次世代スーパーコンピューター「京」にむけた分子動力学コードの最適化

理化学研究所

次世代計算科学研究開発プログラム
次世代生命体統合シミュレーション研究推進グループ
生命体基盤ソフトウェア開発・高度化チーム

大野　洋介

概要

- 「京」の特徴とアプリケーションへの要求
- 分子動力学コードの場合
  - 通信
  - 大規模並列
  - スレッド並列
- まとめ

おわび

- 「京」の実機はまだ非公開
  - 「京」固有の最適化の話は実話ではなく、公開情報や代行実行の結果からの推測
  - 実例は RICC(理研の x86_64 クラスタ) 等での話
  - あとは一般論

「京」の特徴

- ノード数が多い
  - 80000 ノード以上
- TOFU ネットワーク
  - トーラス
  - 比率可変
  - 同時通信数 4 < コア数 8
- スレッド並列時に効果のある機能
  - セクターキャッシュ
  - ハードウェアバリア

次世代スーパーコンピュータ「京」

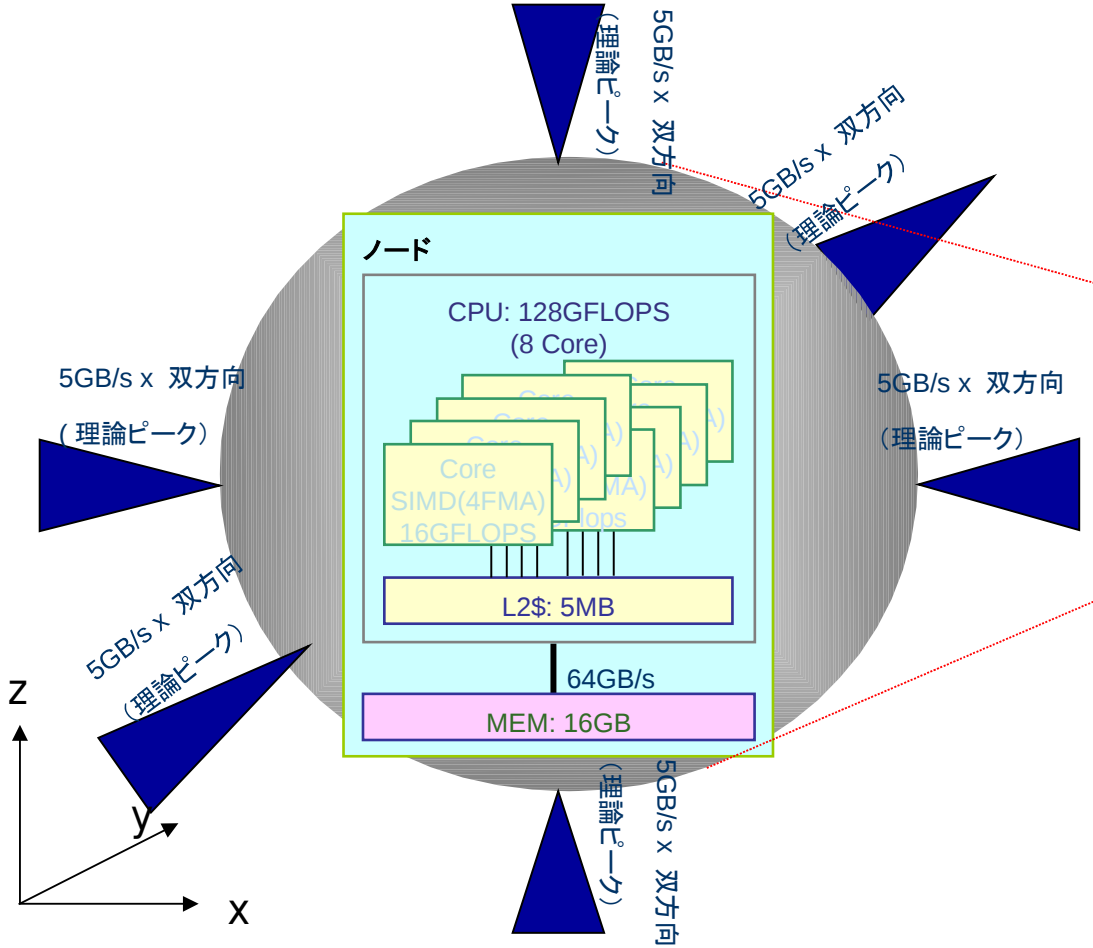
- 国産で世界二位を目指す，汎用スーパーコンピュータの開発プロジェクト
- さまざまな分野での利用（地球シミュレータと異なる性格）

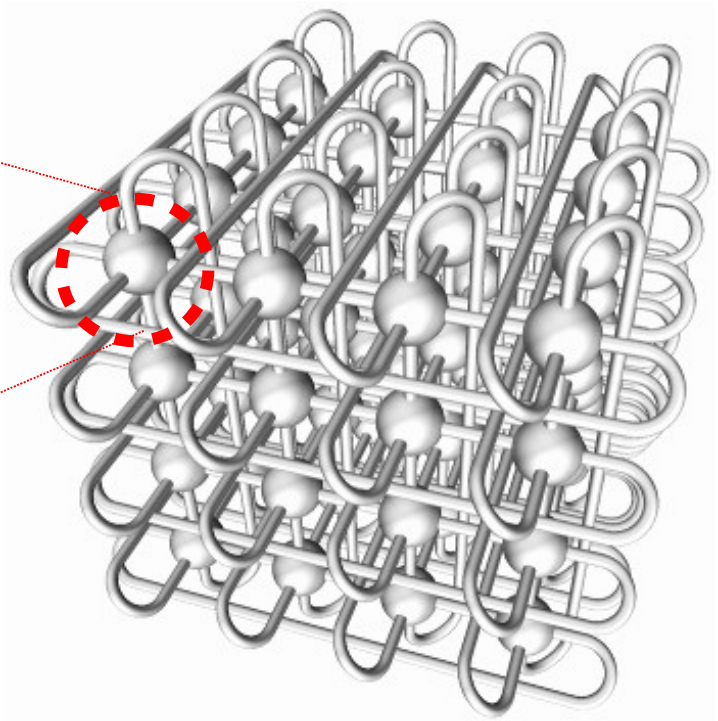
日本における高性能計算・計算科学の拠点を形成

- 神戸ポートアイランドに設置
- 開発企業：富士通
- 性能：10 Peta FLOPS
  LINPACK
- 2011 春　部分稼働開始
- 2012 秋　供用開始

