# ベンチマーク検討結果について

**牧野淳一郎 (国立天文台)**
**戎崎俊一 (理研)**
**安田耕二 (名古屋大学)**
**松原裕樹、中里直人 (理研)**

平成 19 年 5 月 23 日

# 目 次

# 1 結果要約

| コード | 推定性能 (PF) | 備考 |
|---|---|---|
| QCD | 2 | 加速ボード側に専用ネットワークをつけた場合 |
| FD | 5 | 小規模計算の場合 ($2000 \times 1000 \times 1000$ 以下) |
| Multilocas | 3 | 推定精度低い。より詳細な評価が必要 |
| astro | $2.5 \sim 5$ | 上限に近付くにはチューニング必要 |
| GAMESS-FMO | $1.3 \sim 2.8$ | アルゴリズム変更による演算量変化を補正した結果 |
| NICAM_BM | 2.5 | |
| FrontSTR_BM | 効率で 5–20% | 専用ネットワークを使わない場合 |
| simfold | 効率で $> 25\%$ | チューニングの余地あり |
| coevolv | $> 4$ | 並列化手法の変更が必要 |
| ParaMD | 2.9 | |
| RSDFT | 3.9 | |
| 3DRISM | $0.1 \sim 0.5$ | 0.1 は下限。ホスト増強で 0.5 程度までは可能 |

# 2 前提

ここでは、想定したマシン構成についてまとめる。以下について順に説明する

1. 全体構成の概要
2. ホスト計算機のアーキテクチャと構成
3. 加速ボードのアーキテクチャと構成
4. 加速ボード側ネットワーク
5. ホスト計算機側ネットワーク
6. 想定する全体構成案と費用

## 2.1 全体構成

提案システムは、

- ホスト計算機とそのネットワーク
- ホスト計算機 1 ノード毎に装着される加速ボード
- 加速ボード間のネットワーク (オプショナル)

によって構成される。ホスト計算機は基本的には普通の x86 等の COTS システムであり、想定する完成年度において大きな困難なく調達できるものである。ネットワークについても同様で、現在の技術の延長上にある、 Infiniband で 3GB/s 程度の fat tree を想定する。これらは理研次世代スーパーコンピュータ開発実施本部 (以下、実施本部と略して表記) の指定通りである。

加速ボードは、 GRAPE-DR や ClearSpeed CX と類似のワンチップ SIMD アーキテクチャをとるが、量産コストを下げつつ広い範囲の問題で高い性能を実現するために、このどちらとも違う形で大規模なメモリをオンチップ embedded DRAM として実装する。また、ホスト経由では実現困難である高速で低レイテンシなネットワークを、加速ボード側で構成することも検討する。これを実装するかどうかは各アプリケーションベンチマークの性能評価の後で議論する。

## 2.2 ホスト計算機のアーキテクチャと構成

これは、実施本部の指定 (2006/6/8 付け。それ以前のものとは異なる) の通り、1 ノードが以下の構成をもつものとする。

- CPU 演算性能 64 Gflops/chip。実施本部の資料には 8 コア、4 GHz とあるが、4 コア、 4 演算、 4 GHz で達成可能な数字であり 2010 年のものとしては低過ぎるが、これが指定であるのでそれに従う。現実には遅くとも 2008 年にはこの数字は実現される。

- ノード演算速度 128 Gflops。これは 2 チップということである。これも CPU 性能が低めの見積もりなので低過ぎるが、指定なのでやむをえない

- メモリ容量 32 GB。これも現在 (2006 年) 実現できる数字であり少ないが、指定なのでそれに従う。

- CPU チップ・メモリ間バンド幅 Write 12 GB/s, Read 20GB/s。これはチップ毎にこれだけの速度が実現できると仮定する。現在出荷が始まっている Blackford チップセットでメモリへの最大バンド幅は既に 21GB/s に達しており、上の数字は 2010 年のものとしては低いが、指定なのでそれに従う。

- I/O バス幅 8GB/s。これは、PCIe Gen2 16 レーンとなっている。従って、全二重動作で read/write 独立にこの速度がでるものとする。さらに、現在利用可能なディスクトップ用、あるいはサーバ用チップセットでも PCIe 32 レーン (16 レーンを独立に 2 本) 持つものは普通になってきているので、これが 2 本独立にあるものを仮定する。また、メモリはそれだけの速度、つまり、read/write 並行動作でそれぞれ 16GB/s をサポートできるものと仮定する。でなければ 8GB/s の仮定には意味がないためである。

全体に、実施本部の想定は過度に保守的であり、 2010 年時点としても適切ではないように思われるが、指定がそうなっているのでそれに従った性能評価を行う。しかし、より現実的な評価を可能にするため、ホスト性能がボトルネックになる場合についてはホスト側のノード数を変化させた評価も行う。

## 2.3 ホスト計算機側ネットワーク

実施本部の指定は多段ファットツリーで、

- バンド幅 3GB/s
- レイテンシ
  - スイッチ間、スイッチ-NIC 間 100ns
  - CPU-NIC 間 400ns

としている、ファットツリーのスイッチ 1 つのポート数は指定がないが、24 ポートとする。この程度が現状での通常の値であろう。また、ファットツリーは constant bandwidth のもの、つまり、上 12、下 12 のスイッチによる構成とする。ネットワークバンド幅は若干控えめとも思われる。より現実的な評価を可能にするため、ネットワーク性能がボトルネックになる場合についてはバンド幅を変化させた評価も行う。レイテンシについては変化させない。

## 2.4 加速ボードのアーキテクチャと構成

加速ボードのアーキテクチャは、

- PE (Processor Element) が多数集まって放送ブロック (Broadcast Block, BB) を構成
- BB が多数集まってチップを構成
- ボードはチップを複数載せたもの

となる。基本的アーキテクチャについては提案書で既に詳しく述べたので、ここでは提案書のシステムとの違いについてまとめる。

アーキテクチャの基本的な違いは以下の 2 点である。

1. オンボードの外付けメモリを廃し、オンチップメモリのみとした
2. チップ間をメッシュ結合するネットワークも検討する

まず、オンチップメモリについて説明する。

提案書システムではオンボード、オフチップなメモリがあったのに対して今回評価するシステムではこれを排除し、その代わりに BB 毎にもつ放送メモリ (Broadcast memory, BM) を SRAM ではなく embedded DRAM にすることでサイズを大きくしたことである。

BM は以下の仕様を持つものとする。

サイズ: チップ全体で 32MB。BB 毎の大きさはこれを BB 数でわったもの。
PE アレイへのデータ転送幅: 4 語 (1 語は倍精度浮動小数点数)/PE クロック
BM 間転送: 多段Ωネットワーク。クロック毎に 2 語転送可能。
PE との転送と BM 間転送は並列に動作可能

ネットワークについては次節でより詳しくのべるが、チップに対する要求についてここでまとめる。

BM 間ネットワークのノード数を増やす形でホストへの I/O ポートおよびチップ間ネットワークのためのポートを出す。

ホストとの通信のためのバンド幅はチップ毎に 1 語/クロック 双方向とする。

チップ間ネットワークのためのポートは双方向、7 語/クロックとする。

1 枚の加速ボードに何チップのせ、その間の結合と全体の結合をどうするかについてはネットワーク構成と分離しては議論できないので、次で述べる。

チップ単体の諸元は以下の通りとする

- PE 数 484 (22x22)
- BB 数 22
- BB 当りメモリ 1.5MB
- 動作クロック 700MHz
- クロック当り演算数 倍精度加減算+乗算 1 つづつ、または単精度加減算+乗算 2 つづつ。但し、倍精度演算と単精度演算は並列には行えない
- 理論ピーク性能 678Gflops(倍精度)、1.355Tflops(単精度)
- チップ外とのリンクは、それぞれ 5.6GB/s

## 2.5 加速ボード側ネットワーク

加速ボード側では、チップ全体が 2 重化した 3 次元メッシュネットワーク、つまり、$l \times m \times n \times 2 \times 2 \times 2$ の 6 次元ネットワークを構成するようにする。この構成にすることで、システムを分割しても 3 次元トーラスと等価な扱いが可能である。

カード、ラックの実装は 64K チップ ( $32 \times 16 \times 16 \times 2^3$) まで可能な構成とする。これを 512 ラックに収める。このため、ラック当り 128 チップとする。ラック内ネットワークは $16 \times 2^3$ とし、ラック全体は $32 \times 16$ に並べる。 32 本の方向はラック同士を結合し、ネットワーク配線は最短距離で接続する。 16 本の方向はメンテナンススペースを設けるために床下あるいはラック上を通る配線とする。

ラック間の結合は、 128 チップまでフル実装した場合で、横方向リンク 64 本、縦方向リンク 32 本となる。双方向であることを考慮すると、縦方向 (床下配線) はラック当り 64 本、360GB/s のバンド幅となる。3.2 Gbps 程度のシリアル転送とすれば 1000 本 (ラックから出る数は両側あるのでさらに 2 倍だが) となるが、困難というほどの数ではない。

128 チップに対してホスト計算機を 16 ノード用意する。接続は 4 チップ単位とし、2 単位、すなわち 8 チップがホスト 1 ノードに接続されるものとする。

## 2.6 想定する全体構成案と費用

実施本部からの指定は

- ホスト側能力 10Pflops
- 加速ボード側能力 10Pflops

のシステムに対して能力を見積もれ、となっているが、これは上の我々が想定したシステム構成とは本質的に異なっている。ホストの 1 ノード能力は 128 Gflops であり、加速ボード側は 1 チップがそのほぼ 5 倍の能力を持つので、ホストと加速ボードの能力を同じにするためには、加速ボードが 4 チップ載るものとすれば 20 ノードに 1 枚だけ加速ボードをつけること、という要請を実施本部がしていることになる。加速ボード 1 枚の量産コストはホストノード 1 台の量産コストと同じか多少高い程度なので、これはトータルコストの 5% しか加速ボードに使ってはいけない、という要請と等価である。つまり、ほとんど全てのアプリケーションに対して価格性能比が最適点から遠く離れた構成にせよ、といっていることに相当する。

そのような、最適ではない構成で価格性能比を議論することには意味がないので、ここでは実施本部の指定から離れて、上に既に述べた 8 チップ毎に 1 ノードホストがある構成を想定する。ピーク 10 Pflops の時にほぼ 15000 チップとなるので、ネットワークサイズは ( $20^3 \times 2$) とする。 ホストは 2000 ノード、ネットワークは、少し上位でのバンド幅が不足するが例えば $13 \times 13 \times 12$ のファットツリーとする。

この時、全体コストは以下のようになる。

| | | |
|---|---|---|
| 加速チップ開発費 | 25 億 | |
| 加速カード量産費 | 28 億 | (4000 枚、1 枚 70 万と仮定) |
| ホストノードコスト | 24 億 | (2000 ノード、1 ノード 120 万円と仮定) |
| スイッチコスト | 12.5 億 | (500 台、1 ユニット 250 万と仮定) |
| 匡体、電源等 | 5 億程度 | |
| 総計 | 95 億程度 | |
| 総計 (ホスト部分除く) | 58 億程度 | |

# 3 各ベンチマークに対する検討結果

配布された 14 本のうち、 FDM_BM と frontflow は今回入らないとのことなので考慮しない。それ以外の 12 本について以下で順に検討する。

## 3.1 QCD

2010 年に想定される問題サイズは $96 \times 96 \times 96 \times 192$ となっている。これを $20 \times 10 \times 10 \times 8$ のグリッドにマッピングすると、1 ノードは $5 \times 10 \times 10 \times 24$ になる。この時、仮に 100% の効率がでたとしても、プロセッサの 1 部は遊んでいる。このため、理論効率は $(96^3 \times 192)/(96 \times 100^2 \times 192) = 0.885$ を超えることはない。

ソースに添付されている実行ログデータ

```
PetaQCDMultBench/MultBench_v2.63_sse3_64/LOG-32x32x08x16_4x4x1
```

によれば、 $8 \times 8 \times 8 \times 16$ の格子を 1 ノードでもった時の、ベンチマークプログラムでのノード当りの通信総量が 145MB、計算総量が 1553 Mflop となっている。これはプロセッサが 2 次元メッシュであるが、通信量は 3 次元メッシュとして計算されている。

上のデータから、ノード内の 1 次元メッシュ数が 8 の時に、演算当りのデータ通信量は 0.0934 byte/flop であることがわかる。従って、本検討で想定する 1 次元方向のメッシュが 5 の場合は演算量当りではこの 8/5 倍の通信が発生し、約 0.15 byte/flop となる。

なお、現在筑波大学が構築中の PACS-CS システムでは、ハードウェアの理論値で 0.13 byte/flop となり、 QCD 計算に最適化された設計がなされていることがわかる。

