粒子系シミュレータ開発フレームワークFDPSと粒子間相互作用カーネルジェネレータPIKGの紹介

野村 昂太郎 / Kentaro Nomura
神戸大学理学研究科惑星科学研究センター

背景

- 一般の研究者にとって，現代の多様なスパコン上でスケーラブルに動作するプログラムを書くのは非常に大変/困難

- シリアルコードをかけば並列化してくれるフレームワークが必要

- 分子動力学やN体，SPHなどの粒子系シミュレータのコードの性能は(多くの場合)粒子間相互作用計算の実装(最適化)に大きく依存

- ユーザはそれぞれの計算機向けに最適化された相互作用カーネルを書かないといけないができあがるものは(大抵の場合)再利用できない

- 単一のコードからさまざまな計算機上で性能を引き出したい！

唐突ですがここで富岳概要

https://news.livedoor.com/article/detail/18423524/

総ノード数

総ノード数	158,976ノード 384ノードx 396 ラック= 152,064 192 ノードx 36 ラック= 6,912

総理論性能

総演算性能	通常モード（CPU動作クロック周波数2GHz）	• 倍精度理論最高値（64bit）488ペタフロップス • 単精度理論最高値（32bit）977ペタフロップス • 半精度（AI学習）理論最高値（16bit）1.95エクサフロップス • 整数（AI推論）理論最高値（8bit）3.90 エクサオップス
	ブーストモード（CPU動作クロック周波数2.2GHz）	• 倍精度理論最高値（64bit）537ペタフロップス • 単精度理論最高値（32bit）1.07エクサフロップス • 半精度（AI学習）理論最高値（16bit）2.15エクサフロップス • 整数（AI推論）理論最高値（8bit）4.30 エクサオップス
総メモリ容量		4.85 PiB
総メモリバンド幅		163 PB/s

ノード単体性能

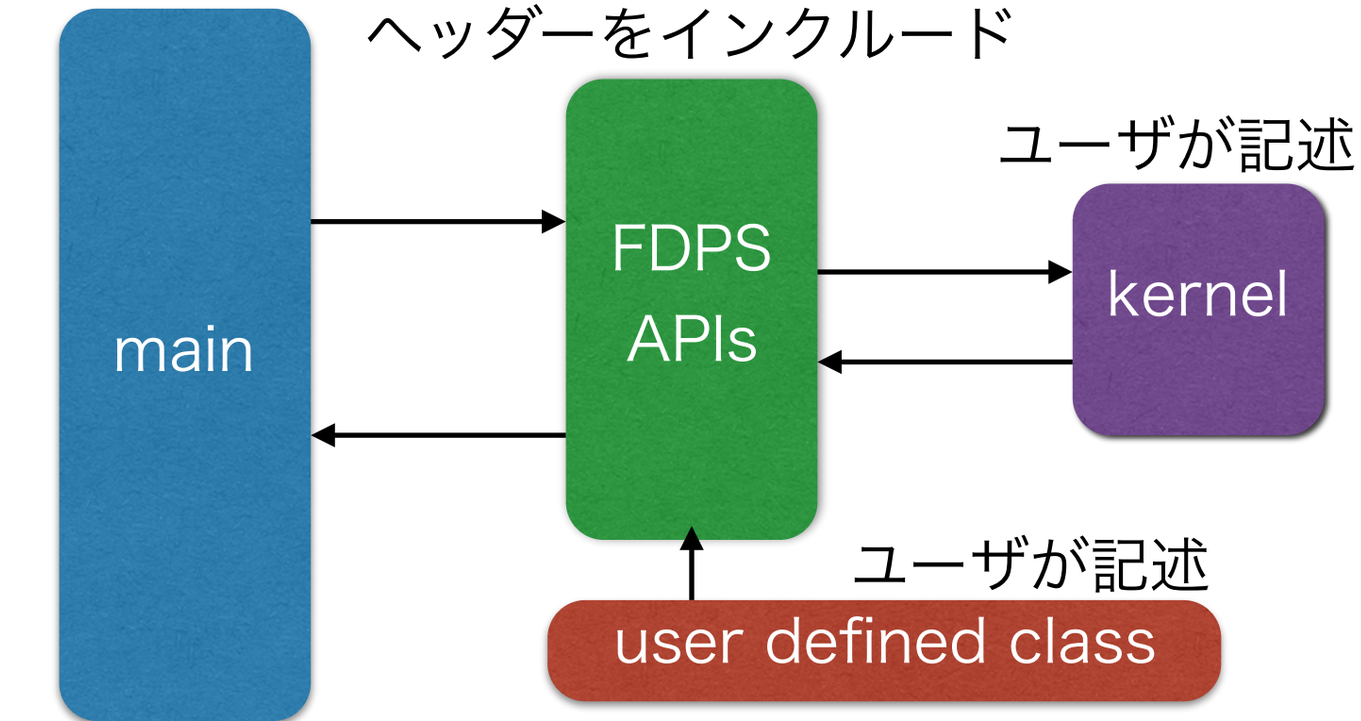
命令セットアーキテクチャ		Armv8.2-A SVE 512bit 富士通拡張:ハードウェアバリア、セクタキャッシュ、プリフェッチ
計算コア数		48 + 2アシスタントコア 4 CMG (Core Memory Group, NUMA nodeのこと)
演算性能	通常モード（CPU動作クロック周波数2GHz）	倍精度: 3.072 TF, 単精度: 6.144 TF, 半精度: 12.288 TF
	ブーストモード（CPU動作クロック周波数2.2GHz）	倍精度: 3.3792 TF, 単精度: 6.7584 TF, 半精度: 13.5168 TF
キャッシュ*1 *2		L1D/core: 64 KiB, 4way, 256 GB/s (load), 128 GB/s (store)
		L2/CMG: 8 MiB, 16way L2/node: 4 TB/s (load), 2 TB/s (store) L2/core: 128 GB/s (load), 64 GB/s (store)
メモリ		HBM2 32 GiB, 1024 GB/s
インターコネクト		Tofu Interconnect D (28 Gbps x 2 lane x 10 port)
I/O		PCIe Gen3 x16
テクノロジー		7nm FinFET

https://www.r-ccs.riken.jp/jp/post-k/overview.html

FDPS (Framework for Developing Particle Simulators)