「京」本体システムの構成図

- 計算ノード数: 8 万以上
  - CPU 数: 8 万以上
  - コア数: 64 万以上
- ピーク演算性能: 10PFLOPS 以上
- メモリ総容量: 1PB 以上(ノード当り 16GB)
- ネットワーク: Tofu インターコネクト(ユーザービューは 3D トーラス)
  - 帯域幅: 3 次元の正負各方向にそれぞれ 5GB/s x 2 (双方向、理論ピーク)
  - 同時通信数 :4
  - バイセクションバンド幅:
    30TB/s (双方向、理論ピーク)以上





3D トーラスのイメージ

アプリケーションへの要求

- トーラス

  近接通信が有利

  - 遠くのノードはレイテンシが増える
    - FFT には不利
- 同時通信数4

  フラット MPI だと通信競合が発生しやすい

  スレッド並列とのハイブリッド並列が望ましい

プロセッサ　SPARC64™ VIIIfx

- 富士通の 45nm 世代サーバプロセッサ
- 8 コア、8 演算 / コア ＝ 64 演算 / サイクル
- 128GFLOPS @ 2GHz
- 58W
- 「 VISIMPACT 」SPARC64TM VII から
- 「 HPC-ACE 」　HPC 向け拡張
- 資料は以下
  http://jp.fujitsu.com/solutions/hpc/brochures/

SPARC64™ VIIIfx の基本アーキテクチャ

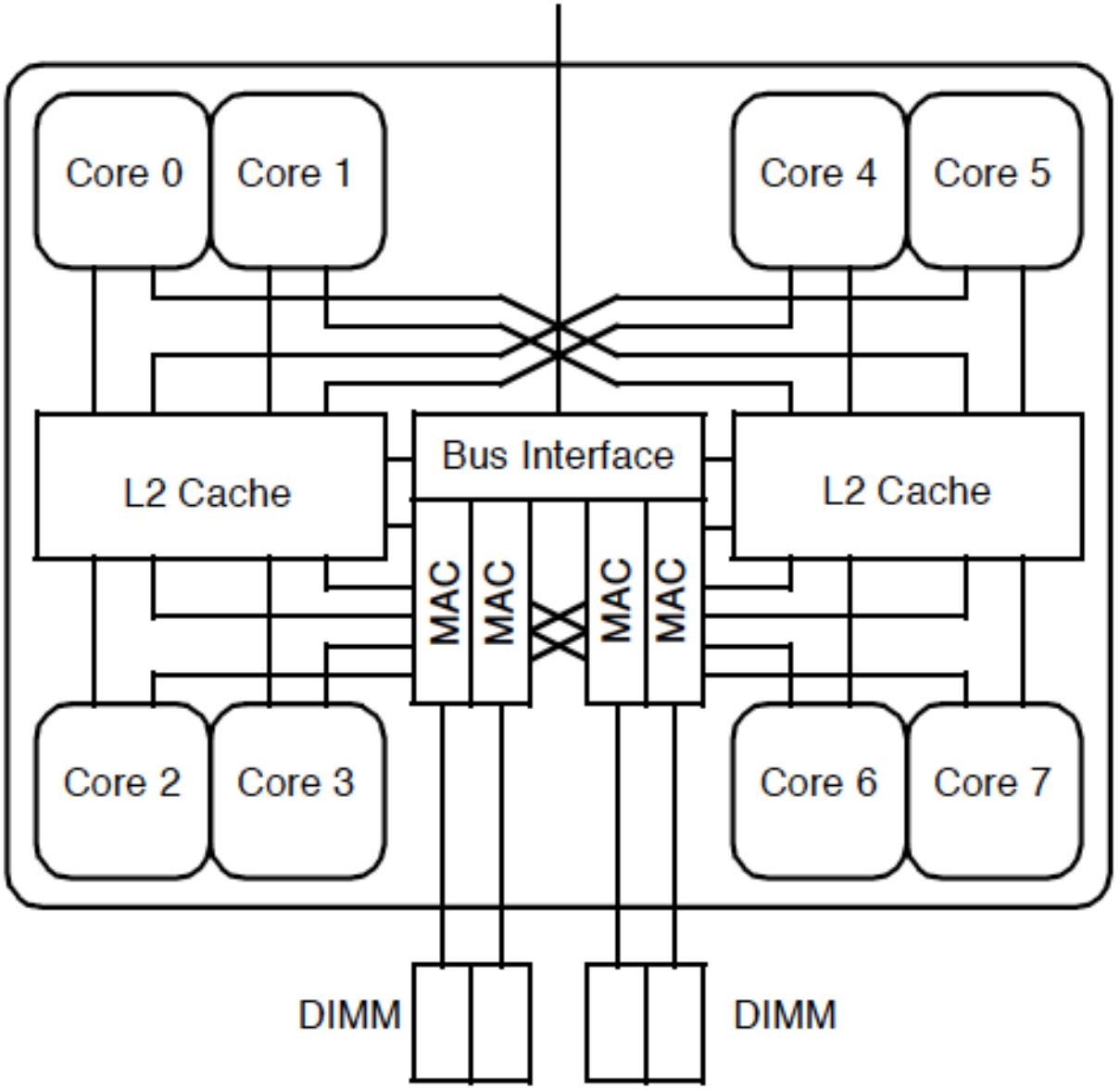
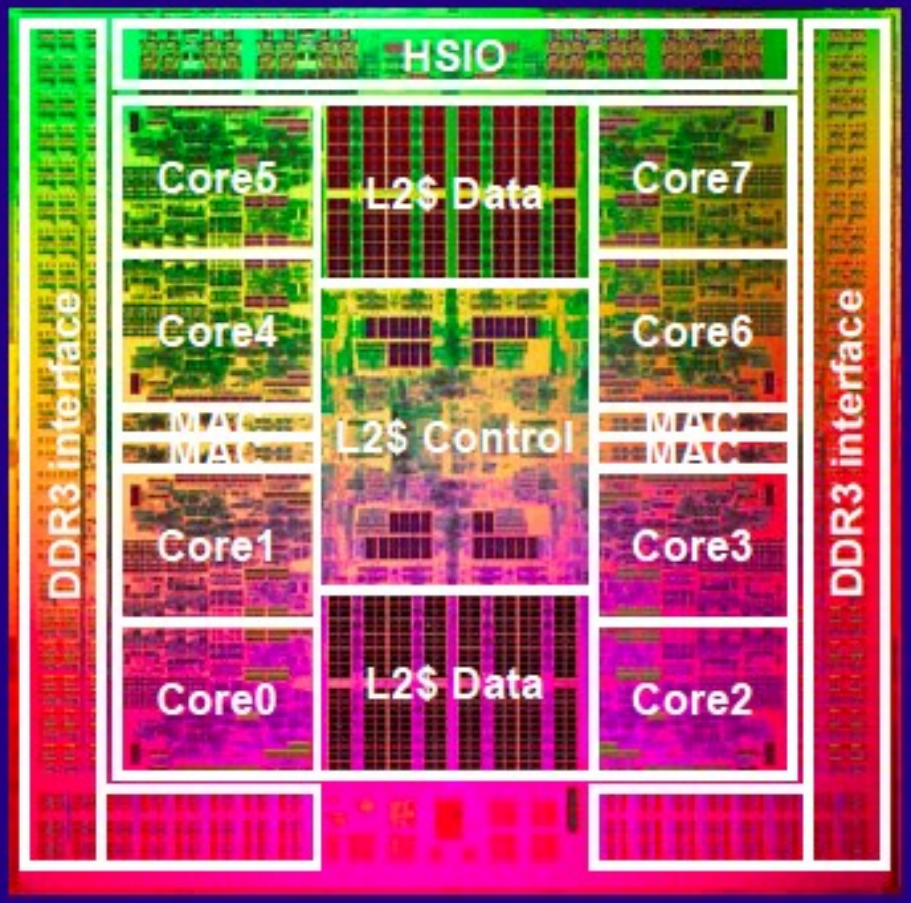