以下、チップ間のメッシュネットワークを使った場合と使わない場合の両方について検討する。

### 3.1.1 チップ間メッシュネットワークを使う場合

チップ間メッシュネットワークを使う場合、ハードウェアの byte/flop 値 がいくつになるかで実効性能が予測できる。ハードウェア的な数字は 5.6GB/s 6 方向、678Gflops となるので、 0.05 byte/flop である。これは必要値の 1/3 であるので、理想的に通信リミットな計算ができた場合には約 33% の効率が期待できる。実際には、演算と通信が完全にはオーバーラップしないので、それによる効率低下が 30% とすれば、 23% となる。これにさらに上の 0.885 を掛けると、約 20% となる。従って、ピーク 10Pflops の場合にほぼ 2Pflops の実行速度が期待できる。

ここではチップ間通信のみを考慮し、チップ内でのデータ転送、計算に対する検討は行っていない。BB とチップ間ポートの結合は多段クロスバーとみなせるので、ここで転送速度が不足することはない。また、大きな効率低下があったとしても、そもそもチップ間ネットワークで性能がリミットされているので問題にならない。このため、実行効率が 20% を大きく割り込むことはないと考える。

### 3.1.2 チップ間メッシュネットワークを使わない場合

チップ間メッシュネットワークを使わない場合、ホスト側のネットワークで通信速度がリミットされる。2000 ノードを $5 \times 5 \times 8 \times 10$ のネットワークにした場合、1 次元方向のメッシュは約 20 となるので、演算当りのデータ通信量は 0.0374byte/flop となる。ネットワークは 3GB/s と仮定されているので、計算速度はノード当り 80 Gflops となる。ピーク演算速度は 8 チップで 5.4 Tflops となるので、実行効率は 1.5% になる。

これは、 fat tree という QCD 計算にはむかないネットワークを使った場合の話であり、同じだけのスイッチ物量を使っても 3 次元ハイパークロスバー結合とすればこの 3 倍、約 5% の実行効率を実現できることに注意したい。また、スイッチコストは全体コストの 10% 程度でしかないので、この部分をさらに 2 倍に増強すればさらに 2 倍の効率、すなわち 10% 程度までは実現可能である。10% からさらに効率を上げるためには、ホストと加速ボードの通信速度自体を上げる必要がある。

## 3.2 FD

想定モデルは $10240 \times 5120 \times 5120$ 程度の格子である。格子点毎のデータは

```
REAL  SXX  (NX1,NY,NZP0:NZP1), SYY (NX1,NY,NZP0:NZP1)
REAL  SZZ  (NX1,NY,NZP0:NZP1)
REAL  SXY  (NX1,NY,NZP0:NZP1), SYZ (NX1,NY,NZP0:NZP1)
REAL  SXZ  (NX1,NY,NZP0:NZP1)
REAL  VX   (NX1,NY,NZP0:NZP1), VY  (NX1,NY,NZP0:NZP1)
```

```
REAL  VZ   (NX1,NY,NZP0:NZP1)

REAL  Q    (NX10,NY10,NZP10),  LAM (NX10,NY10,NZP10)
REAL  DENI (NX10,NY10,NZP10),  RIG (NX10,NY10,NZP10)
REAL  ROXA (NX,NY,NZP),  ROYA(NX,NY,NZP)
REAL  ROZA (NX,NY,NZP)
REAL  RMXYA(NX,NY,NZP),  RMXZA(NX,NY,NZP)
REAL  RMYZA(NX,NY,NZP)

REAL  DXVX (NX1,NY,NZP), DXVY (NX1,NY,NZP), DXVZ (NX1,NY,NZP)
REAL  DYVY (NX1,NY,NZP), DYVX (NX1,NY,NZP), DYVZ (NX1,NY,NZP)
REAL  DZVZ (NX1,NY,NZP), DZVX (NX1,NY,NZP), DZVY (NX1,NY,NZP)
```

(DXSXX 等はエリアスなので省略) となり 28 語である。REAL が単精度か倍精度が必要なのかは明らかではないが、ここでは倍精度を仮定する。 1 格子点当りのデータ量は 224 B、トータルのデータ量は 56 TB となる。これは、ホストメモリには収容可能だが、加速ボード側メモリからは溢れる。加速ボード側ではトータルメモリは 512GB であるので、例えば $2000 \times 1000 \times 1000$ の計算なら実行可能である。これは、現在行われている計算サイズに対応する。

以下の順番で見積もりを行う。

1. 加速ボードメモリに入る場合
2. 加速ボードメモリに入らない場合

### 3.2.1 加速ボードメモリに入る場合 ( $2000 \times 1000 \times 1000$ 以下)

この場合、データは BM にあるとしてよい。空間差分の基本ループでは、1 項求めるのに格子点毎に乗算 8、加減算 15 の 23 演算を行う。入力はベクトル 1 本であり、出力も 1 本である。このため、BM のバンド幅からくる制約は、BB1 つでクロックサイクル毎に最大 2 点となる。演算数はこの時 BB 当り加減算 30 となり 想定した PE 数より多い。従って、計算時間は BB の転送時間ではなく加減算の実行時間で決まる。

但し、これは 3 次元のどの方向のアクセスにも BM のデータアクセスがピーク速度がでたとした場合である。実際には、DRAM はページモードアクセスなので、ループ構造を変化させた場合の効率が問題になる。これによるデータアクセスの速度低下は 2 倍程度と見込まれるので、 BM のバンド幅からくる制約は 1 格子点/クロックサイクルとなり、演算数は BB 当り 23 となる。すなわち、実行効率は 50% となる。

ノード間通信は、プロセッサを $40 \times 20 \times 20$ の格子に配置できた理想的な場合を考えると $50^3$ の格子点を 1 ノードに入れることになる。通信は 8 メッシュ先まで起こるので、空間微分 9 項の演算のために速度の 3 成分を転送するのが主な通信となる。1 ステップ当り 50x50x8x3=60K 語 となる。計算量は 50x50x50x23x9=26M 演算となる。この比は 500:1 であり、ハードウェアの演算と転送速度の比である 161 よりも大きい。さらに、演算側の効率は低いので、ノード間通信の必要度はさらに下がり、1000:1 程度になる。すなわち、4 チップのカードに対して 3GW/s 程度、24GB/s 程度、ということになる。ホストとの転送速度はほぼそれと釣り合うが、ホスト間の速度は不足であり、専用カード側のネットワークは有用である。しかし、QCD 計算のような高いバンド幅が必要なわけではない。ホスト側のネットワーク増強によって十分高い性能を実現できる。

### 3.2.2 加速ボードメモリに入らない場合 ( $2000 \times 1000 \times 1000$ を超える)

この場合、もっとも単純にはチップに入る分だけ計算し、中身を全部入れ替えて別のブロックを計算し、というのを繰り返すことになる。この場合、読み込んだデータ当り 23 演算というのは変わらないので、片方向転送速度がホスト当りトータルで 2GW/s とすると、演算速度は 46 Gflops、効率は 1 % 程度となる。

この場合、領域分割して複数ステップを進め、その後で整合性をとるための反復計算を行う、といった手法が可能かもしれないが、そこまで検討の検討は今回与えられた時間では不可能であり、またアプリケーションプログラム開発者共同で差分スキーム自体から検討することも必要となるためそのような可能性は検討していない。このため、今回の検討結果はアプリケーション性能の下限であり、アーキテクチャの適切な評価にはなっていない。

## 3.3 Multilocas

計算量がもっとも多いのは関数 prob2 で、 iswitch=2 で呼ばれる場合である。この時の関数の主要部分を以下に示す

int

```
prob2(int hapid,
      int upperlimit,
      int nhap,
      int haplen,
      double pbin,
      int ng1[]){
```

```
int i,j,k,iflg,icheck,it;
int tmpi,yates;
int localtype1error=0,significant,newupperlimit,acc,ng2;
double pb,chi;
double a,b,c,d,e,f,g,h;

    /* prob2 2nd */
    for(k=1; k<=icount; ++k){
        /* 2nd CALCULATION START */
        for(j=0; j<haplen; ++j){
            yates = -1;
            a = (double)ng1[j];
            b = (double)(2*n[0] - ng1[j]);
            c = (double)xng2[k][j];
            d = (double)(2*n[1] - xng2[k][j]);

            if ((a + b)*(c + d)*(a + c)*(b + d) == 0){
                if (yates >= 0)
                    printf("chi calc impossible %f %f %f %f    %f\n",
                           a,b,c,d,(a+b)*(c+d)*(a+c)*(b+d));
                chi = -1;
                if(chi > CHITHRESHOLD001) {
                    type1error += xpb[k]*pbin;
                    break;
                }
            } else {
                if(yates == 1){
                    chi = (fabs(a*d - b*c) - 0.5*(a + b + c + d))
                        *(fabs(a*d - b*c) - 0.5*(a + b + c + c))
                        *(a + b + c + d)/((a + b)*(c + d)*(a + c)*(b + d));
                } else {
                    f = (a*d - b*c);
                    g = f*(a*d - b*c);
                    h = g*(a + b + c + d);
                    e = (a + b)*(c + d)*(a + c)*(b + d);
                    chi = h/e;
                    if (chi>=0 && chi<10000)
                        ;
```

```
                    else {
                        printf("chi=%f %f %f %f %f f=%f g=%f h=%f e=%f\n",
                               chi,a,b,c,d,f,g,h,e);
                    }
                }

                if(chi > CHITHRESHOLD001) {
                    type1error += xpb[k]*pbin;
                    break;
                }
            }
        }
        sumprob += xpb[k]*pbin;
        /* 2nd CALCULATION END */
    }

}
```

演算内容は、ng1, xng2 という 2 つの配列 (ng1 は 1 次元、 xng2 は 2 次元) に対して、 xng2 の要素数のオーダーの計算をする、というものである。 xng2 は非常に多数回の prob2 呼び出しに対して変化しない。 ng1 は変化する。 ng1 は hap, x から生成される。 hap, x は小さな配列である。 x は生成可能である。

加速ボードでの実装としては、単純には ng1 を多数ホストで生成し、各プロセッサエレメントにロードし、 xng2 を放送しながら計算する、という手順になる。これでは ng1 の転送に時間がかかりすぎるならば、 PE 側で ng1 を並列に生成することも可能である。従って、通信時間は問題にならず、計算時間だけで効率を評価してよい。

本質的な演算は以下の部分である。

```
            if ((a + b)*(c + d)*(a + c)*(b + d) == 0){
                if (yates >= 0)
                    printf("chi calc impossible %f %f %f %f    %f\n",
                           a,b,c,d,(a+b)*(c+d)*(a+c)*(b+d));
                chi = -1;
                if(chi > CHITHRESHOLD001) {
                    type1error += xpb[k]*pbin;
                    break;
                }
```

```
} else {
    if(yates == 1){
        chi = (fabs(a*d - b*c) - 0.5*(a + b + c + d))
            *(fabs(a*d - b*c) - 0.5*(a + b + c + c))
            *(a + b + c + d)/((a + b)*(c + d)*(a + c)*(b + d));
    } else {
        f = (a*d - b*c);
        g = f*(a*d - b*c);
        h = g*(a + b + c + d);
        e = (a + b)*(c + d)*(a + c)*(b + d);
        chi = h/e;
        if (chi>=0 && chi<10000)
            ;
        else {
            printf("chi=%f %f %f %f %f f=%f g=%f h=%f e=%f\n",
                   chi,a,b,c,d,f,g,h,e);
        }
    }

    if(chi > CHITHRESHOLD001) {
        type1error += xpb[k]*pbin;
        break;
    }
}
```

これは、基本的には 3 つの if 文による条件実行ブロックで、どれか 1 つが実行される。演算量は 2 つのケースでは 20 前後である。 SIMD マシンの性格上、このような分岐は条件実行となるので効率はほぼ 1/2 に低下する。ベクトル化によってパイプラインを埋めることは十分にできるが、加算・乗算の並列実行は困難なケースもある。これらを考慮すると、演算部分の効率は 30% 前後となろう。 すなわち、10Pflops のマシンで 3Pflops 程度が期待できる。

以上の検討は予備的なものであり、精密な性能評価には計算法のアーキテクチャへの最適化を含めたより詳細な検討が必要である。

## 3.4 astro

### 3.4.1 惑星形成コード

想定する粒子数は 100 万程度である。この時、1 ステップで積分される粒子数は 2,000 程度と考えて良い。このコードでつかっているエルミート積分公式の場合、粒子間相互作用の演算数は約 60 である。 GRAPE-DR のコードでは現状では効率は 55% 程度となっている。これはまだ最適化されていないものであり、また提案システムでは単精度演算を強化するので 80% 程度の効率が得られると考えているが、ここではとりあえず 55% を仮定する。1 ステップ当りの重力計算の時間は理想的なロードバランスが実現されていたとして 21 マイクロ秒となる。

ホストと GRAPE の間の通信は、各ノードで $2000/\sqrt{2000} = 50$ 個程度の粒子を送って力を回収することになる。粒子当りのデータ量は 100 バイト程度なので、50 粒子を 8GB/s で転送する必要時間は 0.6 マイクロ秒となる。実際には、拡張バスのプロトコルのオーバーヘッドや、チップセットのレイテンシがあるので 1.5 マイクロ秒程度を仮定する。

このコードでは 1 ステップ当り 4 回のノード間の集団通信が発生する。集団通信といっても、システム全体に渡るものではなく、ノードを仮想的に 2 次元グリッドに配置した時に、縦方向または横方向である。