- 粒子同士が相互作用しながら時間発展するシミュレーションの開発支援フレームワーク
- FDPSはノード間通信・系の分割・相互作用リスト作成など面倒なことを引き受けてくれる
- ユーザはユーザー定義クラス, 時間発展(main関数)と相互作用カーネルを記述
- 粗密にあわせて動的に領域を分割(右図)
- ツリー構造を使った相互作用リストの作成
- Fortran・Cインターフェース/アクセラレータの利用

Iwasawa+, PASJ(2016)



FDPSの計算の流れ

1．領域分割

ロードバランスを考慮しながら計算領域をサブ領域に分割して各MPIプロセスに粒子を割り当て

2．粒子情報の交換

粒子の移動などにあわせてそれぞれのプロセスが持つ粒子の情報を交換

3．相互作用情報の交換

相互作用に必要な情報を隣接するプロセスから取得

並列化
ノード間通信

APIを提供

4．相互作用の計算

各MPIプロセス内で相互作用を計算

相互作用リストを提供し
計算を高速に実行

5．粒子の時間発展

それぞれのプロセスが担当するの相互作用を計算

ユーザーが自由に記述

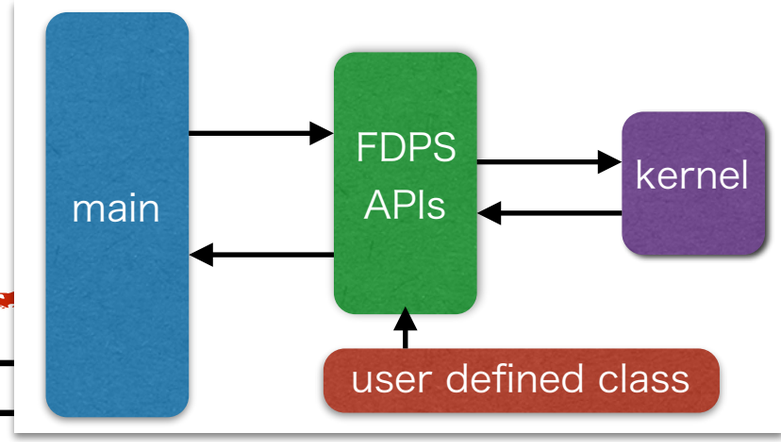
FDPSを用いたコード開発

$$U_i = \sum_{j=0}^{N} \left\{ \left(\frac{1}{|\boldsymbol{r}_{ij}|} \right)^{12} - \left(\frac{1}{|\boldsymbol{r}_{ij}|} \right)^{6} \right\}$$

それではLennard-Jones流体のサンプルコードを

```
00 #include <particle_simulator.hpp>
01 using namespace PS;
02
03 class FP{
04 public:
05   F64vec r,v,f;
06   void clear(){r = v = f = 0.0;}
07   void copyFromForce(const FP& fp){f = fp.f;}
08   F64 getRSearch() const {return 4.5;}
09   F64vec getPos() const {return r;}
10   void setPos(const F64vec &_r){r = _r;};
11   void copyFromFP(const FP &fp){(*this) = fp;}
12 };
13
14 struct Kernel{
15   void operator()(const FP *epi,const S32 ni,
16                   const FP *epj,const S32 nj,
17                   FP *force){
18     for(S32 i=0; i<ni; i++){
19       F64vec ri = epi[i].r; F64vec fi = 0.0;
20       for(S32 j=0; j<nj; j++){
21         const F64vec rij = ri - epj[j].r;
22         const F64 r2 = rij * rij;
23         if(r2==0.0 || r2>4.5*4.5) continue;
24         const F64 r2i = 1.0/r2;
25         const F64 r6i = r2i * r2i * r2i;
26         fi += r6i*(48.0*r6i-24.0)*r2i * rij;
27       }
28       force[i].f = fi;
29     }
30   }
31 };
32
33
34 int main(int argc,char **argv){
35   Initialize(argc,argv);
36   ParticleSystem<FP> ps;
37   ps.initialize();
38   if(Comm::getRank()==0){
39     ps.setNumberOfParticleLocal(1000);
40     S32 count = 0;
41     for(S32 x=0;x<10;x++)
42       for(S32 y=0;y<10;y++)
43         for(S32 z=0;z<10;z++){
44           ps[count].r = F64vec(x,y,z)-5.0;
45           ps[count++].v = 0.0;
46         }
47   }else ps.setNumberOfParticleLocal(0);
48   DomainInfo di;
49   di.initialize(0.3);
50   di.decomposeDomainAll(ps);
51   ps.exchangeParticle(di);
52   TreeForForceShort<FP,FP,FP>::Scatter t;
53   t.initialize(1000,0.0,64,256);
54   t.calcForceAllAndWriteBack(Kernel(),ps,di);
55   S32 nl = ps.getNumberOfParticleLocal();
56   const F64 dt = 0.005; const F64 dth = 0.5*dt;
57   for(int s=0;s<1000;s++){
58     for(int i=0;i<nl;i++) ps[i].v+=ps[i].f*dth;
59     for(int i=0;i<nl;i++) ps[i].r+=ps[i].v*dt;
60     di.decomposeDomainAll(ps);
61     ps.exchangeParticle(di);
62     nl = ps.getNumberOfParticleLocal();
63     t.calcForceAllAndWriteBack(Kernel(),ps,di);
64     for(int i=0;i<nl;i++) ps[i].v+=ps[i].f*dth;
65   }
66   Finalize();
67 }
```