FIGURE 3-1 SPARC64 VIIIfx Block Diagram

- 8-core
- Shared L2
  6MB
- DDR3 interface

富士通株式会社
SPARC64TM VIIIfx Extensions
日本語版 , Ver15, p10.

SPARC64™ VIIIfx Chip Overview



- **Architecture Features**
  - 8 cores
  - Shared 6 MB L2$
  - Embedded Memory Controller
  - 2 GHz
- **Fujitsu 45nm CMOS**
  - 22.7mm x 22.6mm
  - 760M transistors
  - 1271 signal pins
- **Performance (peak)**
  - 128GFlops
  - 64GB/s memory throughput
- **Power**
  - 58W (TYP, 30°C)
  - Water Cooling – Low leakage power and High reliability

SPARC64™ VIIIfx

12

T. Maruyama, Proc. Hot Chips 2009.

Byte per FLOP

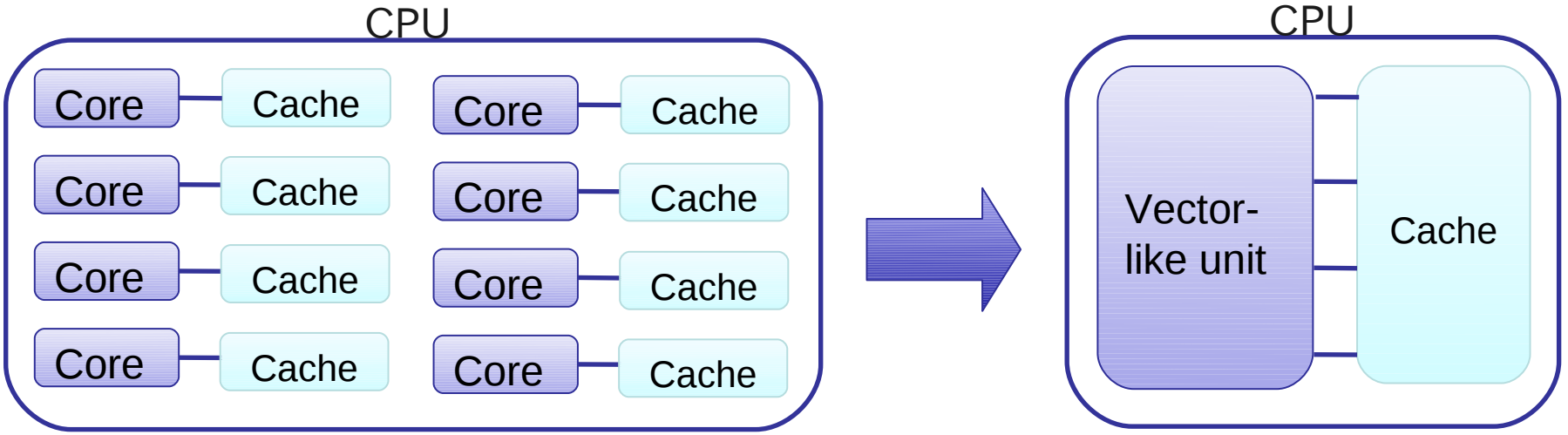
- 主記憶 -L2
  0.5 B/F (64GB/sec, 128GFLOPS)
- L2-L1
  2 B/F
- L1-Register
  4 B/F