- 最小値
- 全対全の放送 (MPI_ALLGATHERV 相当) 2 回
- 総和 (MPI_ALLREDUCE 相当)

である。転送量は最小値は 16 バイト、それ以外は粒子当り 48 バイト程度である。

まず、ホスト側のネットワークで通信した場合について検討する。対数ステップで通信できたと仮定すれば、それぞれ 5 ステージ程度になる。1 ステージのネットワーク遅延は平均 4 ホップと仮定すると 1.2 マイクロ秒である。カットスルーで転送するという楽観的な仮定をすると、1 度の転送時間は 6-7 マイクロ秒、ストアアンドフォワードの場合は、最小値以外 3 回は 12 マイクロ秒となる。ホスト計算機での計算時間はほぼ無視できるので、トータルの 1 ステップ当りの計算時間は 49.5-64.5 マイクロ秒となり、実効性能は 2.4-1.9 Pflops となる。

集団通信を加速ボード側のネットワークで行うことを考えると、レイテンシが現在の SCI なみの 30ns とし、1 次元リングトポロジで放送、総和を行ったとして 50 ステージであり、ホストとの転送オーバーヘッドを考慮しても 2.5-3 マイクロ秒で通信が終了する。この時、1 ステップ当りの計算時間は 34 マイクロ秒となり、実効性能は 3.5 Pflops となる。

ここでは、重力計算自体の効率に 55% というかなり保守的な仮定をし、さらに計算と通信をオーバーラップしてレイテンシを隠す可能性も考慮していないので、推

定性能は下限である。チューニングの程度によっては実効性能が倍程度になる可能もある。

### 3.4.2 銀河形成コード

ベンチマークデータでは、重力相互作用の相互作用リストの長さは 21600 程度、流体部分は 100 程度となっており 2 桁程度の違いがある。相互作用あたりの計算量は流体部分のほうが若干多いが、相互作用の数が少ないのでトータルの計算量の観点からは流体部分は無視してよい。しかし、以下に述べるように通信時間にな無視できない寄与があるので、その部分は評価する必要がある。

重力相互作用の部分は、計算量の大半は 2500 個程度の粒子への 20000 個程度の粒子からの重力を求める計算となる。この部分を加速ボードで行い、そのための相互作用リストやツリー構造を作る、あるい領域分割や、粒子データの通信といった準備的な計算はホスト側で行うものとする。これは、現在 GRAPE を使ったツリーコード、あるいは使わないものでも並列化し、重力計算部分をチューニングしたツリーコードでは広く行われている標準的な方法である。

全体の粒子数はテストデータの数百倍、テストデータは 3M 粒子となっているので、簡単のため 300 倍の 1G 粒子を想定する。ノード数は 2000 なので、1 ノードが 50 万粒子を扱うことになる。この程度粒子数がある場合、よほどノード間通信が遅くなければ通信時間は無視できる程度になるとわかっている。3GB/s はホスト計算速度の 128Gflops, 加速ボード側の 5.6 Tflops に比べて十分以上に速いため、以下ではノード間通信は無視する。

現在の GRAPE-DR のシミュレータで得られている重力計算コードの効率は 60% 程度である。提案システムでは特に単精度演算を強化するためにもう少し高い効率が期待できるが、ここではその効果は考慮しないで 60% の効率であるとする。1 ノード、1 ステップの重力計算の演算量は $5 \times 10^5 \times 2 \times 10^4 \times 38 = 3.8 \times 10^{11}$ であり、これを理論ピーク 5.6 Tflops, 効率 60% で実行すると 0.11 秒となる。ツリー構築等は、現在の典型的は CPU (Athlon 64 2GHz) の場合で 100 万粒子でほぼ 1 秒であるので、単純に浮動小数点演算性能から外挿すると 0.04 秒となる。ホストと加速ボード間の通信は、グループサイズが 2500 と仮定すると、ノード当り 400 万粒子を送る部分が殆どの時間を占めるのでこの部分だけを考慮する。1 粒子のデータは 40 バイト前後となるので、データ総量は 160M バイトとなる。通信速度は 8GB/s と仮定されているので、ピークに近い性能が出れば 0.02 秒で終わることになる。ピークの 1/2 程度でも計算時間の 20% となって大きな部分を占めるわけではない。ここでは 0.04 秒を仮定する。

以上から、重力の部分については 0.38Tflop を 0.11+0.04+0.04 = 0.19 秒で実行することになる。 この部分の実効性能はノード当り 2.0 Tflops、全体システムとしては 4.0 Pflops となる。

SPH 部分については、計算時間は加速ボード側で実行するなら無視できるので、加速ボードとホストの通信時間だけを考えればよい。これは 2 ステップになり、1 粒子のデータ量も多いので重力計算の場合の 5 倍程度になるが、粒子数が 1/3 になるためにほぼ 1.5 倍である。つまり、 0.06 秒程度増加することになる。つまり、SPH 部分も考慮した場合の計算時間はステップ当り 0.25 秒、実効性能は 1.52 Tflops/3.04 Pflops (ノード当り/全体) となる。

### 3.4.3 総合性能

専用チップ間ネットワークがある場合、銀河形成、惑星形成の平均は 3.2 Pflops、ない場合では 2.5 Pflops となる。但し、チューニングを進めると 1.5-2 倍程度向上する余地がある。

## 3.5 GAMESS-FMO

### 3.5.1 想定モデルについて

「ベンチマークコードの概要」(「概要」) には、対象問題サイズは不定とある。そこで「Fock 行列作成に関するベンチマークプログラムについて」(「説明書」) の最後にある「数万〜数 10 万原子クラスの巨大分子を FMO 法で計算する」事を念頭に置く。FMO 法は蛋白質の電子状態計算に使われており、想定モデルを 2 千から 2 万残基の蛋白質とする。FMO では普通これを 2 残基の unit に分割する (unit 数は $10^3 \sim 10^4$ )。そして monomer 計算では各 unit の Hartree-Fock (HF) MO 計算を数 10 回、dimer 計算では unit pair の HFMO 計算を 1 回行う。

我々の提案システムは 2000 ノードなので、各ノードが 1 unit の monomer 計算や 1 unit pair の dimer 計算を並列に行うと仮定する。ベンチマークコードは FMO 計算を含まないので、各ノードのロードバランスやノード間通信は今回評価しない。各ノードが行う、2 又は 4 残基ペプチドの HFMO 計算を評価する。この場合 Fock 行列生成つまり 2 電子積分生成が計算時間の殆どを占めると思われる。そこで我々は、提案システムの 2 電子積分計算での性能を検討する。想定する基底関数は、ベンチマークコードで利用可能な STO-3G, 6-31G, 6-31G*, 6-31G**とする。

### 3.5.2 性能評価の方法

ベンチマークコードでは、2 電子積分は OS や HGP 漸化式で計算し、buffered direct SCF 法で Coulomb 項と exchange 項を計算している。2 電子積分計算法には幾種類かあり、それらは計算量、メモリー使用量が違うが、同じ積分値を与える。つまりユーザーから見てこれらは等価である。「アルゴリズムの改変は認めない」のが実施

本部の方針であるが、限られた時間でベンチマークコードを加速ボードに移植する事は不可能であり、労力に見合った価値が無い。そこで我々は、同じ積分値を与えるが、ベンチマークコードとは異なる 2 電子積分計算法を検討する。公平のため提案システムの性能は、減点法で評価する。つまり提案アーキテクチャの制限による、理想値からの速度低下を考慮する。2 電子積分計算公式には、代表的な場合の理論浮動小数点演算回数が与えられているので、最速アルゴリズムと比較可能である。

より現実的な比較のため、想定基底関数を使い、taxol ( $C_{47}H_{51}NO_{14}$) と valinomycin ( $C_{54}H_{90}N_6O_{18}$) での計算時間を測定する（テスト分子と呼ぶ）。これはアミノ酸 4-5 残基程度の大きさである。x86 processor 上で Gaussian03 を実行し、種々の条件での 2 電子積分計算時間の比を測定し、結果の換算や比較に用いる。

### 3.5.3 評価する部分

短縮 Gauss 基底の 4 つ組が与えられた時、それらの間の電子間反発積分を計算し、これに密度行列をかけ積算し、Coulomb 項と交換項を作る事が目的である（「説明書」参照）。細粒度の並列性があり、計算量が多く、通信量が少ない。加速チップで並列計算する際の問題点は、各 PE のローカルメモリーが 256 words と少ないため、高速のアルゴリズムが実装できるか、という点に絞られる。想定モデルの殆どの原子は HCNOFNa-Cl で、想定基底関数は primitive d 型、短縮長 $K = 3, 2, 1$ の sp 型、短縮長 $K = 6, 3, 2, 1$ の s 型となる。そこでこれらの 2 電子積分の実装法と予測性能を重点的に調べる。金属酵素は僅かに遷移金属元素を含み、6-31G**基底では分極関数として f 関数を含む。またより高精度で高価な計算を行う時、g 関数等の高角運動量の基底関数を使う可能性もある。これらは巨大分子の一部分にのみ使われると思われ、重要度は低いが、念のため評価する。

### 3.5.4 加速チップ上での Coulomb 項の計算法

加速チップ上では、2 電子積分を計算し Fock 行列に変換することのみ行い、ホストでその準備をする。その手順を説明する。

SCF 計算は incremental Fock build を使い、Schwartz cutoff と density-based cutoff を使う。Coulomb 項と交換項を別々に計算する。Coulomb 項には、Hermite Gauss 基底と Rys 求積法を用いた direct J 法を使う。交換項は、Saunders 法か DRK 法に、座標回転を組み合わせる。長距離 Coulomb 項を fast multipole method (FMM) で計算可能にする。

これらは代表的電子状態プログラム Gaussian03 で使われている標準的な方法である。DRK 法は電子状態プログラム Hondo で使われている。これらの cutoff や FMM により Coulomb 項に必要な 2 電子積分総数が $1/20 \sim 1/100$ に減るので、京速計算機上では使用すべきである。

Coulomb 項の計算は、次の手順で行われる。2 電子積分とほぼ同じ計算法で、実は 1 電子積分も計算できる事に注意。従って遠方原子を点電荷で近似した場合の、静電場の 1 電子項も計算可能である。つまり summary sheet にある「Elapsed time for H-core matrix」のかなりの部分も加速ボードで計算可能である。

1. まずホスト上で 1 電子積分、密度行列の初期値、2 電子積分カットオフ、FMM の計算を行う。これには殆ど時間がかからない（「説明書」の例では 1%程度)。又 FMM の計算時間は短距離 Coulomb 項の計算時間の 1/20 程度である（テスト分子での実測値)。

2. ホスト上で計算すべき短縮 Gauss 基底の shell pair (p,P) を作る。Schwartz cutoff と density-based cutoff を使い、shell pair を組み合わせ、4 つ組のリストを作る。この時 22 個の P shell pairs に対し、cutoff より大きな積分値を与える Q shell pairs をまとめる。zero padding が必要で、2 電子積分の計算時間は 2 ～ 5 割増える（テスト分子での実測値)。

3. 加速チップに 4 つ組リストと密度行列値を送る。加速チップ内で各 PE は、自分担当の P shell pair と放送された Q shell pair に対して、2 電子積分を並列に計算する。これに放送された密度行列値 D をかけ、Fock 行列に加算する。全ての Q shell pairs を送ったら、積算した Fock 行列を、結果回収ネットワークで加算しつつ、ホストに回収する。

テスト分子では、計算すべき積分数は $10^8$ 程度はあり、加速チップの $8 \times 22^2 = 3872$ 個の全 PE にタスク分割が可能である。

次に各 PE 上での direct J の実装法を説明する。問題となるのは、256 words という各 PE のローカルメモリー容量だけだが、Rys 法ではこれで十分である。関数の補間表などは全 PE で共通なので、オンチップメモリー (BM) に置く。必要な LM 領域の最小値を示す。

| | Fock | Hermite 多項式 | Q shell 数 | その他 | 計 |
|---|---|---|---|---|---|
| $(pp\|pp)$ | 10 | 4 次 52 個 | 4 | 7 個 | 73 |
| $(dd\|dd)$ | 35 | 8 次 100 個 | 4 | 7 個 | 146 |
| $(ff\|ff)$ | 64 | 12 次 148 個 | 4 | 7 個 | 219 |
| $(gg\|gg)$ | 145 | 16 次 49 個 | 1 | 7 個 | 201 |

$(ss|ss) \sim (ff|ff)$ までは 4 つの Q shell pair を同時に扱い、PE パイプラインを埋める事が可能である。提案システムと類似した GRSPE-DR プロセッサ用に作成したプログラムでは、命令スロットの 7 割程度が埋まっている。他方 $(gg|gg)$ では命令並べ替えでパイプラインを埋める必要が有り、効率は最悪 1/4 となる。

2 電子積分計算では、演算量に比べて通信量はかなり少ない。放送すべき Q shell pair のデータは, $(ss|ss)$ 型積分では 5 words, $(dd|dd)$ 型では 39 words である。他方積分計算には逆数や誤差関数を含み、 $(ss|ss)$ 型で数十演算, $(dd|dd)$ では一般に 2

