated by contemplating the effortless flight of the sea birds and the correlated phenomenon of the beauty and grace of their forms.

We all possess a more or less clear ideal of what an aeroplane should look like; a kind of albatross with one or two pairs of wings—depending on whether we live in Germany or Britain. In our more sanguine moments we even—like Alice and the cat—see the wings without the albatross. But progress towards this ideal, so far as the general purposes craft is concerned is, we must all admit, painfully slow. It has seemed to me that a contributory factor to the slowness of this evolution has been the lack of any generally understood and easily visualized estimate of what could be achieved were the difficulties in the way of realizing the ideal form overcome.

Albatross

Sopwith Camel

(UK WWI fighter aircraft)

They look different

Albatross is clean, and Sopwith Camel is, well, not.

How we can quantify the difference?

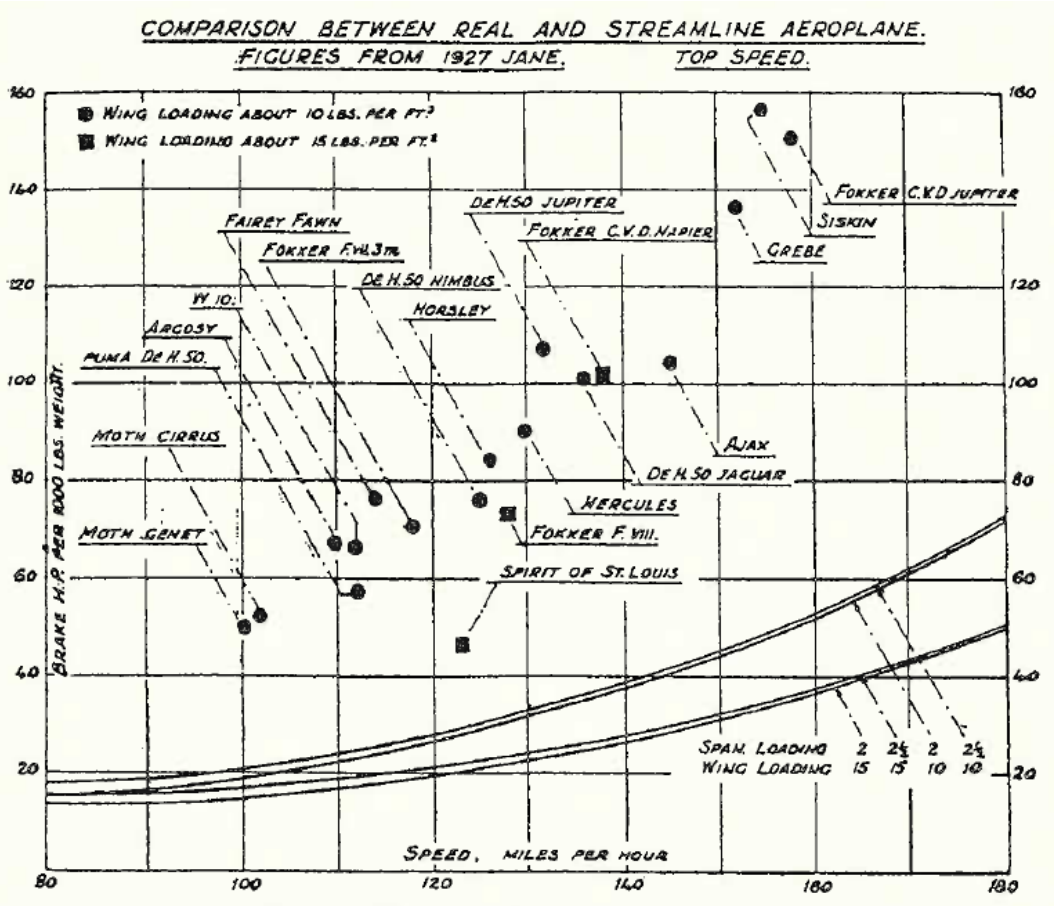
"Looks clean" is not quite enough for science.