特徴的な機能

- VISIMPACT
  - Virtual Single Processor by Integrated Multi-core Parallel Architecture
  - 共有キャッシュ
  - ハードウェアバリアによるコア間同期
- HPC-ACE
  - High Performance Computing – Arithmetic Computational Extensions
  - レジスタ拡張
  - セクタキャッシュ
  - SIMD

VISIMPACT

- 複数のコアを一つのコアに見せる



- 直接の利用：OpenMP
- 自動並列化：ベクトル類似

最内ループの並列化

ハードウェアバリア

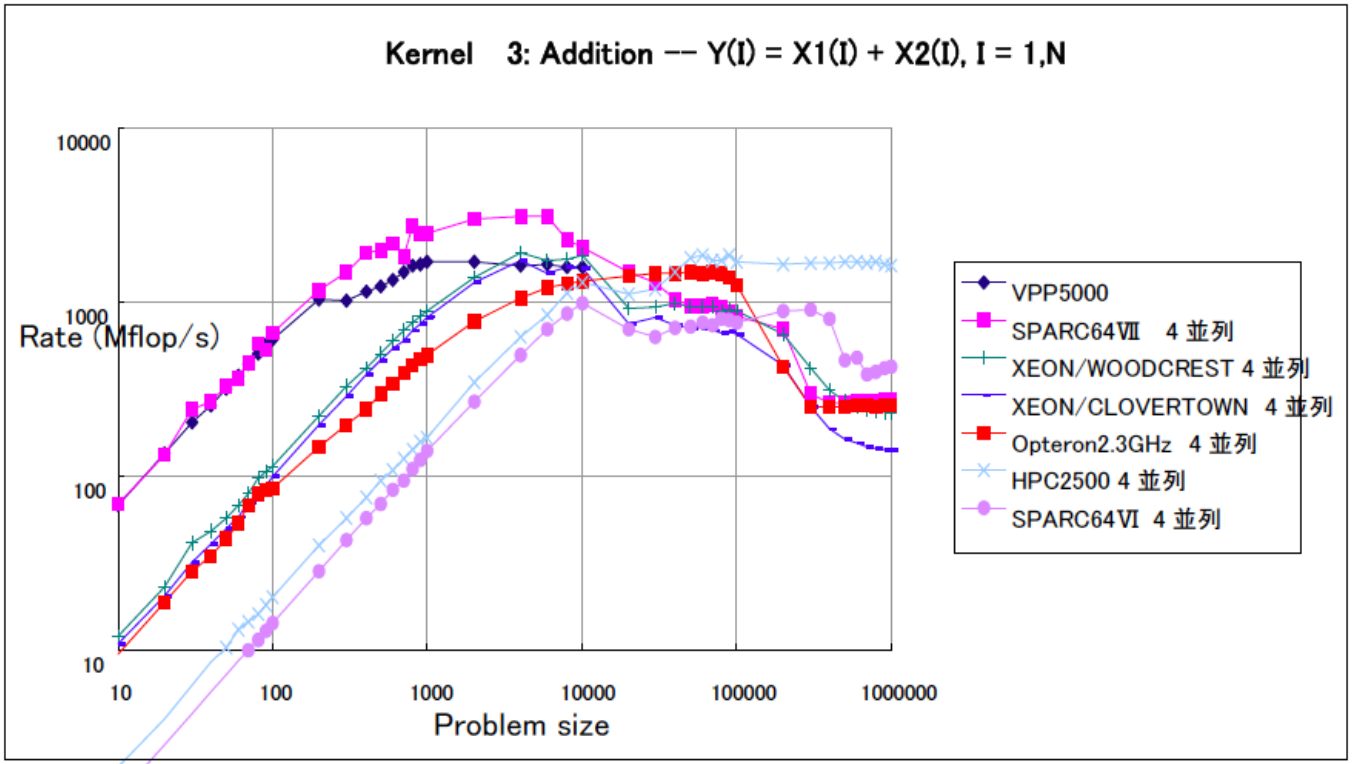
- Barrier, post / wait に対応
- 8 コア Barrier 向け Barrier Blade 4 つ
- Post / wait 向け Barrier Blade 8 つ
- Barrier status bit(BST) が揃ったとき同期
  - Barrier 向け 8bit(マスク可能)
  - post / wait 向け 1bit
- Last Barrier SYnchronization status(LBSY) ビットにコピー
- sleep 解除

ハードウェアバリアの効果: OpenMP

	SPARC64Ⅶ	SPARC64VI (2.28GHz)		Opteron (1.9GHz)		WoodCrest (3.00GHz)		CloverTown (2.66GHz)	HPC2500 (1.3 GHz)
	4 並列	2 並列	4 並列	4 並列 (FJ)	4 並列 (Intel)	2 並列	4 並列	4 並列	4 並列
PARALLELDO	0.30	3.90	9.61	1.93	1.89	0.72	2.15	2.20	2.05
DO	0.10	1.80	6.72	1.08	1.05	0.34	0.79	0.87	0.26
バリア	0.10	1.82	6.69	1.07	1.00	0.36	0.75	0.87	0.27
REDUCTION	0.50	4.16	17.46	3.29	3.35	1.20	3.17	3.35	4.43