万〜3 万演算が必要な事が知られている。以上をまとめると、各 PE 上での計算効率は、少なくとも他の一般的な CPU と同じと思われる。

現在の実装では、積分の電子 1 と 2 に関する対称性は使わず、primitive Gauss 関数のみサポートする。積分対称性は、incremantal Fock build を使うため、重要度は高くないと思われるが、現在プログラムを改良中である。Zero padding と primitive Gauss 関数を使うため、Coulomb 項の演算回数は最小値の 1.5 ∼ 2.5 倍となる（テスト分子での実測値）が、SIMD プロセッサの特性上この程度の増加はやむを得ない。

### 3.5.5 加速チップ上での交換項の計算法

この場合も問題点は、各 PE の使える作業領域が、高速アルゴリズムの実装に十分か、という点に絞られる。作業領域には、局所メモリー（256 words）とオンチップメモリー（BM, 1PE あたり約 70000 words）がある。オンチップメモリーとの通信速度は 4 words/cycle なので、これは平均アクセス速度が 6 cycle/word の遅いメモリーと見なせる。これだけの作業領域があればほぼ全ての 2 電子積分計算アルゴリズムは実装可能である。つまり各 PE での計算効率の下限値は、読み 6 cycle, 演算 1 cycle, 書き 6 cycle と考えると $1/13 = 8$ %となる。

交換項では primitive Gaussian を用いると、計算量が 2.4 ∼ 6 倍になる（テスト分子での実測値)。他方 primitive Gaussian を用いた積分計算法では作業領域が少なくて済み、オンチップメモリーとの通信時間が少ない。そこで我々は、基底関数の型と短縮度に応じて、最適なアルゴリズムで計算する。

まず 4 個の基底関数のうち s 型を 2 個以上含む場合、中間積分は局所メモリーに格納可能で、短縮基底関数に対する、普通の DRK 法が使える。他方 d 型関数を多数含む場合、想定基底関数では d 型 Gauss 関数は primitive としてのみ含まれるから、Gauss 関数を uncontract して計算する。この時局所座標系を使うと、Saunders 法や DRK 法を 2 倍高速化、省メモリー化できる。これは現在プログラム中である。

### 3.5.6 まとめ

Coulomb 項は f 関数までなら効率 70%, g 関数は最悪 25%, 交換項は s 型を 2 個以上含む場合最悪 25%, それ以外は最悪 8%程度と思われる。テスト分子での実測値とベンチマークサンプル出力から考えると、Coulomb 項と交換項の積分計算時間は 1 : 1 ∼ 1 : 2 程度、さらに交換項のうち 40%は s 型を 2 個以上含む。これに zero padding による積分数の増加を考えると、理論性能の 13 ∼ 28%で計算できると思われる。システム全体では 1.3 ∼ 2.8 PFLOPS となる。交換項の計算速度は下限値であり、プログラム最適化でかなり加速可能と思われる。

## 3.6 NICAM_BM

資料では、計算時間がもっともかかっているのは以下の差分計算ルーチンである

```
subroutine src_flux_convergence (&
       rhogvx, rhogvx_pl,        & !--- IN : rho*Vx  ( gam2 X G^{1/2} )
       rhogvy, rhogvy_pl,        & !--- IN : rho*Vy  ( gam2 X G^{1/2} )
       rhogvz, rhogvz_pl,        & !--- IN : rho*Vz  ( gam2 X G^{1/2} )
       rhogw,  rhogw_pl,         & !--- IN : rho*w   ( gam2 X G^{1/2} )
       grhog,  grhog_pl,         & !--- OUT: source
       f_type )                    !--- IN : flux type
  !------
  !------ Flux convergence calculation
  !------     1. Horizontal flux convergence is calculated by using
  !------        rhovx, rhovy, and rhovz which are defined at cell
  !------        center (verticaly) and A-grid (horizontally).
  !------     2. Vertical flux convergence is calculated by using
  !------        rhovx, rhovy, rhovz, and rhow.
  !------     3. rhovx, rhovy, and rhovz can be replaced by
  !------        rhovx*h, rhovy*h, and rhovz*h, respectively.
  !------     4. f_type can be set as below.
  !------     5. Calculation region of grho, grho_pl
  !------             : (:, ADM_kmin:ADM_kmax,:)
  !
  use mod_adm, only :  &
       ADM_gall,       &
       ADM_kall,       &
       ADM_lall,       &
       ADM_GALL_PL,    &
       ADM_LALL_PL,    &
       ADM_kmin,       &
       ADM_kmax,       &
       ADM_prc_me,     &
       ADM_prc_pl
  use mod_vmtr, only : &
       VMTR_RGSGAM2,   &
       VMTR_RGSGAM2_pl,&
       VMTR_GZXH,      &
       VMTR_GZXH_pl,   &
```

```
     VMTR_GZYH,      &
     VMTR_GZYH_pl,   &
     VMTR_GZZH,      &
     VMTR_GZZH_pl,   &
     VMTR_GSGAMH,    &
     VMTR_GSGAMH_pl, &
     VMTR_RGSH,      &
     VMTR_RGSH_pl,   &
     VMTR_RGAM,      &
     VMTR_RGAM_pl
use mod_grd, only :  &
     GRD_gzh,        &
     GRD_gz,         &
     GRD_afac,       &
     GRD_bfac
use mod_oprt, only :        &
     OPRT_divergence
use mod_perf, only : &                  ! 06/05/23 NEC R.Ogata [add]
     t_flag,                           & ! 06/05/23 NEC R.Ogata [add]
     t_src_flux_convergence              ! 06/05/23 NEC R.Ogata [add]
!
implicit none
!
real(8), intent(in) :: rhogvx(ADM_gall,ADM_kall,ADM_lall)
real(8), intent(in) :: rhogvx_pl(ADM_GALL_PL,ADM_kall,ADM_LALL_PL)
real(8), intent(in) :: rhogvy(ADM_gall,ADM_kall,ADM_lall)
real(8), intent(in) :: rhogvy_pl(ADM_GALL_PL,ADM_kall,ADM_LALL_PL)
real(8), intent(in) :: rhogvz(ADM_gall,ADM_kall,ADM_lall)
real(8), intent(in) :: rhogvz_pl(ADM_GALL_PL,ADM_kall,ADM_LALL_PL)
real(8), intent(in) :: rhogw(ADM_gall,ADM_kall,ADM_lall)
real(8), intent(in) :: rhogw_pl(ADM_GALL_PL,ADM_kall,ADM_LALL_PL)
!
real(8), intent(out) :: grhog(ADM_gall,ADM_kall,ADM_lall)
real(8), intent(out) :: grhog_pl(ADM_GALL_PL,ADM_kall,ADM_LALL_PL)
!
character(LEN=*),intent(in) :: f_type
!<-----                     = 'HORIZONTAL' : horizontal convergence
!<-----                     = 'VERTICAL'   : vertical convergence
```

```
!<-----                      = 'DEFAULT'    : both of them
!
real(8) :: flx_vz(ADM_gall,ADM_kall,ADM_lall)
real(8) :: flx_vz_pl(ADM_GALL_PL,ADM_kall,ADM_LALL_PL)
!
real(8) :: hdiv(ADM_gall,ADM_kall,ADM_lall)
real(8) :: hdiv_pl(ADM_GALL_PL,ADM_kall,ADM_LALL_PL)
real(8) :: rhogvx_f(ADM_gall,ADM_kall,ADM_lall)
real(8) :: rhogvx_f_pl(ADM_GALL_PL,ADM_kall,ADM_LALL_PL)
real(8) :: rhogvy_f(ADM_gall,ADM_kall,ADM_lall)
real(8) :: rhogvy_f_pl(ADM_GALL_PL,ADM_kall,ADM_LALL_PL)
real(8) :: rhogvz_f(ADM_gall,ADM_kall,ADM_lall)
real(8) :: rhogvz_f_pl(ADM_GALL_PL,ADM_kall,ADM_LALL_PL)
!
integer :: l,k
!
t_start=mpi_wtime()       ! 06/05/23 NEC R.Ogata [add]
!
!--- horizontal contribution to flx_vz
if(f_type=='VERTICAL') then
   flx_vz(:,:,:) = 0.0D0
   flx_vz_pl(:,:,:) = 0.0D0
else !--- default or 'HORIZONTAL'
   do l=1,ADM_lall
      do k = ADM_kmin, ADM_kmax+1
         flx_vz(:,k,l) = (                                              &
              ( GRD_afac(k)*VMTR_RGSGAM2(:,k  ,l)*rhogvx(:,k  ,  l)   &
              +GRD_bfac(k)*VMTR_RGSGAM2(:,k-1,l)*rhogvx(:,k-1,  l) )  &
              * 0.5D0*VMTR_GSGAMH(:,k,l)*VMTR_GZXH(:,k,l)             &
              +( GRD_afac(k)*VMTR_RGSGAM2(:,k  ,l)*rhogvy(:,k  ,  l)  &
              +GRD_bfac(k)*VMTR_RGSGAM2(:,k-1,l)*rhogvy(:,k-1,  l) )  &
              * 0.5D0*VMTR_GSGAMH(:,k,l)*VMTR_GZYH(:,k,l)             &
              +( GRD_afac(k)*VMTR_RGSGAM2(:,k  ,l)*rhogvz(:,k  ,  l)  &
              +GRD_bfac(k)*VMTR_RGSGAM2(:,k-1,l)*rhogvz(:,k-1,  l) )  &
              * 0.5D0*VMTR_GSGAMH(:,k,l)*VMTR_GZZH(:,k,l)             &
              )
      end do
   end do
```

```
   if(ADM_prc_me==ADM_prc_pl) then
      do l=1,ADM_LALL_PL
         do k = ADM_kmin, ADM_kmax+1
            flx_vz_pl(:,k,l) = (                                                    &
                 ( GRD_afac(k)*VMTR_RGSGAM2_pl(:,k  ,l)*rhogvx_pl(:,k  ,  l)   &
                 +GRD_bfac(k)*VMTR_RGSGAM2_pl(:,k-1,l)*rhogvx_pl(:,k-1,  l) )  &
                 * 0.5D0*VMTR_GSGAMH_pl(:,k,l)*VMTR_GZXH_pl(:,k,l)             &
                 + ( GRD_afac(k)*VMTR_RGSGAM2_pl(:,k  ,l)*rhogvy_pl(:,k  ,  l) &
                 +GRD_bfac(k)*VMTR_RGSGAM2_pl(:,k-1,l)*rhogvy_pl(:,k-1,  l) )  &
                 * 0.5D0*VMTR_GSGAMH_pl(:,k,l)*VMTR_GZYH_pl(:,k,l)             &
                 +( GRD_afac(k)*VMTR_RGSGAM2_pl(:,k  ,l)*rhogvz_pl(:,k  ,  l)  &
                 +GRD_bfac(k)*VMTR_RGSGAM2_pl(:,k-1,l)*rhogvz_pl(:,k-1,  l) )  &
                 * 0.5D0*VMTR_GSGAMH_pl(:,k,l)*VMTR_GZZH_pl(:,k,l)             &
                 )
         end do
      end do
   end if
end if
!
!--- vertical contribution to flx_vz
if(f_type/='HORIZONTAL') then
   do l=1,ADM_lall
      do k = ADM_kmin, ADM_kmax+1
         flx_vz(:,k,l)                                      &
              = flx_vz(:,k,l)                               &
              + rhogw(:,k,l) * VMTR_RGSH(:,k,l)
      end do
   end do
   if(ADM_prc_me==ADM_prc_pl) then
      do l=1,ADM_LALL_PL
         do k = ADM_kmin, ADM_kmax+1
            flx_vz_pl(:,k,l)                                &
              = flx_vz_pl(:,k,l)                          &
              + rhogw_pl(:,k,l) * VMTR_RGSH_pl(:,k,l)
         end do
      end do
   end if
end if
```

```
!
!--- vertical flux is zero at the top and bottom boundary.
flx_vz(:,ADM_kmin,:)   = 0.0D0
flx_vz(:,ADM_kmax+1,:) = 0.0D0
if(ADM_prc_me==ADM_prc_pl) then
   flx_vz_pl(:,ADM_kmin,:)   = 0.0D0
   flx_vz_pl(:,ADM_kmax+1,:) = 0.0D0
end if
!
!--- Horizontal flux convergence
if(f_type/='VERTICAL') then
   hdiv(:,:,:) = 0.0D0
   hdiv_pl(:,:,:) = 0.0D0
   !
   rhogvx_f(:,:,:) = rhogvx(:,:,:)*VMTR_RGAM(:,:,:)
   rhogvy_f(:,:,:) = rhogvy(:,:,:)*VMTR_RGAM(:,:,:)
   rhogvz_f(:,:,:) = rhogvz(:,:,:)*VMTR_RGAM(:,:,:)
   !
   if(ADM_prc_me==ADM_prc_pl) then
      rhogvx_f_pl(:,:,:) = rhogvx_pl(:,:,:)*VMTR_RGAM_pl(:,:,:)
      rhogvy_f_pl(:,:,:) = rhogvy_pl(:,:,:)*VMTR_RGAM_pl(:,:,:)
      rhogvz_f_pl(:,:,:) = rhogvz_pl(:,:,:)*VMTR_RGAM_pl(:,:,:)
   end if

   call OPRT_divergence(      &
        hdiv,hdiv_pl,         &  !--- out
        rhogvx_f, rhogvx_f_pl,&  !--- in
        rhogvy_f, rhogvy_f_pl,&  !--- in
        rhogvz_f, rhogvz_f_pl &  !--- in
        )
else
   hdiv(:,:,:) = 0.0D0
   hdiv_pl(:,:,:) = 0.0D0
end if
!
!--- Total flux convergence
do l=1,ADM_lall
   do k = ADM_kmin, ADM_kmax
```

```
         grhog(:,k,l)                                      &
              = - ( flx_vz(:,k+1,l) - flx_vz(:,k,l) ) &
                / ( GRD_gzh(k+1) - GRD_gzh(k) )       &
                - hdiv(:,k,l)
      end do
   end do
   grhog(:,ADM_kmin-1,:) = 0.0D0
   grhog(:,ADM_kmax+1,:) = 0.0D0
   !
   if(ADM_prc_me==ADM_prc_pl) then
      do l=1,ADM_LALL_PL
         do k = ADM_kmin, ADM_kmax
            grhog_pl(:,k,l)                                       &
                 = - ( flx_vz_pl(:,k+1,l) - flx_vz_pl(:,k,l) ) &
                 / ( GRD_gzh(k+1) - GRD_gzh(k) )               &
                 - hdiv_pl(:,k,l)
         end do
      end do
      grhog_pl(:,ADM_kmin-1,:) = 0.0D0
      grhog_pl(:,ADM_kmax+1,:) = 0.0D0
   end if
   !
   t_end=mpi_wtime()                       ! 06/05/23 NEC R.Ogata [add]
   if(t_flag/=0)then                       ! 06/05/23 NEC R.Ogata [add]
   t_src_flux_convergence=                &! 06/05/23 NEC R.Ogata [add]
   t_src_flux_convergence+(t_end-t_start) ! 06/05/23 NEC R.Ogata [add]
   endif                                   ! 06/05/23 NEC R.Ogata [add]
   !
 end subroutine src_flux_convergence
```