Question here: How much we can reduce the aerodynamic drag?

Answer (from fluid dynamics):
There are terms that can be reduced and terms that cannot.

$$\text{Drag} \begin{cases} \text{induced} \quad \text{drag} - \text{remain finite (Aspect ratio)} \\ \text{parasite drag} \begin{cases} \text{Pressure drag} - \text{can be reduced down to zero} \\ \text{Frictional drag} - \text{limit due to total area} \end{cases} \end{cases}$$

Result of measurements



horizontal axis: velocity
vertical: power normalized by weight
Solid curves: theoretical limits
(different curves with span- and area-load)
Points: real aircrafts. Best one: Spirit of St.Louis

Split of St. Louis

Still more than three times the limit
Further possible improvements include:
engine cowl, retractable gear, cantilever wing

Modern aircrafts

A glider

Boeing 787

Even the B787 looks quite smart and close to ideal one.

Another example: The Carnot Cycle

Question here: How much "usable" work we can extract from a heat source

(Final theoretical) Answer: The first and second laws of thermodynamics.

Actual efficiency cannot exceed that of the Carnot Cycle:

$$\eta_c = \frac{T_h - T_l}{T_h}$$

where: T_h temperature of high-temperature source
T_l temperature of low-temperature source(ambient)

Some examples

	T_h(C)	η_c	η
Modern Natural Gas	**1500**	**0.83**	**0.60**
Nuclear (LWR)	**330**	**0.52**	**0.33**

One need to go to high temperature to achieve high efficiency

What is the meaning of Scientific approach to HPC R&D?

- In the above two examples of aircrafts and heat engines, the goal is the energy efficiency.
- For HPC R&D, the ultimate goal should also be the energy efficiency too, because:

  for modern HPC systems, the cost of electricity is becoming higher than the hardware cost. Thus, energy efficiency directly determines the available computing power.

For a given calculation, there must be the lower limit for the required energy, and the best computer is defined as a machine which achieves that minimum required energy.

Can we define the "lower limit"?

Possible objections include:

1. **Lower limit depends on the semiconductor technology.**
2. **Even if we assume that there is a lower limit for a given application, each application requires specific architecture to realize its lower limit. It is clearly impossible to build a machine for each application, and thus such lower limit is practically useless.**
3. **Since the algorithms used for applications will change, the lower limit will also change, and we cannot define the lower limit as a long-term target.**

We'll discuss each point.

Point 1

Lower limit depends on the semiconductor technology.

- **Well, in the post-Moore era, the semiconductor technology doesn't evolve as fast as it did in 20C.**
- **Therefore, now it is meaningful to ask: What is the minimum energy consumption for a given semiconductor technology?**
- **We should be able to give a simple and fundamental answer as in the case of aircrafts and heat engines, and such an answer should be the basis for the scientific theory for HPC**

Point 2

Even if we assume that there is a lower limit for a given application, each application requires specific architecture to realize its lower limit. It is clearly impossible to build a machine for each application, and thus such lower limit is practically useless.

- **This was certainly an meaningful argument in 20C. General-purpose machines built with the latest technology outperformed application-specific ones in a few years.**
- **Now in the post-Moore era, this will no longer happen.**
- **On the other hand, it becomes prohibitingly expensive to make an ASIC for a specific application. We need something else.**

Point 3

Since the algorithms used for applications will change, the lower limit will also change, and we cannot define the lower limit as the long-term target.