FDPSを用いたコード開発



それではLennard-Jones流体のサンプルコ

粒子クラス

```
#include <particle_simulator.hpp>
using namespace PS;

class FP{
public:
  F64vec r,v,f;
  void clear(){r = v = f = 0.0;}
  void copyFromForce(const FP& fp){f = fp.f;}
  F64 getRSearch() const {return 4.5;}
  F64vec getPos() const {return r;}
  void setPos(const F64vec &_r){r = _r;};
  void copyFromFP(const FP &fp){(*this) = fp;}
};
```

相互作用カーネル

```
struct Kernel{
  void operator()(const FP *epi,const S32 ni,
                  const FP *epj,const S32 nj,
                  FP *force){
    for(S32 i=0; i<ni; i++){
      F64vec ri = epi[i].r; F64vec fi = 0.0;
      for(S32 j=0; j<nj; j++){
        const F64vec rij = ri - epj[j].r;
        const F64 r2 = rij * rij;
        if(r2==0.0 || r2>4.5*4.5) continue;
        const F64 r2i = 1.0/r2;
        const F64 r6i = r2i * r2i * r2i;
        fi += r6i*(48.0*r6i-24.0)*r2i * rij;
      }
      force[i].f = fi;
    }
  }
};
```

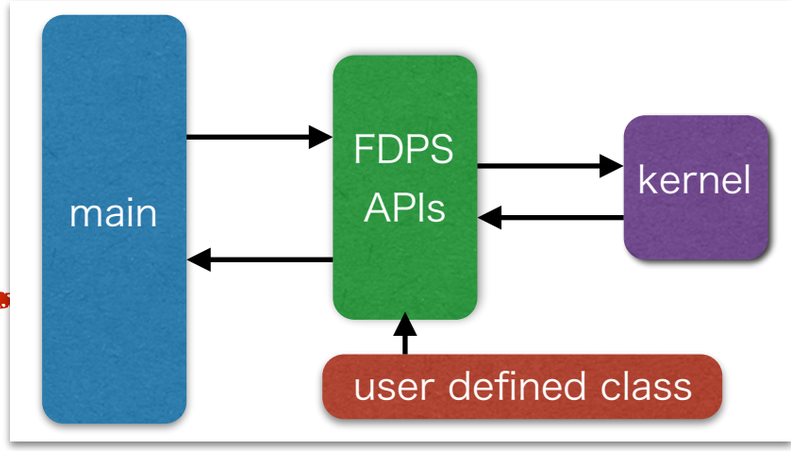
初期化

```
int main(int argc,char **argv){
  initialize(argc,argv);
  ParticleSystem<FP> ps;
  ps.initialize();
  if(Comm::getRank()==0){
    ps.setNumberOfParticleLocal(1000);
    S32 count = 0;
    for(S32 x=0;x<10;x++)
      for(S32 y=0;y<10;y++)
        for(S32 z=0;z<10;z++){
          ps[count].r = F64vec(x,y,z)-5.0;
          ps[count++].v = 0.0;
        }
  }else ps.setNumberOfParticleLocal(0);
  DomainInfo di;
  di.initialize(0.3);
  di.decomposeDomainAll(ps);
  ps.exchangeParticle(di);
  TreeForForceShort<FP,FP,FP>::Scatter t;
  t.initialize(1000,0.0,64,256);
```

時間発展

```
  S32 nl = ps.getNumberOfPar
  const F64 dt = 0.005; cons
  for(int s=0;s<1000;s++){
    for(int i=0;i<nl;i++) ps[i].v+=ps[i].f*dth;
    for(int i=0;i<nl;i++) ps[i].r+=ps[i].v*dt;
    di.decomposeDomainAll(ps);
    ps.exchangeParticle(di);
    nl = ps.getNumberOfParticleLocal();
    t.calcForceAllAndWriteBack(Kernel(),ps,di);
    for(int i=0;i<nl;i++) ps[i].v+=ps[i].f*dth;
  }
}
```

解説① 3つの主要クラス




```
00 #include <particle_simulator.hpp>
01 using namespace PS;
```

ヘッダーファイルをインクルード

particleSystem

粒子の情報を保持・通信

FullParticle
pos
vel
force
id

exchangeParticle()

domainInfo

計算領域の情報を保持

decomposeDomainAll()

copyFromFP

copyFromForce

treeForForce

相互作用計算を担当

calcForceAllAndWriteBack()