そのうち、計算時間のほとんどは、以下の 2 つの 3 重ループ (最内側は配列演算形式) が占める。

```
      do l=1,ADM_lall
         do k = ADM_kmin, ADM_kmax+1
            flx_vz(:,k,l) = (                                          &
                 ( GRD_afac(k)*VMTR_RGSGAM2(:,k  ,l)*rhogvx(:,k  ,  l)  &
                 +GRD_bfac(k)*VMTR_RGSGAM2(:,k-1,l)*rhogvx(:,k-1,  l) )  &
```

```
            * 0.5D0*VMTR_GSGAMH(:,k,l)*VMTR_GZXH(:,k,l)             &
            +( GRD_afac(k)*VMTR_RGSGAM2(:,k  ,l)*rhogvy(:,k  ,  l)  &
            +GRD_bfac(k)*VMTR_RGSGAM2(:,k-1,l)*rhogvy(:,k-1,  l) )  &
            * 0.5D0*VMTR_GSGAMH(:,k,l)*VMTR_GZYH(:,k,l)             &
            +( GRD_afac(k)*VMTR_RGSGAM2(:,k  ,l)*rhogvz(:,k  ,  l)  &
            +GRD_bfac(k)*VMTR_RGSGAM2(:,k-1,l)*rhogvz(:,k-1,  l) )  &
            * 0.5D0*VMTR_GSGAMH(:,k,l)*VMTR_GZZH(:,k,l)             &
            )
      end do
   end do

   do l=1,ADM_LALL_PL
      do k = ADM_kmin, ADM_kmax+1
         flx_vz_pl(:,k,l) = (                                          &
            ( GRD_afac(k)*VMTR_RGSGAM2_pl(:,k  ,l)*rhogvx_pl(:,k  ,  l)   &
            +GRD_bfac(k)*VMTR_RGSGAM2_pl(:,k-1,l)*rhogvx_pl(:,k-1,  l) )  &
            * 0.5D0*VMTR_GSGAMH_pl(:,k,l)*VMTR_GZXH_pl(:,k,l)             &
            + ( GRD_afac(k)*VMTR_RGSGAM2_pl(:,k  ,l)*rhogvy_pl(:,k  ,  l) &
            +GRD_bfac(k)*VMTR_RGSGAM2_pl(:,k-1,l)*rhogvy_pl(:,k-1,  l) )  &
            * 0.5D0*VMTR_GSGAMH_pl(:,k,l)*VMTR_GZYH_pl(:,k,l)             &
            +( GRD_afac(k)*VMTR_RGSGAM2_pl(:,k  ,l)*rhogvz_pl(:,k  ,  l)  &
            +GRD_bfac(k)*VMTR_RGSGAM2_pl(:,k-1,l)*rhogvz_pl(:,k-1,  l) )  &
            * 0.5D0*VMTR_GSGAMH_pl(:,k,l)*VMTR_GZZH_pl(:,k,l)             &
            )
      end do
   end do
```

必要メモリ量はベンチマークコードで 1 GB となっている。これがメッシュ数に比例して増えると仮定すると、 2010 年の glevel=14、 垂直メッシュ 100 での計算ではこの 655360 倍、約 650 TB のメモリが必要となる。このため、ホスト計算機 1 台当り 32 GB メモリという実施本部の指定に従うと、 2 万台のホストが必要になる。ここではそれだけのホストがあり、ホスト 1 台に 1 チップの加算ボードがついているものとして検討を行う。

別資料[1]からスケーリングすると、 2010 年時点での計算量は 1 ステップ当り 2 Pflop となる。加速ボードで実行することを考えると、最低限 1 ステップに全データ

[1] http://www.gfd-dennou.org/arch/gfdsemi/2005-09-12/tomita/pub-web/065.html.ja

が 1 度加速ボードにいって、戻ってくる必要がある。加速ボードとホスト間の全通信バンド幅は 160 TB/s (双方向) となるので、650 GB のデータをやりとりするには 4 秒が必要となり、実効性能は 500 Tflops、実行効率は 5% となる。

但し、NICAM は水平方向は 2 次精度の陽解法であるので、1 ステップで隣のメッシュまでしか情報がつたわらない。このことを考慮してアルゴリズムを変更しないで計算順序の入れ替えを行うと、バッファゾーンを設けることで複数ステップの計算を進めることができる。トータルの水平方向 1 次元メッシュ数は数万のオーダーになるので、数十ステップ分程度のバッファゾーンを設けても計算量の増加は無視できる。 この場合、分散メモリの並列ベクトル機とほぼ同等の実行効率が期待できるので、上記資料に従って効率を 25% 程度と推測する。すなわち、実行速度は 2.5 Pflops 程度となる。

## 3.7 FrontSTR_BM

計算量の大半が
hecmw/src/solver/solver_33/hecmw_solver_CG_33.f
にあるサブルーチン hecmw_solve_CG_33 で消費される
計算時間は主に SSOR 部分と推測されるので、その部分についてのみ検討する。

```
!C-- FORWARD

        do i= 1, N
          SW1= WW(3*i-2,indexA)
          SW2= WW(3*i-1,indexA)
          SW3= WW(3*i  ,indexA)
          isL= INL(i-1)+1
          ieL= INL(i)
         do j= isL, ieL
              k= IAL(j)
             X1= WW(3*k-2,indexA)
             X2= WW(3*k-1,indexA)
             X3= WW(3*k  ,indexA)
            SW1= SW1 - AL(9*j-8)*X1 - AL(9*j-7)*X2 - AL(9*j-6)*X3
            SW2= SW2 - AL(9*j-5)*X1 - AL(9*j-4)*X2 - AL(9*j-3)*X3
            SW3= SW3 - AL(9*j-2)*X1 - AL(9*j-1)*X2 - AL(9*j  )*X3
         enddo

          X1= SW1
```

```
          X2= SW2
          X3= SW3
          X2= X2 - ALU(9*i-5)*X1
          X3= X3 - ALU(9*i-2)*X1 - ALU(9*i-1)*X2
          X3= ALU(9*i  )*  X3
          X2= ALU(9*i-4)*( X2 - ALU(9*i-3)*X3 )
          X1= ALU(9*i-8)*( X1 - ALU(9*i-6)*X3 - ALU(9*i-7)*X2)
          WW(3*i-2,indexA)= X1
          WW(3*i-1,indexA)= X2
          WW(3*i  ,indexA)= X3
        enddo

!C
!C-- BACKWARD

        do i= N, 1, -1
          isU= INU(i-1) + 1
          ieU= INU(i)
          SW1= 0.d0
          SW2= 0.d0
          SW3= 0.d0
          do j= isU, ieU
              k= IAU(j)
             X1= WW(3*k-2,indexA)
             X2= WW(3*k-1,indexA)
             X3= WW(3*k  ,indexA)
            SW1= SW1 + AU(9*j-8)*X1 + AU(9*j-7)*X2 + AU(9*j-6)*X3
            SW2= SW2 + AU(9*j-5)*X1 + AU(9*j-4)*X2 + AU(9*j-3)*X3
            SW3= SW3 + AU(9*j-2)*X1 + AU(9*j-1)*X2 + AU(9*j  )*X3
          enddo
          X1= SW1
          X2= SW2
          X3= SW3
          X2= X2 - ALU(9*i-5)*X1
          X3= X3 - ALU(9*i-2)*X1 - ALU(9*i-1)*X2
          X3= ALU(9*i  )*  X3
          X2= ALU(9*i-4)*( X2 - ALU(9*i-3)*X3 )
          X1= ALU(9*i-8)*( X1 - ALU(9*i-6)*X3 - ALU(9*i-7)*X2)
```

```
      WW(3*i-2,indexA)=  WW(3*i-2,indexA) - X1
      WW(3*i-1,indexA)=  WW(3*i-1,indexA) - X2
      WW(3*i  ,indexA)=  WW(3*i  ,indexA) - X3
    enddo
```

計算量のほとんどは以下の 2 つのループである。

```
    do j= isL, ieL
         k= IAL(j)
        X1= WW(3*k-2,indexA)
        X2= WW(3*k-1,indexA)
        X3= WW(3*k  ,indexA)
       SW1= SW1 - AL(9*j-8)*X1 - AL(9*j-7)*X2 - AL(9*j-6)*X3
       SW2= SW2 - AL(9*j-5)*X1 - AL(9*j-4)*X2 - AL(9*j-3)*X3
       SW3= SW3 - AL(9*j-2)*X1 - AL(9*j-1)*X2 - AL(9*j  )*X3
     enddo

     do j= isU, ieU
         k= IAU(j)
        X1= WW(3*k-2,indexA)
        X2= WW(3*k-1,indexA)
        X3= WW(3*k  ,indexA)
       SW1= SW1 + AU(9*j-8)*X1 + AU(9*j-7)*X2 + AU(9*j-6)*X3
       SW2= SW2 + AU(9*j-5)*X1 + AU(9*j-4)*X2 + AU(9*j-3)*X3
       SW3= SW3 + AU(9*j-2)*X1 + AU(9*j-1)*X2 + AU(9*j  )*X3
     enddo
```

領域分割した後での SSOR 前処理は、全格子点に対して並列実行可能であるが、現状のコードは非構造格子向けとなっており、上のループ構造からわかるように間接アクセスが主体となる。加速ボードでこの部分の演算を効率的に行うには、データ構造を部分的に構造格子であるような形で表現しなおす必要がある。メッシュ生成のところでは多くの場合に細分化されたメッシュは構造格子となるので、そのようなデータ構造の変更は可能だが、今回のベンチマークの対象外とも考えられる。

従って、以下は、そのような変更を行った場合についての大雑把な見積もりである。ベンチマーク評価としては対象外かもしれないが、参考データとしての検討結果である。

まず、問題サイズは極めて小さいことに注意する。格子点数は $10^7$ 程度とされており、メモリ量は 200GB 程度になる。従って、格子点当り 20kB 程度、倍精度として 2500 語程度、となる。これは格子点当りの自由度の数からはいかにも多い。増

えている理由の一部は、間接アクセスの連続化のためにワーク配列等を用意していることによると思われる。(上のコード断片で AU 等)

従って、部分的に構造格子にした場合には、メモリ必要量は大きく減少し、例えば 10GB 程度となるであろう。このように、問題サイズが極めて小さいため、10Pflops のマシン全体を使って解析することはいかなるアルゴリズムを使っても困難であるので、必要メモリが加速ボード側のオンチップメモリでまかなえる 40 ノード (全体の 1/50, 200 Tflops) のマシンでの性能を検討する。