- **Computational Science has now the history of 70 years, and basic algorithms for various problems has now become sort of stable.**
- **There will be many changes in details, but the basic concepts like regular grid, irregular grid, particles and graphs will remain unchanged.**
- **Many new methods for parallelization are now being developed, but they are mostly solutions for the problem that hardware is becoming more complex, and does not lead to the reduction of operation count.**

Classification of power consumption

Aircrafts:

- Drag
  - induced drag – remain finite (Aspect ratio)
  - parasite drag
    - Pressure drag – can be reduced down to zero
    - Frictional drag – limit due to total area

Computers (for HPC)

- Energy consumption
  - Combinatorial Logic for Arithmetic operation
    - Dynamic
    - Static(leak)
  - Storage(Memory, Register)
  - Data movement(Clock, Latch, Wires)
  - Control logic(instruction decode etc)

Dynamic power for arithmetic operations cannot be eliminated. Everything else can be.

How far are we from energy-minimum computing?

Example: K computer — roughly 1GF/W with 100% efficiency.

- **Optimized grid CFD calculation would achieve 15%: 0.15GF/W.**
- **The energy consumption of an FP64 unit in 40/45nm technology would be 25GF/W or around. Thus, our energy efficiency is around 0.6%.**
- **Even for an application which achieves 50% "efficiency" on K, the energy efficiency is still around 2%.**
- **Post-K will be around 3 times more energy efficient (purely by design)**
- **X86 processors are/will be much worse.**

Possible criticisms

- Data movement is essential for computation and its cost should not be ignored.
- Universality is more important
- This is clearly an extreme argument with little practical meaning.
- Even when we specify an application, we cannot make "others" zero.

In the following, we'll discuss the last one.

Minimum-energy computers for specific applications

Let's consider

1. Regular grid (neighbor communication only, explicit stepping)
2. Particles
3. Dense Matrix
4. Irregular grid

Regular grid

- For explicit timestepping, we can construct a specialized pipeline for arithmetic operations, which would minimize the main memory access.
- Modern high-order, high-accuracy schemes require very large numbers of operations per step per grid point. Thus, memory access cost can be made small.
- (Not that we can achieve this on existing or coming machines)

Particles

- **Operations per particle per step is huge, of the order of 10^4 or more.**
- **Specialized pipeline for particle-particle interaction is always possible.**
- **Thus memory access cost can be made negligible.**

Dense Matrix

- Most operations can be transformed to matrix-matrix multiplications
- By blocking, memory access of matrix-matrix multiplications can be minimized.

Irregular grid

- This is problematic
- Classical CG requires large amount of memory access.
- Multigrid is even more problematic
- On the other hand, in some of modern parallel methods, locally dense matrices are used.
- In my opinion, we should develop stable and accurate explicit schemes for irregular grids

Minimum-energy computers for specific applications

- Seems possible except for irregular grids.
- iterative methods on irregular grids are and will be problematic on large-scale parallel machines. We probably will need something else.

Thus, we can measure the difference between the theoretical limit and real machines, by measuring the power consumption of arithmetic units and total power consumption.