EssentialParticle
pos
id

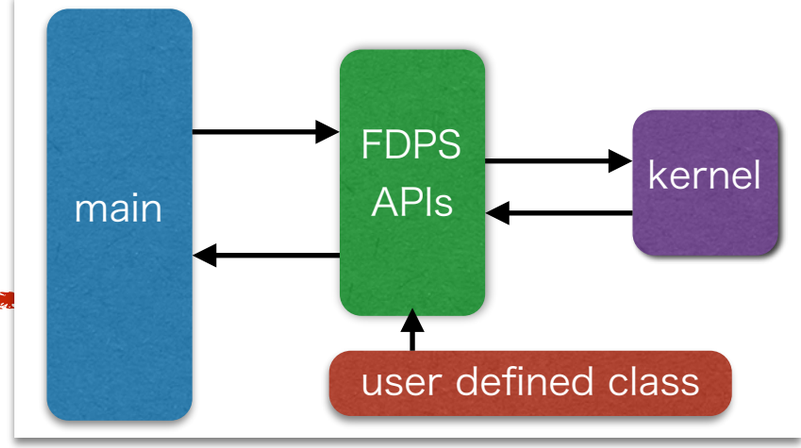
ツリー構造生成

相互作用リスト生成

ユーザ定義相互作用カーネル

Force
force

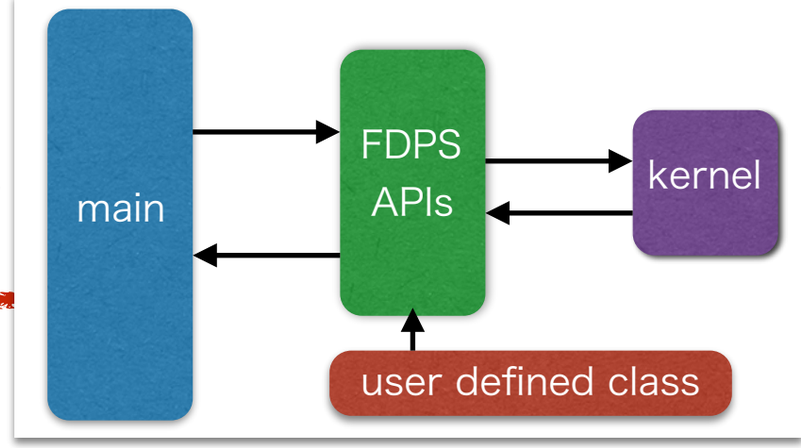
解説② 粒子クラスの定義




```
03 class FP{
04 public:
05    F64vec r,v,f;
06    void clear(){r = v = f = 0.0;}
07    void copyFromForce(const FP& fp){f = fp.f;}
08    F64 getRSearch() const {return 4.5;}
09    F64vec getPos() const {return r;}
10    void setPos(const F64vec &_r){r = _r;};
11    void copyFromFP(const FP &fp){(*this) = fp;}
12 };
```

- ここでは，粒子の全情報を持ったFullParticleクラス(FP)を定義
  - 粒子の座標(r)，速度(v)，力(f)を保持
  - FDPSのAPIに必要なメンバ関数を定義
- 力の計算に必要な力を持ったEssentialPraticleクラス(EPI/EPJ)や相互作用計算の計算結果を保持するForceクラスも設計できる
- 本サンプルでは簡単のために全てFPで代用している
- (基本的にはEPやForceはFPのサブセット)

解説③ 相互作用カーネルの定義



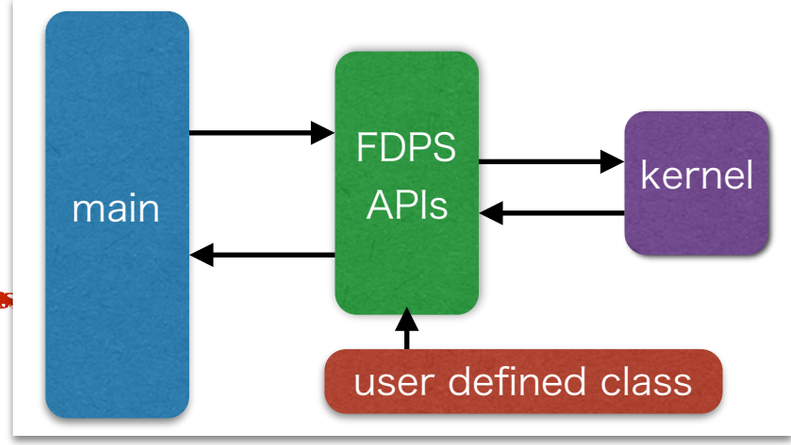

```
14 struct Kernel{
15   void operator()(const FP *epi,const S32 ni,
16                   const FP *epj,const S32 nj,
17                   FP *force){
18     for(S32 i=0; i<ni; i++){
19       F64vec ri = epi[i].r; F64vec fi = 0.0;
20       for(S32 j=0; j<nj; j++){
21         const F64vec rij = ri - epj[j].r;
22         const F64 r2 = rij * rij;
23         if(r2==0.0 || r2>4.5*4.5) continue;
24         const F64 r2i = 1.0/r2;
25         const F64 r6i = r2i * r2i * r2i;
26         fi += r6i*(48.0*r6i-24.0)*r2i * rij;
27       }
28       force[i].f = fi;
29     }
30   }
31 };
```

引数は決まっている

i粒子のEPのリスト，リスト長
j粒子のEPのリスト，リスト長，
力を保持するForceのリスト
(ここではEPもForceもFPで代用)

$$F_i = \sum_{j=0}^{N} \left\{ 48 \left(\frac{1}{|\boldsymbol{r}_{ij}|} \right)^{12} - 24 \left(\frac{1}{|\boldsymbol{r}_{ij}|} \right)^{6} \right\} \frac{\boldsymbol{r}_{ij}}{|\boldsymbol{r}_{ij}|^2}$$ を計算

解説④ main関数(初期条件)




```
34 int main(int argc,char **argv){
35   Initialize(argc,argv);
36   ParticleSystem<FP> ps;
37   ps.initialize();
38   if(Comm::getRank()==0){
39     ps.setNumberOfParticleLocal(1000);
40     S32 count = 0;
41     for(S32 x=0;x<10;x++)
42       for(S32 y=0;y<10;y++)
43         for(S32 z=0;z<10;z++){
44           ps[count].r = F64vec(x,y,z)-5.0;
45           ps[count++].v = 0.0;
46         }
47   }else ps.setNumberOfParticleLocal(0);
48   DomainInfo di;
49   di.initialize(0.3);
50   di.decomposeDomainAll(ps);
51   ps.exchangeParticle(di);
52   TreeForForceShort<FP,FP,FP>::Scatter t;
53   t.initialize(1000,0.0,64,256);
54   t.calcForceAllAndWriteBack(Kernel(),ps,di);
```