規則格子で全てメモリにのっている場合には、上の SSOR 前処理は積和演算の形になっているのでほぼピークに近い性能がでる。資料の $40^3$ メッシュの場合から計算時間をスケールすると、反復当りが 512 倍である。計算量は資料のケースで 6.8Gflop であるので 2010 年の問題規模では 2.9 Tflop となる。実際に 200Tflops のピークに近い性能がでたとすると、 15ms で計算終了となり、反復は 300 回なので 200us で反復 1 回が終了する。通信は、グローバルオペレーションが 3 回である。これらに 10us 程度かかるとしても 15% 程度である。近傍通信は、1 ノードが $50 \times 50 \times 100$ 程度の領域を扱うので最大持つデータ量の 1/20 程度が転送される。加速ボード側ネットワークを使った場合には転送時間は無視できるが、そうでない場合にはノード当り、反復当り 12MB の通信が発生する可能性があり、これには 4ms かかるのでこの場合の実行効率は 5% 程度となる。

但し、この場合、ホストノードの数を増やすことで実効効率の向上が可能である。例えば、 1 チップ当り 1 ノード程度まで増やした場合、1 次元方向のメッシュサイズが半分になるので通信量は 1/4 になる、実効効率は 4 倍の 20% 程度まで向上することが期待できる。

## 3.8 simfold

ベンチマーク対象になっているサブルーチンのうち、計算量の 90%以上は PE_hydro6 と PE_HB5_Born がしめる。この 2 つのサブルーチンは制御構造が同じなので、ここでは hydro6 だけについて検討する。

以下はサブルーチンの全体ソースである。

```
      subroutine PE_hydro6(lunout,naa,
     *           Lnsite0aa,Lid0aa,xyz_uar,
     *           HP_energy)
      implicit real (a-h,o-z), integer (i-n)

c-----------------------------------------------------------------------
c
c     Compute hydrophobic interactions when iHPtype = 6.
```

```
c     This is a 20 letter version of energy of burial.
c
c-----------------------------------------------------------------------
      include 'Maxsize.f'

      common/V_HP_5/HP_rho_min,HP_SHP_linear,HP_SHP_S
      common/V_HP_6/HP6_epsilon0id(21),HP6_con0id(21),HP6_consat0id(21),
     *              HP6_sigma_min(21,21),HP6_sigma_max(21,21)
c---- arrays to be prepared -------------------------------------------
      real    xyz_uar(4,naa,3)
      integer Lnsite0aa(naa)
      integer Lid0aa(naa)

c---- temporary arrays
      real    rho(2,mxaa)          ! # of coordination for HP-interaction site
      real    SHP_A      ! degree of being buried of alpha carbons
      real    SHP_B      ! degree of being buried of residues

      parameter(pi=3.14159265e0)

c---- set up rho (local density) for CA and sidechain
      do iaa=1,naa
         rho(1,iaa)=0.0
         rho(2,iaa)=0.0
      enddo

      do iaa=1,naa-1
       igly=4-Lnsite0aa(iaa)
       do isite=1,2-igly
         iid=Lid0aa(iaa)-1000
         if(isite.eq.1) iid=21
         xi=xyz_uar(2*isite,iaa,1)
 yi=xyz_uar(2*isite,iaa,2)
 zi=xyz_uar(2*isite,iaa,3)
         do jaa=iaa,naa
          jgly=4-Lnsite0aa(jaa)
          do jsite=1,2-jgly
           if(iaa.ne.jaa .or. isite.lt.jsite) then
```

```
        jid=Lid0aa(jaa)-1000
        if(jsite.eq.1) jid=21
        xj=xyz_uar(2*jsite,jaa,1)
yj=xyz_uar(2*jsite,jaa,2)
zj=xyz_uar(2*jsite,jaa,3)

        xij=xi-xj
yij=yi-yj
zij=zi-zj
rij2=xij*xij+yij*yij+zij*zij
        rij=SQRT(rij2)

        rij_min=HP6_sigma_min(iid,jid)
        rij_max=HP6_sigma_max(iid,jid)
        if(rij.gt.rij_min.and.rij.lt.rij_max) then
          pioverwidth=pi/(rij_max-rij_min)
          uHP=0.5*(1+COS(pioverwidth*(rij-rij_min)))
          rho(isite,iaa)=rho(isite,iaa)+ HP6_con0id(jid)*uHP
        else if(rij.lt.rij_min) then
          rho(isite,iaa)=rho(isite,iaa)+ HP6_con0id(jid)
        endif

        rji=rij
        rji_min=HP6_sigma_min(jid,iid)
        rji_max=HP6_sigma_max(jid,iid)
        if(rji.gt.rji_min.and.rji.lt.rji_max) then
          pioverwidth=pi/(rji_max-rji_min)
          uHP=0.5*(1+COS(pioverwidth*(rji-rji_min)))
          rho(jsite,jaa)=rho(jsite,jaa)+ HP6_con0id(iid)*uHP
        else if(rji.lt.rji_min) then
          rho(jsite,jaa)=rho(jsite,jaa)+ HP6_con0id(iid)
        endif

       endif
      enddo  !over jsite
     enddo   !over jaa
   enddo     !over isite
  enddo      !over iaa
```

```
c&&&&
c      write(6,*) (rho(2,i),i=1,naa)
c      write(6,*) '****'
c      write(6,*) (HP6_consat0id(Lid0aa(iaa)-1000),iaa=1,naa)
c      stop
c---- normalize rho
      do iaa=1,naa
         rho(1,iaa)=rho(1,iaa)/HP6_consat0id(21)
         igly=4-Lnsite0aa(iaa)
         if(igly.eq.0)
     *     rho(2,iaa)=rho(2,iaa)/HP6_consat0id(Lid0aa(iaa)-1000)
      enddo

c---- make SHP_A, SHP_B, S_HB and compute HP interaction

      HP_energy=0.e0
      pioverwidthR=pi/(1.0-HP_rho_min)

      do iaa=1,naa
c---- second for SHP_A
        if(rho(1,iaa).gt.1.e0) then
          SHP_A=1.e0
        elseif(rho(1,iaa).lt.HP_rho_min) then
          SHP_A=HP_SHP_linear*rho(1,iaa)
        else
          SHP_A=HP_SHP_linear*rho(1,iaa)
     *         +HP_SHP_S*0.5*(1+COS(pioverwidthR*(1-rho(1,iaa))))
        endif

c---- third, for SHP_B
        SHP_B=0.
        if(rho(2,iaa).gt.1.e0) then
          SHP_B=1.e0
        elseif(rho(2,iaa).lt.HP_rho_min) then
          SHP_B=HP_SHP_linear*rho(2,iaa)
        else
          SHP_B=HP_SHP_linear*rho(2,iaa)
```

```
     *         +HP_SHP_S*0.5*(1+COS(pioverwidthR*(1-rho(2,iaa))))
        endif

c---- compute HP interaction
        HP_energy=HP_energy-HP6_epsilon0id(21)*SHP_A
     *                     -HP6_epsilon0id(Lid0aa(iaa)-1000)*SHP_B

c&&&&
c        write(6,*) 'SHPs',SHP_A,SHP_B

      enddo
c      write(6,*) 'HP_energy=',HP_energy
c      stop

      return
      end
```

計算量は、以下の形式的には 4 重ループになっているものが殆ど全部を占める。

```
      do iaa=1,naa-1
       igly=4-Lnsite0aa(iaa)
       do isite=1,2-igly
         iid=Lid0aa(iaa)-1000
         if(isite.eq.1) iid=21
         xi=xyz_uar(2*isite,iaa,1)
 yi=xyz_uar(2*isite,iaa,2)
 zi=xyz_uar(2*isite,iaa,3)
         do jaa=iaa,naa
          jgly=4-Lnsite0aa(jaa)
          do jsite=1,2-jgly
           if(iaa.ne.jaa .or. isite.lt.jsite) then
            jid=Lid0aa(jaa)-1000
            if(jsite.eq.1) jid=21
            xj=xyz_uar(2*jsite,jaa,1)
    yj=xyz_uar(2*jsite,jaa,2)
    zj=xyz_uar(2*jsite,jaa,3)

           xij=xi-xj
```

```
yij=yi-yj
zij=zi-zj
rij2=xij*xij+yij*yij+zij*zij
        rij=SQRT(rij2)

        rij_min=HP6_sigma_min(iid,jid)
        rij_max=HP6_sigma_max(iid,jid)
        if(rij.gt.rij_min.and.rij.lt.rij_max) then
          pioverwidth=pi/(rij_max-rij_min)
          uHP=0.5*(1+COS(pioverwidth*(rij-rij_min)))
          rho(isite,iaa)=rho(isite,iaa)+ HP6_con0id(jid)*uHP
        else if(rij.lt.rij_min) then
          rho(isite,iaa)=rho(isite,iaa)+ HP6_con0id(jid)
        endif

        rji=rij
        rji_min=HP6_sigma_min(jid,iid)
        rji_max=HP6_sigma_max(jid,iid)
        if(rji.gt.rji_min.and.rji.lt.rji_max) then
          pioverwidth=pi/(rji_max-rji_min)
          uHP=0.5*(1+COS(pioverwidth*(rji-rji_min)))
          rho(jsite,jaa)=rho(jsite,jaa)+ HP6_con0id(iid)*uHP
        else if(rji.lt.rji_min) then
          rho(jsite,jaa)=rho(jsite,jaa)+ HP6_con0id(iid)
        endif

       endif
      enddo  !over jsite
     enddo   !over jaa
   enddo     !over isite
  enddo      !over iaa
```

実際の計算は

```
       xij=xi-xj
yij=yi-yj
zij=zi-zj
rij2=xij*xij+yij*yij+zij*zij
        rij=SQRT(rij2)
```

```
rij_min=HP6_sigma_min(iid,jid)
rij_max=HP6_sigma_max(iid,jid)
if(rij.gt.rij_min.and.rij.lt.rij_max) then
  pioverwidth=pi/(rij_max-rij_min)
  uHP=0.5*(1+COS(pioverwidth*(rij-rij_min)))
  rho(isite,iaa)=rho(isite,iaa)+ HP6_con0id(jid)*uHP
else if(rij.lt.rij_min) then
  rho(isite,iaa)=rho(isite,iaa)+ HP6_con0id(jid)
endif
```

(j から i の部分だけ示す) であり、カットオフがある 2 体相互作用のポテンシャルのようなものの積算になっている。

ノード間並列は考える必要がない、とのことなので、このサブルーチンで行っている計算を加速ボードで実行することを考える。 isite, jsite は 1 または 2 なので、 ホストから加速ボードに送るデータ量は分子数 499 の大規模データで 3000 語、 24KB である。回収量は少ないため、送信速度を考えればよい。ボード 1 枚 で 8GB/s とすると 3 マイクロ秒で送信できる。 DMA 起動のオーバーヘッド等を考えると最大 5 マイクロ秒程度を見込むべきであろう。この間には 2.8 Tflops とすると $10^7$ 演算程度が実行できる。これに対して、ペア相互作用 1 つ当りの演算量は 30 程度である。相互作用の全計算量は、ペアの相互作用を全て計算したとすると $1000^2/2 = 5 \times 10^5$ 相互作用であり、演算時間のほうが長い。ハードウェアで実行する場合は全部計算すると考えてよいので、演算時間が性能をリミットすることになる。

加速ボードでの実行効率は、対称性を利用しないことで 2 倍、 isite 等による条件分岐を条件実行で行うことによる実行時間の増加を 50% 程度、さらに機械語自体の実行効率を 70% 程度とすると、全体効率は 25% となる。従って、理論ピーク性能の 25% 程度が期待できることになる。

なお、最新の資料 (6/27 付け) によると、大規模計算の場合のこの関数呼び出し当りの演算数は $2 \times 10^7$ となっている。対称性を使わないコーディングの場合、提案システムでは 1 PE で 1 相互作用の計算には 30 クロックサイクル程度必要になる。2048 PE では $1.47 \times 10^4$ クロックサイクルになり、 700MHz クロックなので 21 マイクロ秒になる。通信を 5 マイクロ秒とすると、この関数は 26 マイクロ秒で終わることになる。演算数から計算した効率は 27% である。

ベンチマークの実行時間は、計算量の多い 2 つだけを考慮するとどちらも 27 マイクロ秒であり、それらを 736,533 回呼ぶので 38.3 秒となる。

## 3.9 coevolv

このアプリケーションベンチマークでは、S-system と呼ばれる連立常微分方程式を CVODE パッケージを使って解く、というものを多数並列に行う。自由度は、想定する大規模問題で 900 程度である。但し、1 回の試行では、全変数の連立系の数値解を求めるわけではなく、1 つの変数以外は実験値、あるいは前の反復解のテーブルから補間したものを用い、それを外場として 1 つの変数について解く形になる。これを方程式の係数を変えて最適化する。

従って、異なる変数について解く計算は全て独立に実行でき、さらに遺伝的アルゴリズム等で複数のモデルに対しても独立に計算できる。このため、全体としてはいわゆる EP タイプの計算になっている。また、1 つの自由度に対する微分方程式の右辺には、他の項が全て現れるので、1 つの自由度に対する計算量は全体の自由度数に比例することになりかなり多い。