単位： μ 秒

青木正樹、SS 研技術 WG 成果報告書

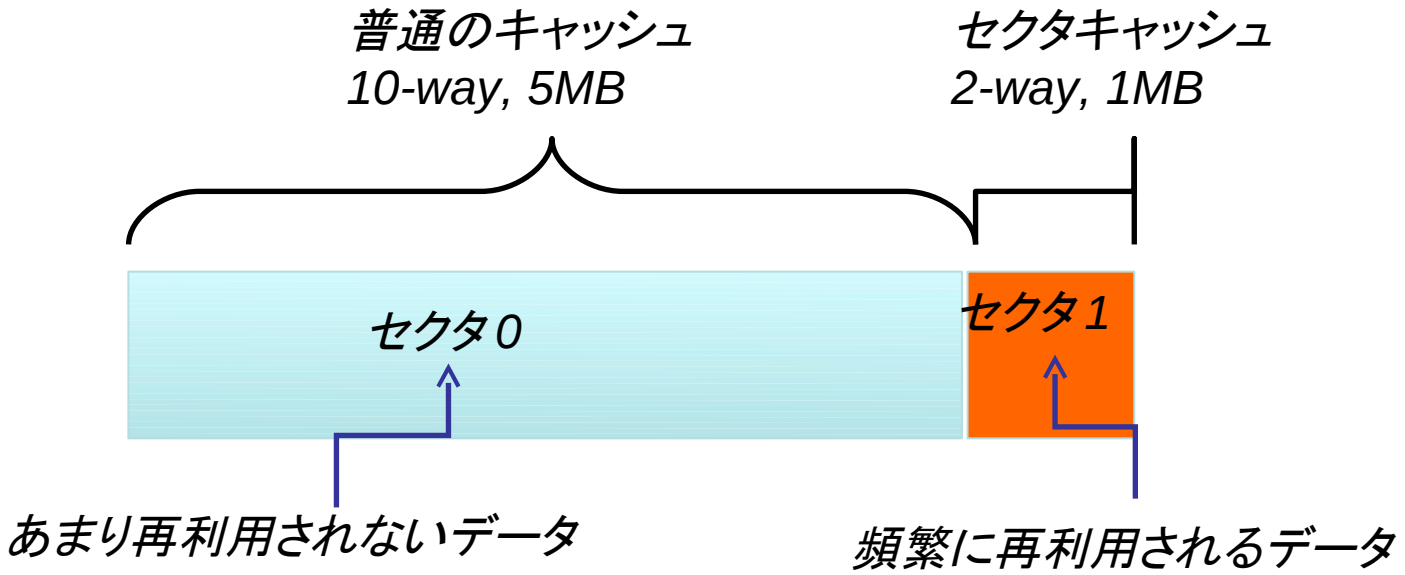


HPC-ACE: キャッシュ

- ２次キャッシュ
  - ８コア共有
  - 6MB, 12 way, 512KB/way
  - 2Byte/FLOP
- １次キャッシュ
  - コア毎独立
  - D-cache:32Kbytes, 2-way
  - I-cache：32Kbytes, 2-way
  - 4Byte/FLOP
- Line Size: 128 bytes

HPC-ACE: セクタキャッシュ(1)

- キャッシュを分割し、ローカルメモリのように利用可能



- 頻繁に再利用される特定のアドレス空間をセクタキャッシュに割り当て、効率を向上

HPC-ACE: セクタキャッシュ(2)

- 任意の way 数をセクタキャッシュに割り当て可能
- L1/L2 共に可能
- pragma 文で指定可能

```
!OCL L2_SECTOR_NWAYS_LOOP(10,2)
!OCL SECTOR1_DATA(a)
do j=1,m
  do i=1,n
     a(i) = a(i) + b(i,j) * c(i,j)
  enddo
enddo
```