ParticleSimulatorの初期化

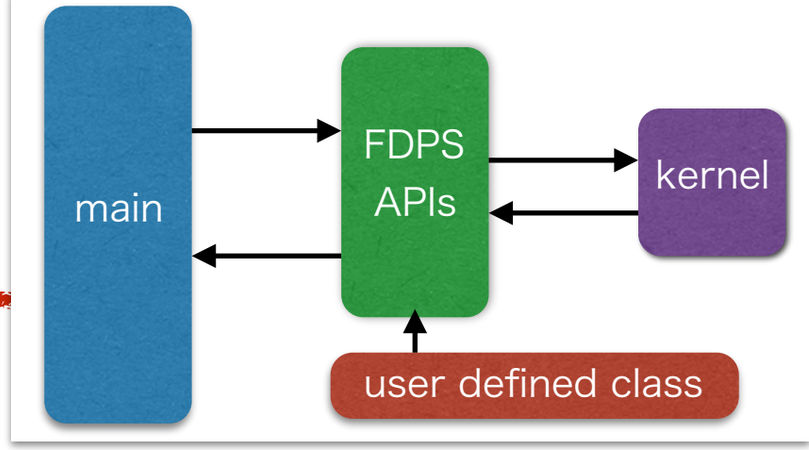
立方格子上に1000分子を配置

領域の分割

粒子の交換

相互作用計算

解説⑤ main関数(時間発展)




```
55    S32 nl = ps.getNumberOfParticleLocal();
56    const F64 dt = 0.005; const F64 dth = 0.5*dt;
57    for(int s=0;s<1000;s++){
58      for(int i=0;i<nl;i++) ps[i].v+=ps[i].f*dth;
59      for(int i=0;i<nl;i++) ps[i].r+=ps[i].v*dt;
60      di.decomposeDomainAll(ps);
61      ps.exchangeParticle(di);
62      nl = ps.getNumberOfParticleLocal();
63      t.calcForceAllAndWriteBack(Kernel(),ps,di);
64      for(int i=0;i<nl;i++) ps[i].v+=ps[i].f*dth;
65    }
66    Finalize();
67 }
```