しかし、単純に、各自由度に対する積分を独立に行うだけでは、メモリ階層が極度に深い提案システムの加速ボードのような計算機では高い性能を得ることが困難である。これは、計算のほとんどが右辺値を評価するための他の変数の解の補間と、右辺値自体の計算になり、補間のための解のテーブルの読み出しのためのメモリアクセスに比べて演算数が少ないためである。

但し、この問題は、並列化の方法の変更により回避できる。2 つの自由度に対する最適化計算を並列に行うことを考える。この 2 つ以外の 898 自由度に対して与えるテーブルは共通なので、2 つの自由度の積分をスケジューリングして行い、テーブルアクセスを 1 度ですますことにすればメモリアクセスは 1/2 になる。これを 900 変数の全てに対して行えば、メモリアクセスは 1/900 になることになる。これは、変数毎の部分問題に対する時間積分を同じノード内で並列に行うことになり、並列化の手法が現在のソースプログラムとは異なる。しかし、これは処理全体からみればループ順序の交換程度の話であり、アルゴリズムの変更というには値しないであろう。

この変更を行った場合、ノード間の並列化は GA の試行間の並列化になる。これは現行のソースでは考慮されていないが、特に困難はないと思われる。但し、利用可能なノード数等にはより詳細な検討が必要である。ここでは、従って、ノード間並列化は考慮しない。

この場合、右辺値の計算だけを加速ボードで行い、時間積分はホスト側で行うことが考えられる。 右辺値の計算は、既に述べたように補間と関数評価であり、補間は元のソースコードでは以下の関数が行っている。

```
double SPLINE::Interpolation(double x)
{
  int klo,khi,k;
  double h,b,a,ret;
```

```
    //*********************************
    if (initflag == 0) init();

    //*********************************
    klo = 0;
    khi = NumberOfData - 1;
    while (khi - klo > 1)
    {
      k = (khi + klo) >> 1;
      if (Xa[k] > x) khi = k;
      else klo = k;
    }
    h = Xa[khi] - Xa[klo];
    if (h == 0.0) error();
    a = (Xa[khi] - x)/h;
    b = (x - Xa[klo])/h;
    ret = a*Ya[klo] + b*Ya[khi]
         + ((a*a*a - a)*y2a[klo] + (b*b*b - b)*y2a[khi])*(h*h)/6.0;
    return ret;
  }
```

関数評価のうち、自由度数に比例してコストが高い部分は上の補間結果の対数をとる部分である。それ以外は無視してよい。

900 変数の場合の実行プロファイルでは、補間部分のほうが関数評価よりもコストがかかっている。 対数計算のコストは高いのに、補間部分に計算時間がかかっているのは以下の 2 つの理由による。

1. テーブル参照に毎回 bisection search を行っていて、テーブルサイズの対数程度のコストがかかっている。

2. 補間は 3 次多項式によるものであるが、スプライン多項式の定義式通りの演算順序で計算しており演算数が 加減算 7、 乗算 10、除算 3 となっている。あらかじめ多項式の係数を計算しておけば、3 次多項式の計算量は乗算 3、加算 3 にすぎない。

実施本部の指定は「アルゴリズムを変更しないこと」である。数学的に等価な式変更はアルゴリズムの変更にあたるかどうかは微妙な問題であるが、ソースプログラムのままで演算量を評価することには以下の 2 つの問題がある。

1. 不要な bisection search に時間がかかるので何を評価しているかわからない。この bisection search は容易に取り除けるので、 2010 年時点での性能評価としては適切ではない

2. 不要な演算の存在によってメモリアクセス負荷が相対的にさがっているので、メモリ性能に対して演算性能が高いアンバランスな機械が有利になる

まず、 bisection search については、時間積分するので少なくともサーチの下限を前のステップで使った値にするべきであり、殆ど確実にそこから線形サーチをしたほうが bisection より速い。線形サーチでは平均的には $O(1)$ の計算量になるからである。補間演算については、除算は通常のソースレベル最適化によって 1 回にできるが本来不要であり、他の演算もほぼ 1/3 まで容易に減らすことができる。

対数関数の演算数は要求精度と実装に強く依存する。倍精度が必要なら区分多項式では困難であり、例えば multiplicative normalization 系のアルゴリズムを使うことになる。この場合の演算数は 40 程度である。補間演算とあわせると、微分方程式の右辺値の計算は 1 自由度当たり 50 程度になる。単純に、全自由度が同じ時間刻みで積分されるとすると、1 ステップ当りの計算量は $900^2 \times 50 = 4 \times 10^7$ 演算となり、カードが 5.6 Tflops のピーク性能がでた時に 7 マイクロ秒である。これに対して、通信はテーブルデータ、数値解の両方を考慮しても $900 \times 2 \times 8 = 14$KB 程度であり、16GB/s のバンド幅では 1 マイクロ秒で転送が終わるので通信は 15% 程度である。。相互作用計算には実際には理論ピークはでない。しかし、V-GRAPE アーキテクチャでは効率が 50% 以下になる可能性は低いので、ここでは 50% とする。通信による効率低下は 10% 程度、またホストでの時間積分の実行による性能低下をさらに 10% とすると、総合的な効率は 40% 程度になり、理論ピークの 4 割程度の性能がでることが期待できる。

ノード間並列化ができるものと仮定すれば、総合的な実効性能は 4Pflops となる。

但し、これは自由度間でタイムステップ分布に大きな違いがないとした場合である。 900 遺伝子の場合の実計算データがないのでタイムステップ分布推定は困難である。しかし、実験データからの再構成という手法上、極度に短いタイムスケールは発生しにくいと考えられる。従って、ここではタイムステップは一様に近いと仮定した結果のみとする。

## 3.10 ParaMD

2010 年頃のモデルとして提案されている water7-tree を対象に考える。重いルーチンとして ENERGY_DIRECT,ENERGY_TREE,RATTLE,RATTLE2 の４つが提示されているが RATTLE と RATTLE2 は最も重いルーチン ENERGY_TREE に比べて 1%に満たないのでホストで計算し、ENERGY_TREE と ENERGY_DIRECT ルーチンを加速ボードで高速化する。その他の計算はすべてホストで計算する。

計算法は本質的には 8 重極までを考慮する tree 法 (MD の業界では Cell Multipole Method と呼ばれる) である。通常の実装では、セルの分割レベルを、最低次のセルでは粒子が数個になる程度、このデータの場合 6 ないし 7 レベルにとる。この時、直接計算の部分の計算量はほぼ無視でき、ツリー部分だけを考えればよい。 1 粒子当りの計算コストは、相互作用の数が 189*7 で 約 1300 であり、相互作用当りの計算量は 90 程度となっているので、粒子当り $1.2 \times 10^5$ 演算、全体で $1.2 \times 10^{12}$ 演算となる。ここで、189 はツリーのレベル当りで 1 粒子が相互作用するセルの数であり、CMM のこの実装では幾何学的に数が決まり、 $6^3 - 3^3 = 189$ となる。7 はツリーのレベルの数である。 これから SR-11000 の演算速度を逆算すると 250 Mflops 程度となり少し遅過ぎる気がするが、ソースコードから判断すると主記憶アクセスによって性能がリミットされているのでこの程度かもしれない。

ここでは、加速ボードで CMM を実装する際には標準的な方法である、セルの分割レベルを減らすことで直接部分の計算量を増やし、通信を減らすやり方で計算時間を評価し、それで上の演算数を割ることで実効性能を評価する。演算数が増えた結果を補正した方法になっていることに注意して欲しい。

理論的には、全体システムの演算速度と通信速度からは、最低レベルのセルの中の粒子数が 100 前後になる場合が最適となる。この時の直接部分の計算時間は粒子数、1 粒子が相互作用する相手粒子の数、1 相互作用当りの必要サイクル数をシステム全体での 1 秒当りのマシンサイクル数で割ったものになる。それぞれ $10^7$, $100 \times 27$, 25, $5 \times 10^{15}$ となる。ここで、相互作用数がセル当りの粒子数の 27 倍になっているのは、隣接したい 27 セルの粒子全てと直接計算すると仮定したからである。これは、コードのチューニングで減らすことができるが、ここではその可能性は考慮しない。

従って、計算時間は $10^7 \times 100 \times 27 \times 25/(5 \times 10^{15})$ であり、 135 マイクロ秒となる。通信時間は、ツリー部分については $13 \times 8 \times 189 \times 6 \times 10^7/(10^2 \times 2.56 \times 10^{14})$ となり、46 マイクロ秒である。ここで、13 は 8 重極のデータ語数、8 はワード当りのバイト数、 189 は前と同じ 1 レベル当りの相互作用数、6 はレベル数、 $10^7$ は粒子数である。100 個程度の粒子がツリーデータを共有するので、データ量は 1 つの相互作用リストのデータ量に粒子数/100 をかけたものになる。最後の $2.56 \times 10^{14}$ はシステム全体としてのホストと加速ボードの間のデータ転送速度である。 実際には、最低レベル以外のレベルについてはそれを共有する粒子グループの数を増やすことができる (Barnes の新しいアルゴリズム) ので、通信量はほぼ 1 レベル分だけまで減らすことが可能であるが、ここではその可能性は考慮しない。

直接部分の通信時間は、同様に $27 \times 32 \times 10^7/(2.56 \times 10^{14})$ であり、 30 マイクロ秒である。ここで 32 は 1 粒子当りのデータ量 (バイト) である。また、ツリー部分の計算時間は効率を 70% として 170 マイクロ秒となり、合計は 381 マイクロ秒となる。

ノード間通信は、セルを領域分割した場合 1 ノードに 6000 個程度がはいることになり、約 2 万粒子を隣接ノードから受け取る必要がある。 ノード間通信がボード・

ホスト間通信の 10 倍程度遅いことを考慮すると、これは直接部分の通信とほぼ同じ時間を必要とするので、これを余裕をみて 40 マイクロ秒とすればトータル 420 マイクロ秒となる。従って、通信を考慮した演算速度は、 2.9 Pflops となる。実際のハードウェアの計算速度はほぼこの 2 倍であるが、これは通信を減らすために直接部分の計算量を増やしたからである。2.9 Pflops はこの増加分を補正した結果になっている。

なお、通信、演算量ともに比較的単純なチューニングでかなり減らすことが可能であるので、上の 2.9Pflops という数字はかなり保守的な見積もりとなっている。

## 3.11 RSDFT

### 3.11.1 ベンチマークコードの妥当性

Si 原子 46656 個が指定された想定モデルである。この大きさでは Linear scaling (LS) アルゴリズムを使う方が圧倒的に速い。(例えば Shimoji et.al., Comp. Phys. Comm., 140, 303 (2001) 参照)。ベンチマークコードはこの論文の RS algorithm に相当する。この論文の図 4 と 5 を比較して分かる様に、想定モデルの規模では LS が RS より数万倍速いと予想できる。京速計算機上では当然 LS アルゴリズムが使われるはずである。悪いことに LS と RS では、必要とされる通信速度やメモリー量がかなり異なる。絶縁体や半導体の場合、波動関数を $10\mathring{A}$ 程度に局在化でき、計算や通信が大幅に減り、必要なリソースは原子数に比例する。指定された RS アルゴリズムのベンチマーク結果に基づいて、計算機を設計、選択しても、見当はずれとなる危険性がある。半導体や絶縁体より金属では軌道局在化が難しいので、LS と RS の差は小さくなるが、LS 法の要求リソースが RS 法より 1 桁以上少ないのは、確実と思われる。

そこで我々はまず RS アルゴリズムでの通信、計算時間等を 46656 原子で検討する。次に LS アルゴリズムと似た状況でベンチマークを行うため、LS アルゴリズムが有効になり始める領域, 512 原子のモデルで同様の解析をし、LS アルゴリズムの性能を定性的に議論する。

### 3.11.2 検討するサブルーチン

2 つの想定モデルの計算パラメターを整理する。

| | | |
|---|---|---|
| 原子数 $MI$ | 46656 | 512 |
| ホスト数 $NP$ | 1458 | 16 |
| 加速チップ数 | $18^3 \times 2$ | $4^3 \times 2$ |
| $Si_8$ の box 数 | $18^3$ | $4^3$ |
| 電子数 $ME$ | $1.9 \times 10^5$ | 2048 |
| 全波動関数 $MB$ | $9.4 \times 10^4$ | 1024 |
| 全格子点数 $ML$ | $1.0 \times 10^7$ | $1.1 \times 10^5$ |

2個の加速チップが $Si_8$ の1 box を担当する。各チップが担当する格子点数は $ML0 = 864$ 点、差分近似の次数 $Md = 4$ である。変数は倍精度複素数型とする。