- 機械語では、XAR レジスタを利用

アプリケーションへの要求

- セクターキャッシュで L2 のヒット率があがる
  - 再利用頻度が高いとわかっているデータを明示的に優先させる
- メモリ帯域が問題になる計算で効果的

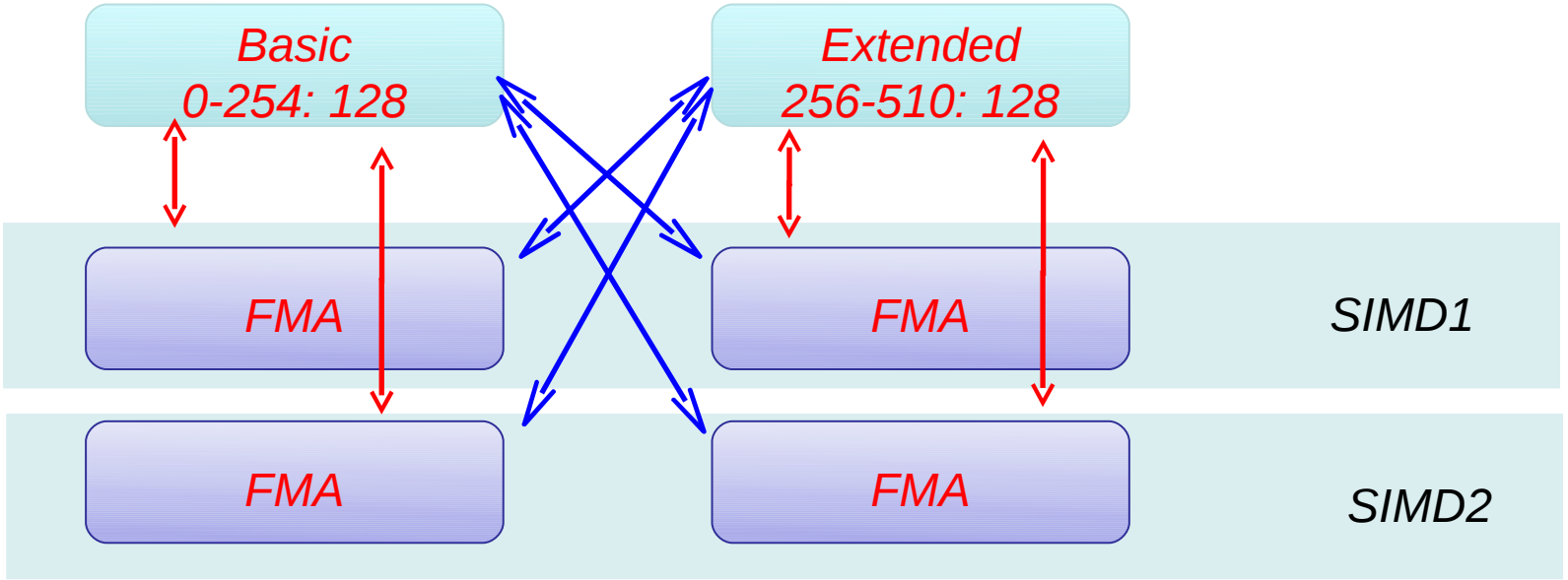
  ただし、スレッド並列でないと使えない

HPC-ACE: レジスタ拡張

- 256 本の倍精度浮動小数点レジスタ
  (単精度でも使える)
  ← 32 in SPARC64 VII
- Flat にアクセス可能
- SIMD 命令実行時には、basic(0-254) と extended(256-510) に分かれる
  ※ 倍精度浮動小数のレジスタ番号は偶数のみ
- Unrolling / Software Pipelining に有効

HPC-ACE: SIMD (1) 基本

- 2FMA の SIMD 命令を同時に 2 命令実行可能
- 2 命令 x 2SIMD x 2FMA = 8 演算
- Basic(0-254) と Extended(256-510) レジスタを使って独立に SIMD 実行。指定するのは Basic の番号のみ。(例外あり: FMADD)
- ロードストアは 8 バイト境界で可能



アプリケーションへの要求

- FMA　積和演算
  - かけ算の結果が足し算の入力に直結

  d[i] = a[i]*b + c　のようなかけて足すのに最適

  無関係なかけ算と足し算は同時にできない

  かけ算だけ、足し算だけが連続するのは損

HPC-ACE: SIMD (2) 実行できない命令

SIMD 実行できない命令

- 整数命令
- フラグに結果を立てる比較命令
  - SIMD 向けの比較命令あり
- 除算 (FDIV)、開平 (FSQRT)
  - SIMD 向けの近似命令あり
- 可視化向け演算(Pixel 演算など)
- その他、分岐命令など自明なもの

Ibid, p22, p59-65

HPC-ACE: 条件実行

SIMD でのマスク実行のための命令

- FCMP(条件) %frs1,%frs2,%frd
  条件に応じ %frd の全ビットをセット
- FSELMOV %frs1, %frs2, %frs3, %frd
  %frs3 の最上位ビットが 1 なら %frd←%frs1,
  0 なら %frd←%frs2
- ST(D)FR %frd, %frs2, [addr]
  %frs2 の最上位ビットが 1 なら %frd をストア

SIMD 化できる条件分岐の実例

```
for (j=0; j<nj; j++) {
  dr = rj - ri;
  r2 = dr*dr;
  if (r2 < cutoff^2) {
    s = 1.0;
  }else{
    s = 0.0;
  }
  fi += s * f(r2) *dr;
}
```

その他の命令拡張

- FMAX / FMIN： 最大･最小
- POPC：1 の数を数える
- FRCPA: 逆数近似（誤差 <1/256 ）
- FRSQRTA: 逆数平方根近似（誤差 <1/256 ）
- 三角関数評価のための命令群

アプリケーションへの要求

- 1/x,1/sqrt(x) の SIMD 近似値命令とイテレーション
  SIMD 化しないと遅い割り算器が使われる

- トーラスネットワーク　近接通信が速い
  - 近接以外のノードも全て近接への経路を通過

  大域通信は少ない方がいい
- スレッド並列、SIMD 化で

  通信競合がへる

  ハードウェアバリアが使える

  セクターキャッシュでメモリ帯域節約

  1/x,1/sqrt(x) の高速化

  スレッド並列でないと効率低下

分子動力学コードの場合　前提条件

- 水中タンパク質の全原子
  - そのままでは $O(N^2)$
- 空間分割　セルインデックス
- 直接和は近距離カットオフ　$O(N)$

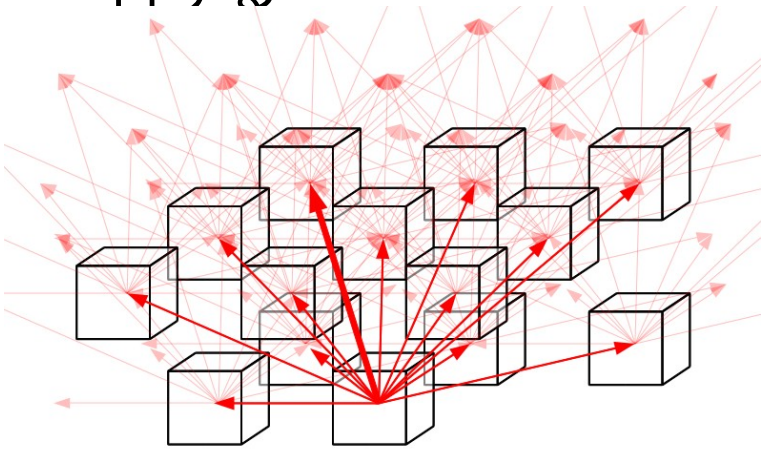
} 近接通信でよい

- 遠距離は近似
  - PME　波数空間、FFT $O(N \log N)$ <-　現在の主流
  - FMM　$O(N)$
  - 実空間ポアソン解法　MultiGrid $O(N)$
  - 近距離へ繰り込み　$O(N)$
  - 無視