速度ベルレ法で時間発展

領域の分割，粒子の交換

力の計算

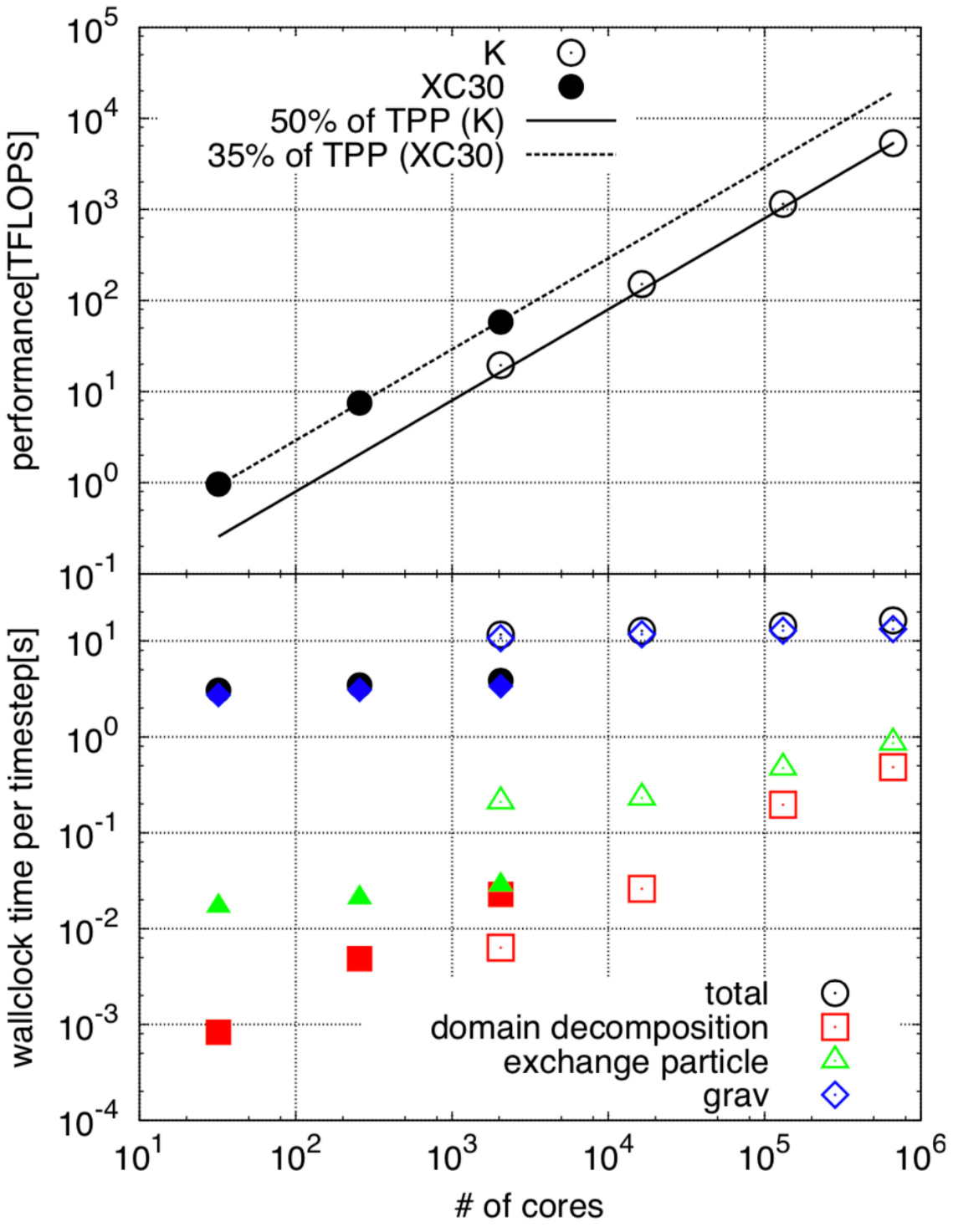
速度ベルレ法で時間発展

ParticleSimulatorの終了処理

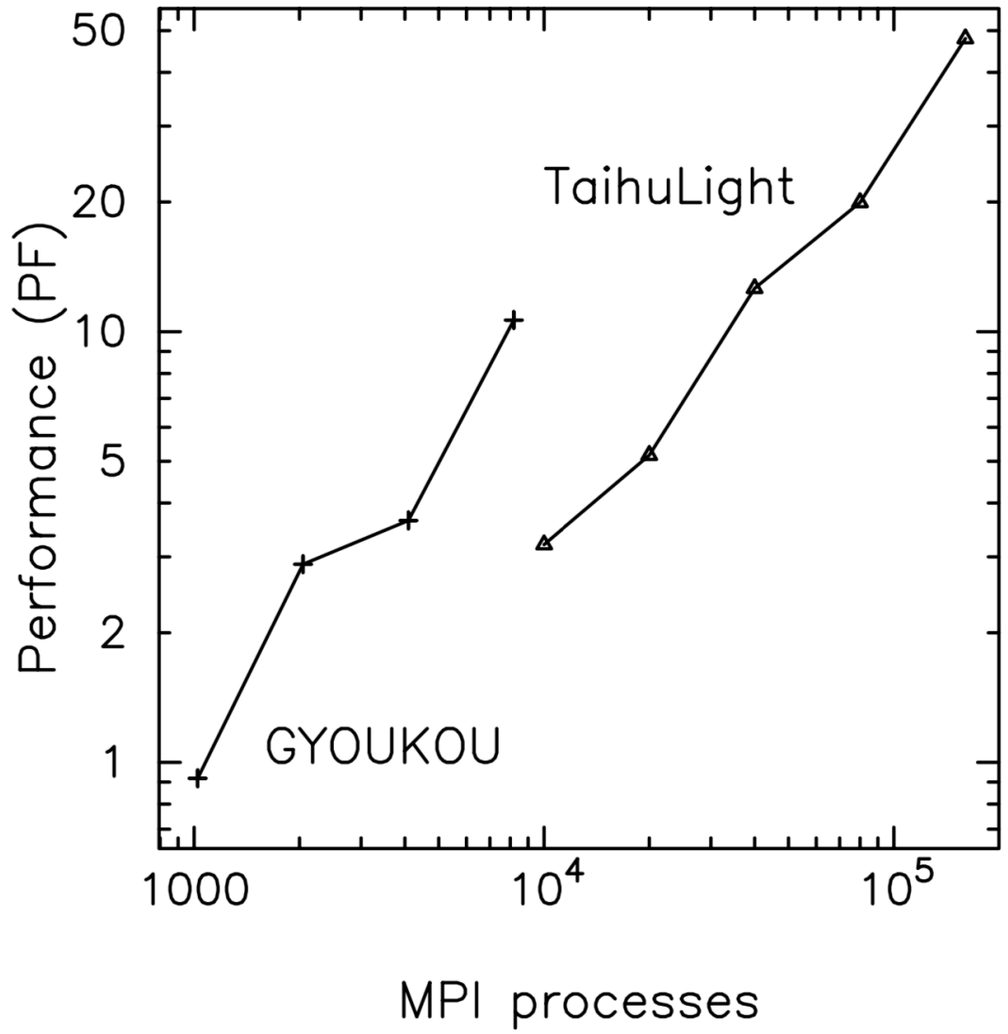
70行程度でMPI/OpenMPハイブリッド並列のMDコードが完成！
(あと出力いれるとか最適化が必要ですが…)

ますは https://github.com/FDPS/FDPS からダウンロード

並列化性能



Iwasawa+, PASJ(2016)



- 様々なスパコンで高い並列化効率
- 相互作用カーネルの速度が性能に大きく影響
- これまではエキスパートたちの手によって素晴らしいカーネルが作られてきたが…

唐突ですがここで富岳概要

https://news.livedoor.com/article/detail/18423524/

総ノード数

総ノード数	158,976ノード 384ノードx 396 ラック= 152,064 192 ノードx 36 ラック= 6,912

総理論性能

総演算性能	通常モード（CPU動作クロック周波数2GHz）	• 倍精度理論最高値（64bit）488ペタフロップス • 単精度理論最高値（32bit）977ペタフロップス • 半精度（AI学習）理論最高値（16bit）1.95エクサフロップス • 整数（AI推論）理論最高値（8bit）3.90 エクサオップス
	ブーストモード（CPU動作クロック周波数2.2GHz）	• 倍精度理論最高値（64bit）537ペタフロップス • 単精度理論最高値（32bit）1.07エクサフロップス • 半精度（AI学習）理論最高値（16bit）2.15エクサフロップス • 整数（AI推論）理論最高値（8bit）4.30 エクサオップス
総メモリ容量		4.85 PiB
総メモリバンド幅		163 PB/s