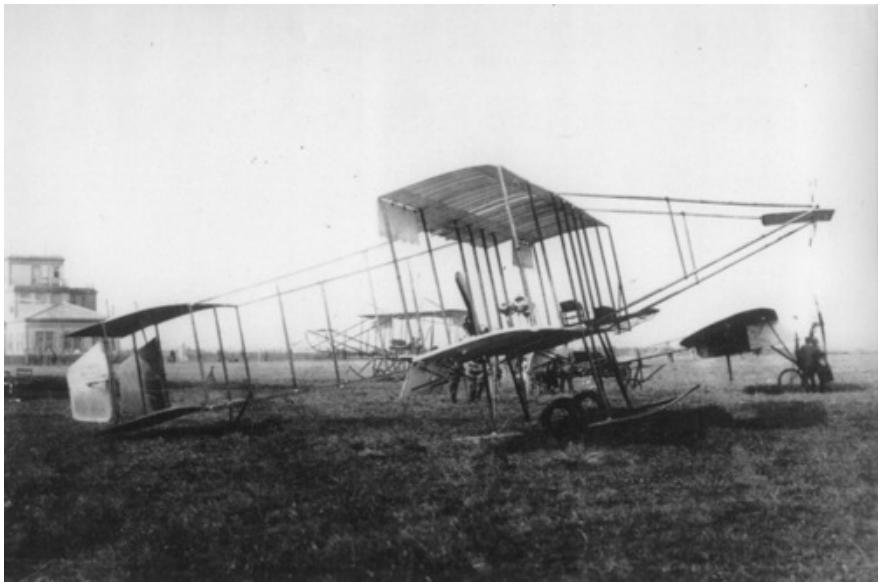
Minimum-energy General-Purpose computer

- We can define the minimum-energy computer for each applications
- For general-Purpose computers, one can simply measure the difference with the theoretical limits for several "typical" applications and take whatever mean one like.
- For a given set of application workload, there must be an optimal architecture. In other words, this is a mathematically well-posed problem.
- So "general purpose" might difficult, but "Multi-purpose" is certainly possible.

Summary

- Concepts like "Streamline Aeroplane" and "Carnot Cycle" played extremely important roles as guiding principles.
- They are important because they define the theoretical limit in what we can do.
- There is no such clear guiding principle which defines the theoretical limit in computer architecture.
- In this talk, I tried to define such theoretical limit, for numerical calculations.
- It is defined as the power consumption of combinatorial arithmetic logic.
- Current computers are far from the limit, typically by a factor of 100 or more.

Computers now and Ideal computer



Computers now

Ideal computer

Background

Where will we be in 2028?

- 2nm technology?
- Should achieve 40-60 times better power efficiency compared to TSMC 40LP
- 1-1.5TF/W
- Even when we consider loss in DC/DC conversion, 500GF/W (DP) should be possible.
- With the efficiency less than 10%, we will end up with less than 100GF/W.

Five eras of evolution of CPUs

I —1969: Before CDC7600
(before fully pipelined multiplier)

II —1989: Before Intel i860 (single-chip CPU with (almost) fully pipelined multiplier)

III —2003:CMOS scaling era (Power $\propto$ size^3)
From i860 to Pentium 4

IV —2022(?):Post-CMOS scaling era (Power $\propto$ size)
From Athlon 64 X2 to Knights Hill(?)

V 2022(?)—: Post-Moore era(miniaturization stops)
???

Evolution of Big Irons

I —1969: Before CDC7600
(before fully pipelined multiplier)

II —1982: Before Cray XMP
(before two multipliers)

III —1992: From Cray XMP to Cray T-90, or Before Fujitsu VPP500 (shared memory vector machines)

IV —2002? From VPP500 to VPP5000
(distributed memory vector machines)

V — now? From Earth Simulator (???)

Evolution of microprocessors

I —1989: Before Intel i860
(before fully pipelined multiplier)

II —2003: From i860 to Pentium 4
(deep pipeline, single core)

III —201X? : From Athlon64 X2 to Xeon Phi
(multicores with SIMD units)

IV — ???

Big Irons and microprocessors

Era		Big Irons	microprocessors
I	multi-cycle MULT	- 1969	- 1989
II	one MULT	- 1982	- 2003
III	multicore	- 1992	- 201X?
IV	distributed	- 2002	???

Observations:

- Microprocessors follows the evolutionary track of big irons with 20-year delay.
- For microprocessors, transition from shared memory to distributed memory should have happened in 2012. It did not. (Even GPGPUs have physically shared memory)

Why not distributed memory microprocessors?

Well, what do I mean by "distributed memory microprocessors"? Can there be anything like that?

- If we take the similarity with the Big Irons, it means "VPP500 in a chip"
- This, however, does not make sense, since giving up the cache coherency does not increase the off-chip memory bandwidth.
- With VPP500, by putting one processor to one board, local memory bandwidth is increased.
- With many-core microprocessors, it is not clear how we can increase the local memory bandwidth per core.

Where are we now?

- **Past trend: multi- and many-core processors with wide SIMD FPUs.**
- **Hierarchical cache memory**
- **relatively slow and high-latency interconnection**

Question: How one can develop efficient HPC applications?