通信

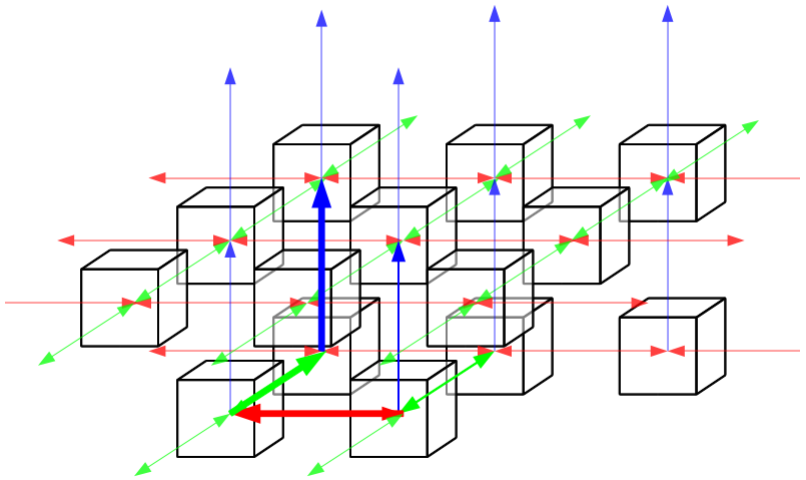
- 全方向直接
  - 簡単
  - 同時通信数の制限 近接だけでも 26

    トーラスでは斜め方向の通信もいずれかの軸方向と競合する

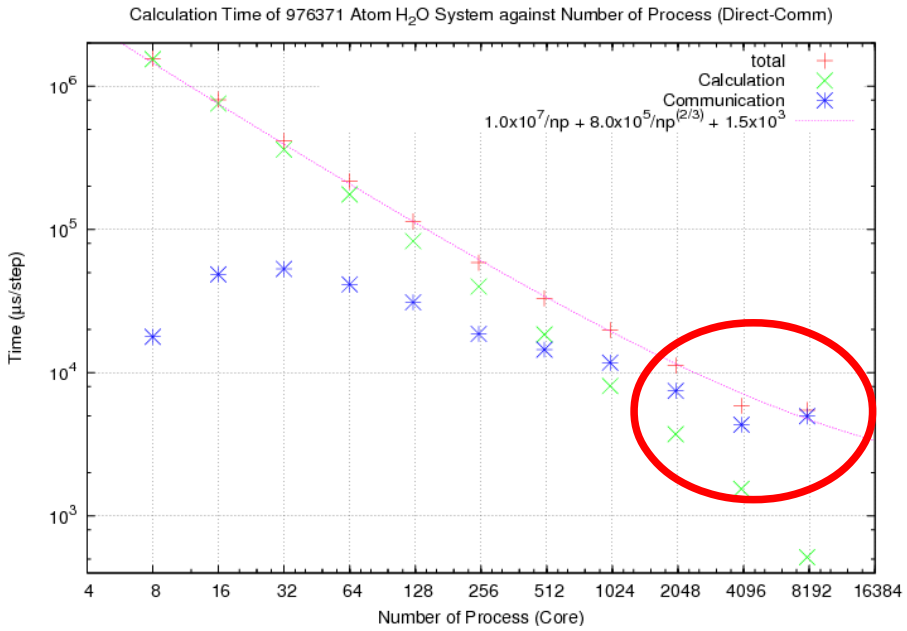


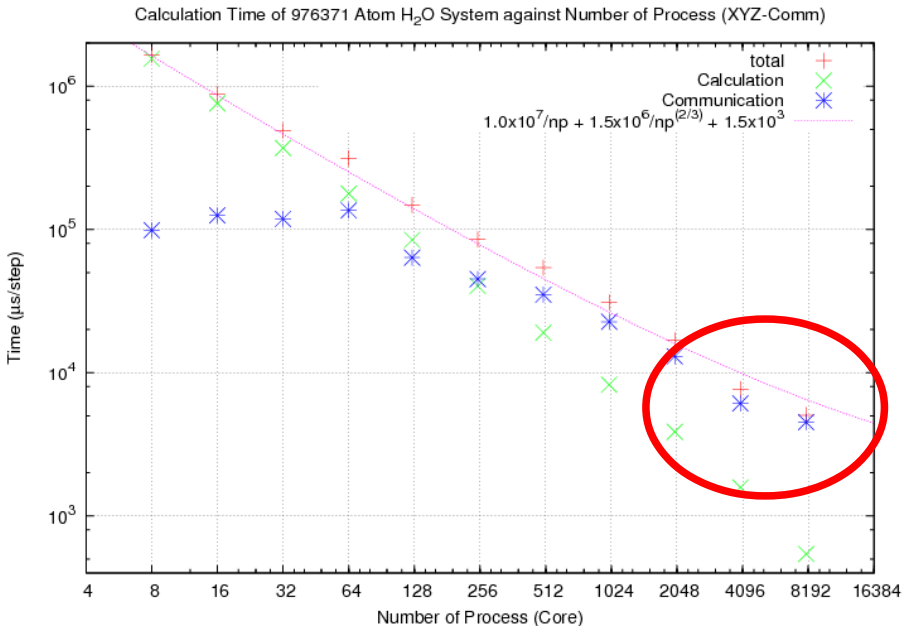
- XYZ
  - 軸方向のみ トーラス向き
  - 斜めは多段

    トーラスでは物理的にも斜めはホップ数が多い



RICC での結果





- 全方向同時
  4000 で増加
- XYZ
  8000 まで減少

分子動力学コードの場合　大規模並列

- カットオフ直接和　局所的･近接通信　　問題なし
- PME　FFT の大域通信が問題
  - TOFU に適した通信パターンではない
- 他の手法へ乗り換え
  - FMM 演算量が多い
  - MultiGrid　ポアソン解法　MD での実績が乏しい
  - 近距離への繰り込み カットオフ距離の制限
    - 一様性、電気的中性等
  - 無視 目的による

ハイブリッド並列化

- フラット MPI
- 単純 (1 階層)
- データ共有は全て通信
  - 記述が面倒
  - 競合しようがない
- ノードに複数プロセス
  - コア数倍

    メモリ、通信も

- OpenMP
- 複雑 (2 階層)
- メモリ共有
  - 記述は簡単
  - 競合しないように記述しなければならない
- ノードに１プロセス

分子動力学コードの場合　カーネルループ

```
for ( i=0; i<ni; i++) {
  for ( j=0; j<nj; j++) {
     dr = rj[j] - ri[i];
     r2 = dr^2;
     if (r2<cutoff^2) {
         fi[i] += f(r2) * dr;
         fj[j] -= f(r2) * dr;
      }
  }
}
```

どこをスレッド化するか

- i loop

  粒度が大きい

  fj[j] への書き込みが競合する
- j loop

  粒度が小さい
  - オーバーヘッド　「京」の場合は低減
  - アンバランス

  競合なし
  - fi は OMP REDUCTION

分子動力学コードの場合　スレッド並列の例

```
#pragma omp parallel for
for ( i=0; i<ni; i++) {  // このループを並列化
  for ( j=0; j<nj; j++) {
     dr = rj[j] - ri[i];
     r2 = dr^2;
     if (r2<cutoff^2) {
        fi[i] += f(r2) * dr;
        fj[j] -= f(r2) * dr;  // 異なるスレッドが同じ j を
      }                          変更する
  }
}     // スレッドの合流は 1 回
```

分子動力学コードの場合　スレッド並列の例

```
for ( i=0; i<ni; i++) {
  fitemp = fi[i];
#pragma omp parallel for reduction (+: fitemp)
  for ( j=0; j<nj; j++) {  // このループを並列化
     dr = rj[j] - ri[i];
     r2 = dr^2;