ノード単体性能

命令セットアーキテクチャ		Armv8.2-A SVE 512bit 富士通拡張:ハードウェアバリア、セクタキャッシュ、プリフェッチ
計算コア数		48 + 2アシスタントコア 4 CMG (Core Memory Group, NUMA nodeのこと)
演算性能	通常モード（CPU動作クロック周波数2GHz）	倍精度: 3.072 TF, 単精度: 6.144 TF, 半精度: 12.288 TF
	ブーストモード（CPU動作クロック周波数2.2GHz）	倍精度: 3.3792 TF, 単精度: 6.7584 TF, 半精度: 13.5168 TF
キャッシュ*1 *2		L1D/core: 64 KiB, 4way, 256 GB/s (load), 128 GB/s (store)
		L2/CMG: 8 MiB, 16way L2/node: 4 TB/s (load), 2 TB/s (store) L2/core: 128 GB/s (load), 64 GB/s (store)
メモリ		HBM2 32 GiB, 1024 GB/s
インターコネクト		Tofu Interconnect D (28 Gbps x 2 lane x 10 port)
I/O		PCIe Gen3 x16
テクノロジー		7nm FinFET

https://www.r-ccs.riken.jp/jp/post-k/overview.html

PIKG

- (Paricle-particle Interaction Kernel Generatorの略)
- 相互作用計算をDSLで簡単に記述
- ジェネレータにDSLコードとオプション(アーキテクチャの指定とか)を与えると最適化された粒子間相互作用カーネルコードがでてくる
- ユーザーは(ほとんど)最適化については考えない
- 単一コードから複数アーキテクチャの最適化コードを生成

使い方

- ユーザは相互作用を及ぼされる粒子(EPI)と及ぼす粒子(EPJ)とEPIにかかる相互作用(FORCE)の変数を定義
- 定義した関数などを用いて2粒子間相互作用の計算を記述
- オプション等を設定してジェネレータにかけるとカーネルができる

例(N体)

Variable definition:

[*class_type*] *type* var*name* [: *member_name*]

Function definition:

function *name*(*variables*...)
  [*statements*...]
  return *val*
end

Kernel definition:

variable (=|+=|-=) *expression*

```
EPI F32vec xi:pos
EPI F32    epsi:eps
EPJ F32vec xj:pos
EPJ F32    mj:mass
EPJ F32    epsj:eps
FORCE F32vec f:acc
FORCE F32 phi:pot
```

```
function sub(a,b)
  return a-b
end
```

```
rij = xi - xj
r2 = epsi*epsi + rij*rij
rinv = rsqrt(r2)
mrinv = mj*rinv
f -= mrinv*rinv*rinv * rij
phi -= mrinv
```

生成コード(最適化なし)

```
template<typename Tepi,typename Tepj,typename Tforce>
struct Kernel{
  Kernel(){}
  void operator()(const Tepi* epi,const int ni,const Tepj* epj,const int nj,Tforce *force){
    for(int i=0;i<ni;i++){
      PS::F32vec xi = epi[i].pos;
      PS::F32 epsi = epi[i].eps;
      PS::F32vec f = force[i].acc;
      PS::F32 phi = force[i].pot;
      for(int j=0;j<nj;j++){
        PS::F32vec xj = epj[j].pos;
        PS::F32 mj = epj[j].mass;
        PS::F32 epsj = epj[j].eps;
        PS::F32vec rij;
        rij.x = (xi.x-xj.x);
        rij.y = (xi.y-xj.y);
        rij.z = (xi.z-xj.z);
        PS::F32 r2 = madd<PS::F32,PS::F32,PS::F32,PS::F32>(epsi,epsi,madd<PS::F32,PS::F32,PS::
F32,PS::F32>(rij.x,rij.x,madd<PS::F32,PS::F32,PS::F32,PS::F32>(rij.y,rij.y,(rij.z*rij.z))));
        PS::F32 rinv = rsqrt<PS::F32,PS::F32>(r2);
        PS::F32 mrinv = (mj*rinv);
        PS::F32 __fkg_tmp0 = ((mrinv*rinv)*rinv);
        f.x = nmsub<PS::F32,PS::F32,PS::F32,PS::F32>(__fkg_tmp0,rij.x,f.x);
        f.y = nmsub<PS::F32,PS::F32,PS::F32,PS::F32>(__fkg_tmp0,rij.y,f.y);
        f.z = nmsub<PS::F32,PS::F32,PS::F32,PS::F32>(__fkg_tmp0,rij.z,f.z);
        phi = (phi-mrinv);
      }
      force[i].acc = f;
      force[i].pot = phi;
    }
  }
  template<typename Tret,typename Ta,typename Tb>
  Tret sub(Ta a,Tb b){
    return (a-b);
  }
  template<typename Tret,typename Top>
  Tret rsqrt(Top op){ return (Tret)1.0/std::sqrt(op); }
  template<typename Tret,typename Top>
  Tret sqrt(Top op){ return std::sqrt(op); }
  template<typename Tret,typename Top>
  Tret inv(Top op){ return 1.0/op; }
  template<typename Tret,typename Ta,typename Tb,typename Tc>
  Tret madd(Ta a,Tb b,Tc c){ return a*b+c; }
  template<typename Tret,typename Ta,typename Tb,typename Tc>
  Tret msub(Ta a,Tb b,Tc c){ return a*b-c; }
  template<typename Tret,typename Ta,typename Tb,typename Tc>
  Tret nmadd(Ta a,Tb b,Tc c){ return -(a*b+c); }
  template<typename Tret,typename Ta,typename Tb,typename Tc>
  Tret nmsub(Ta a,Tb b,Tc c){ return c-a*b; }
};
```

富岳向け最適化

- ループ分割
  - 長い(というわけでもないものも)ループを分割してレジスタへの負荷を減らす
- ストリップマイニング
  - L1$にのるようにループの回転数を分割
- ループアンロール
  - (とても)長い命令レイテンシを隠蔽