     if (r2<cutoff^2) {
         fitemp += f(r2) * dr;
         fj[j] -= f(r2) * dr;
      }
  }   // スレッドの合流は ni 回
  fi[i] = fitemp;
}
```

反作用再利用は得か

- fij = - fji であるから、演算そのものはほぼ半減
- j-loop 中は

  1 個の fi に nj 回 足し込む -> レジスタ演算
  - i 並列なら独立、j 並列なら reduction

  nj 個の fj に 足す -> キャッシュ load store
  - i 並列なら競合 スレッド別バッファを用意する等の対策が必要
    L2 キャッシュを圧迫
  - j 並列なら独立
    キャッシュ単位では競合する可能性あり
- fj の演算で性能効率が低下する可能性あり

カットオフの条件分岐

- SIMD 化の障害
  - マスク演算等で SIMD 化できる場合もある。
- 効率が真率依存
  - セルインデックスで立方体単位で選別
  - セルサイズ＝カットオフ距離で約 1/2
- 近接粒子リスト／ペアリスト
  - カーネル効率は上がる
  - リスト作成コスト　リスト作成は間隔をあける

ペアリスト 作成

```
for ( i=0; i<ni; i++) {
  for ( j=0; j<nj; j++) {
     dr = rj[j] - ri[i];
     r2 = dr^2;
     if (r2<cutoff^2) {
       pi[np] = i; ip[i][npi] = np;   //  スレッド競合
       pj[np] = j; jp[j][npj] = np;   //  スレッド競合
       np++;  npi++; npj++;           //  スレッド競合
     }
  }
}
```

i 毎にリストを分ければ i 並列で解消
i 粒子に対する近接粒子リストと同じ

ペアリスト　カーネルループ

```
for ( p=0; p<np; p++) {     //  このループを並列化
    dr = rj[pj[p]] - ri[pi[p]];  //  間接参照
    r2 = dr^2;
    if (r2<cutoff^2) {
      s = 1.0;
    }else{
      s = 0.0;
    }
    fp[p] = s * f(r2) * dr;
  }
}
```

不連続でキャッシュミスしやすい
ここだけ SIMD 化されない

f(r2) の演算が多ければ
プリフェッチで隠蔽できるかも

ペアリスト　後処理

```
for ( i=0; i<ni; i++) {
  for (p=0; p<npi; p++){
    fi[i] += fp[ip[i][p]];  //  単純な実装では連続
  }
}
for ( j=0; j<nj; j++) {
  for (p=0; p<npj; p++){
    fj[j] -= fp[jp[j][p]]; //  単純な実装では不連続
  }
}
```

近接粒子リストで反作用なしなら
このループ自体不要

まとめ

- 「京」では近接通信が有利
  - 「京」に限らない
- 「京」ではスレッド並列が必須
- OpenMP は記述そのものは単純だが、
  アクセス競合を考慮すると面倒
  性能を追求すると大変
- 演算が増えても、メモリアクセスやアクセス競合を減らした方が速い可能性もある。