元コード

```
for(int i=0;i<N;i++){ A(i); B(i);}
```

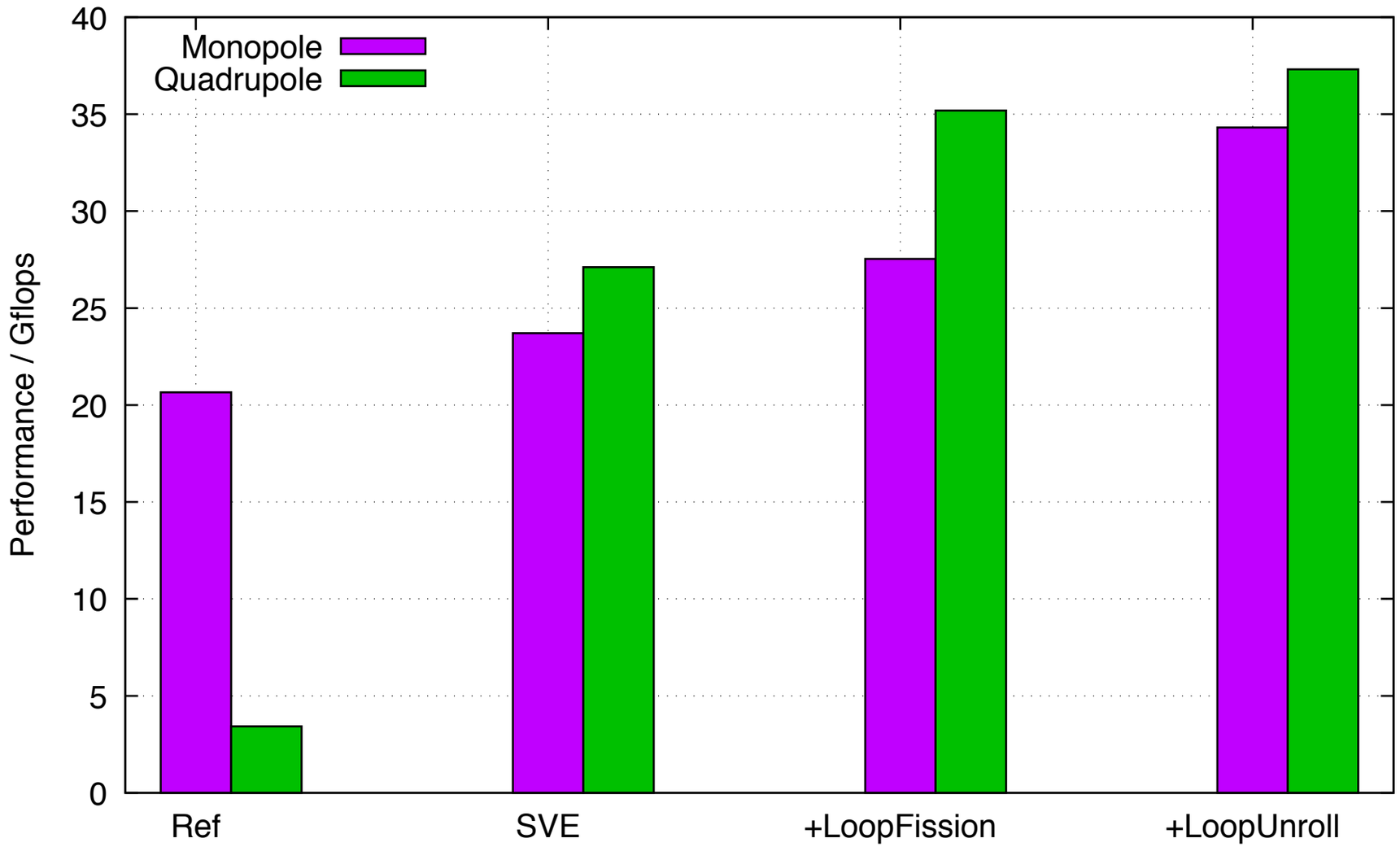
```
for(int i=0;i<N;i++){ A(i);}
for(int i=0;i<N;i++){ B(i);}
```

```
for(int i=0;i<N/n;i++){
  for(int j=0;j<n;j++) A(n*i+j);
  for(int j=0;j<n;j++) B(n*i+j);
}
```

```
for(int i=0;i<N/n;i++){
  for(int j=0;j<n/m;j+=m){
    A(n*i+j+0); … A(n*i+j+m-1);
  }
  for(int j=0;j<n/m;j+=m){
    B(n*i+j+0); … B(n*i+j+m-1);
  }
}
```

最適化の効果

A64FX 2.0 GHz (1コア, 単精度ピーク128Gflops)
コンパイラ: FCC 4.1.0 20200415
オプション: -Kfast -Nclang
条件: 512 x 512のN²ループ



※富岳共用開始後の性能を保証するものではありません

まとめ

- 粒子系シミュレータ開発フレームワークFDPSの紹介
  - FDPSのAPIを用いることで並列化(MPI/OpenMP)に関わる部分を考えずにシミュレータの実装を行える
  - 使ってみたい方はぜひ初級講習会へ(今のところ8月3週の予定)

- 粒子間相互作用計算カーネルジェネレータPIKGの紹介
  - DSLで粒子間相互作用計算を記述してジェネレータにかけると，オプションによってさまざまなアーキテクチャに最適化されたカーネルが生成される
  - 現在ARM SVE/AVX-512/AVX2向けの最適化が利用可能
  - GPU(CUDA)やPEZY-SC2などにも対応する予定

謝辞

本研究は，文部科学省「富岳」成果創出加速プログラム「宇宙の構造形成と進化から惑星表層環境変動までの統一的描像の構築」の一環として実施したものです。

その他富岳を使って

- 使い勝手としては基本的に京と変わらない(らしい)
- できたてのシステムではよくあることでしょうが，実装の過程でコンパイラのバグを何個か踏んだ．対処法はあるが直ってなかったりするのでポータルのレポートやマニュアル(既知の問題とか)はしっかり読みましょう
- A64FXの詳細はA64FX_Microarchitecture_Manualを参照
- (頑張って最適化したい人向け)A64FXの気持ちになるためには理研シミュレータ＋gem5可視化ソフト(Konata)が便利
  - ロボ太先生(@kaityo256)の記事がわかりやすい
  - https://qiita.com/kaityo256/items/00fc50221d86ce3ff2ea