配布された「実空間 DFT まとめ」(「まとめ」) から $Si_{10648}/Si_{1000}$ での計算/通信時間を抜粋し「ベンチマークコードの概要」(「概要」) に従い総実行時間を算出し、その割合を調べた。計算/ 通信時間は原子数のそれぞれ3乗、2乗に比例し、この割合は2つの想定モデルでも大差無いと思われる。

| | $Si_{10648}$ | | | $Si_{1000}$ | | |
|---|---|---|---|---|---|---|
| 時間（単位秒） | 計算 | (%) | 通信 | 計算 | (%) | 通信 |
| diag 行列要素 | $2.7 \times 10^5$ | (15.1) | $1.0 \times 10^4$ | 250 | (11.0) | 93 |
| diag 回転 | $5.4 \times 10^5$ | (30.1) | $2.0 \times 10^4$ | 500 | (21.9) | 180 |
| DTcg その他 | $3.6 \times 10^5$ | (20.1) | $7.1 \times 10^3$ | 340 | (14.7) | 95 |
| Gram-Schmidt | $5.4 \times 10^5$ | (30.1) | $1.0 \times 10^4$ | 500 | (21.9) | 110 |
| HPsi | $6.7 \times 10^4$ | (3.8) | $8.1 \times 10^3$ | 660 | (28.7) | 190 |
| その他 | $1.5 \times 10^4$ | (0.8) | $1.7 \times 10^4$ | 41 | (1.8) | 430 |
| 計 | $1.8 \times 10^6$ | (100) | $7.2 \times 10^4$ | 2300 | (100) | 1100 |

DTcg その他とは実質 Gram-Schmidt 直交化 (cg-4), (cg-6) と考えて良い。以下各部分の計算法を検討する。

### 3.11.3 $Si_{46656}$ モデルでの結果

diag 行列要素, diag 回転, DTcg その他, Gram-Schmidt を加速チップで行う。残りはホストで行うが、その計算主要部は HPsi(3.8%)、通信主要部は HPsi と diag-Pzheevd である。空間分割した計算では 1458 台のホストのみ使用することを今回は考える。これは HPsi には影響するが、他ではデータの再配置を行うので考える必要はなく、全システムを使う。

MB 個の波動関数はホストメモリーに割付する。1 ホストは 8 チップを担当するので、必要領域は $8 \times 16MB \times ML0 = 10$ GB となる。この 2, 3 倍程度の作業領域が必要で、ホストメモリーに確保できる。

#### 3.11.3.1 Diag 行列要素

「まとめ」の式 (diag-1) の計算法を検討する。 $MB = 9.4 \times 10^4$ 本の 2 組の vector $\phi_m(i), \phi'_m(i)$ はホストメモリー上にある。

ここで行うことは、(ML,MB) 次元の行 $\Phi^*$ と (ML,MB) 次元の行 $\Phi'^t$ の積を求めることである。求まったものは MB 次元の正方行列である。

ML は $10^7$、 MB も $10^5$ と大きいので、適当なデータ再配置を行ってもそれに必要な通信時間は計算時間に比べてほぼ無視できる。従って、行列乗算を良く知られた Canon のアルゴリズムを使って実現するものとして計算時間と通信時間を評価する。

トータルの演算数は、$8MB^2ML$ である (通常の 4 倍なのは複素演算であるため)。演算速度は 10Pflops とすると、理想的に並列化された時の計算時間は 80 秒である。実際に上半三角部分だけで良いので 40 秒である。

通信量を考える。単純に (ML, MB) の行を 2 次元分割したのでは通信量が多くなりすぎるので、 まず MB 次元の正方行列 100 個同士の積の形に書き直し、複数の行列積を並列に行った上で最後に合計することにする。この時、 1 ノードが受け取る通信量は以下の式で与えられる。

$$D_{total} = D_m + D_s \tag{1}$$

ここで、 $D_m$ は乗算の時に受け取るデータ量、 $D_s$ は合計の時に受け取るデータ量であり、それぞれ

$$D_m = 16lb/\sqrt{pg} \tag{2}$$

$$D_s = 16b^2g/p \tag{3}$$

で与えられる。

ここで、 $l = ML, b = MB$ であり、 $p, g$ はそれぞれノード数、ノードグループ数である。ノードグループ数とは、別々の行列乗算を並列に行う単位の数である。 例えば、 2000 ノードを 100 ノードづつの 20 グループに分けて、20 個の行列乗算を並列に行うなら $g = 20$ である。 $D_s$ の計算では、総和はパイプライン的に実行できて、通信時間に対する $\log_2 g$ に比例する項の寄与は無視できるとした。これらの式に意味があるためには $g < l/b$ でなければならない。従って、 $p¿ (l/b)^2$ ならば $D_{total}$ を最小にする $g$ は単純に $l/b$ である。この時、$D_s$ の寄与は無視できる。 $l, b$ 等の具体的な数字を入れると、$D_m$ は 36GB となり、実効転送速度 1.5GB/s としても 30 秒以下で終わる。Canon のアルゴリズムでは計算・転送は並列に実行できるので、行列乗算全体が 40 秒で終わり、ほぼ理論ピークの性能がでることになる。

この step の総実行時間は $40 \times 60 = 2400$ sec となる。

#### 3.11.3.2 Diag 回転

式 (diag-4) の計算だが、これは全体としてみると $(ML, MB)$ の行列 $\phi$ と $(MB, MB)$ の行列 C の行列積計算である。演算量は前節の Diag 行列要素と同じだが、対称性がないので 2 倍になる。計算時間は通信を無視すると 1 回に 80 秒、総実行時間は $80 \times 60 = 4800$ sec となる。

#### 3.11.3.3 Gram-Schmidt

Gram-Schmidt 直交化の計算法を検討する。

gs-1 の計算量は行列要素生成と同じであり、効率 100% なら 40 秒で終わる。gs-3 も同様である。但し、これらはロードバランスを保って並列化するためには、元の行列をブロックサイクリック分割する必要がある。このため、通信量が増大する。ブロックを $3 \times 3$ 程度とすると、通信量が 3 倍になって 90 秒前後となる。この時、ロードバランスの悪化による計算時間増大は通信時間増大より小さいので無視すると、結局直交化の部分は 90 秒程度、理論ピークの 4 割程度の速度になる。

#### 3.11.3.4 DTcg

DTcg の部分 cg-4, cg-6 は形式的には Gram-Schmidt 直交化と同じ形をしていて、計算量が Mcg 倍になるだけだが、 CG 反復の 1 部であるので Gram-Schmidt 直交化のように行列積に書き直せるわけではない。従って、cg-4 では (ML, MB) 行列と 長さ ML のベクトルの積を並列に実行できる必要がある。 ロードバランスを考えないと、行列要素計算の場合と同じく (ML, MB) 行列を正方行列に分割したあとで分散させ、合計を後で計算させるのが効率がよい。 cg-6 の部分についても同様である。これらそれぞれが Mcg 回の反復 1 回につき 80/MB 秒となるので、全体の実行時間は 160Mcg 秒となる。Mcg=4 としてさらに 10 反復とすると、6400 秒程度である。この時の性能は理論ピークのほぼ 1/2 である。

#### 3.11.3.5 HPsi, その他

計算量が少なく通信量が多いのでホストで行う。モデルの大きさ、ホストの数、計算速度を考慮すると、これらの計算時間は表の $(46656/10648)^2 \times (512/1458) \times (0.6/128) = 0.03$ 倍、つまり $(6.7 \times 10^4 + 1.5 \times 10^4) \times 0.02 = 2.6 \times 10^3$ sec 程度となる。他方通信時間は $(46656/10648)^2 \times (0.2/3) \times (512/1458)^{2/3} = 0.64$ 倍、つまり $(8.1 \times 10^3 + 1.7 \times 10^4) \times 0.64 = 1.6 \times 10^4$ sec 程度で、通信時間が支配する。

#### 3.11.3.6 まとめ

トータルの演算時間は $2400 + 4800 + 5400 + 6400 + 16000 = 3.5 \times 10^4$ 秒となる。演算数は $ML \times MB^2$ を単位にすると $60(4 + 8 + 4 + 4) + 10 \times 4 \times 4 = 1360$ であり、

$1.36 \times 10^{20}$ 演算になる。従って、総合的な計算速度は 3.9 Pflops となる。

なお、全体的な計算手順についてまだ詳細を検討中であり、行列演算部分の実装については通信の見積もりが若干楽観的過ぎる可能性がある。実効性能は最悪 3Pflops 程度まで下がるかもしれない。

LS アルゴリズムの場合の予想性能を簡単に説明する。波動関数が空間的に局在化するので、計算/通信量は 512 原子の RS アルゴリズムと似ている。ホスト全体での求和が不要なので、ホスト間通信が減少し効率が上がる。他方 HPsi の割合が大きくなる。この部分は計算に比べて通信が多いため最悪 1/10 程度の効率となる。全体としてこれらが相殺し、3-4 割程度の効率となる。

### 3.12 3DRISM

付属文書の実測データによれば、計算時間の 95%以上が 3DFFT であるので、その部分のみの評価を行う。FFT 以外に計算量の多い部分は UCOULU と MDIIS である。UCOULU（や ULJUV）では natu 個のサイトが作るポテンシャル値を各 grid 点で計算する。これは astro や ParaMD で行う計算と本質的に同じ。また MDIIS も本質的には行列積の計算である。これらは提案システムで高い効率で計算できる。

3DFFT に、最近 BG/L や QCDOC で実装された volumetric FFT を使ったとすると、FFT サイズが固定の時の通信時間はノード数の $-2/3$ 乗に比例する。Eleftheriou et al. (2005) のモデル (実測との誤差は 20% 程度) では、N=128, 512 ノードの時に通信時間が 2ms である。通信時間は FFT サイズの 3 乗に比例する。通信バンド幅は提案システムでは BG/L の 34 倍であるので、サイズ 512、 512 ノードの時の通信時間は 3.76 ms となり、16000 ノードまで増やした時は 0.37 ms となる。演算数は $3.6 \times 10^{10}$ であるので、効率はほぼ 1% である。演算時間は通信時間に対して無視できることになる。

メッシュネットワークを使わないでホスト側のネットワークを使うことを考える。この場合、単純な実装ではグローバルな転置が 2 回発生する。ファットツリーの場合、通信時間は単純にトータルバンド幅に反比例する。転送量はそこそこ多いので、レイテンシは無視してよい。ホスト 2000 ノードの構成では このための通信時間が 0.7 ms となり、さらに遅くなる。しかし、ホスト数を 8 倍程度に増やす、あるいはネットワークバンド幅を同程度に増やせば効率を 5% 程度まで上げることができる。

## 4 加速ボード側ネットワークの必要性について

ここでは、加速ボード側ネットワークの必要性についてまとめる。以下の表は要約である。

| コード | ネットワーク必要度 |
|---|---|
| QCD | 必須 |
| FD | ホスト増強でも対応可能 |
| Multilocas | 不要 |
| astro | 集団通信にはあると有利だが必須ではない |
| GAMESS-FMO | 不要 |
| NICAM_BM | 不要 |
| FrontSTR_BM | ホスト増強で対応可能 |
| simfold | 不要 |
| coevolv | 不要 |
| ParaMD | 不要 |
| RSDFT | 不要 |
| 3DRISM | 不要 |

結局、殆どのアプリケーションで不要である。必須なのは QCD だけである。それ以外のうち FD と FrontSTR_BM はホスト 2000 ノード、ノードあたりのバンド幅 3GB/s のファットツリーという条件では若干ネットワークバンド幅が不足している。ホストノード数とノードあたりのバンド幅の積を数倍にすれば十分である。

astro のうち惑星形成コードでは、ネットワークのレイテンシが性能をリミットしている。総和、放送などを高速に行えるネットワークがあれば理論ピークの 2/3 程度まで性能を上げることができる。これは惑星形成コードだけの特性ではなく、問題サイズが小さい時には必ずネットワークレイテンシが問題になる。今回のベンチマークでは、ベンチマーク部分に通信が入っていなかったり、想定している問題規模が極めて大きいために通信が問題になっていなかったりするケースがあることには注意するべきである。

3DRISM では実効性能が極めて低いが、専用ネットワークはあまり性能向上には助けにならないため不要とした。

多次元 FFT を大規模並列計算機で行って高い効率を実現することは、ネットワークがどのような構成であっても困難である。これはトータルのデータサイズを $N$ とした時に通信量に対する演算量の比が $\log N$ 程度でしかないからである。このため、問題サイズが大きくなってもノード間通信が殆ど減少しない。

この問題に対する抜本的な解決は、これまでスペクトル法が使われてきた気象計算の分野で NICAM のような差分法が提案されており、また DFT 計算でも CG 法ベースの RSDFT が提案されているように、大局的な通信を必要としない計算アルゴリズムを開発することである。今回のベンチマークではそのようなプログラムが多く採択されている。このことは、今後の計算科学技術の方向性を示しているようにも思われる。

しかしながら、短・中期的には多次元 FFT を使った大規模数値計算の必要性は

決して小さくはなく、そのような計算が効率的に行えるシステムを持つことは重要である。シャッフルネットワーク、ファットツリー、ハイパーキューブといった、大域通信で効率が極度には低下しないネットワークでバンド幅が高いものを検討する必要がある。これは、加速ボードというよりは汎用計算機側の課題であると考える。

# 5 加速ボード側メモリの必要性について

ここでは、加速ボード側に大規模メモリをもたせることで性能があがるアプリケーションがあるかどうかについて簡単にまとめる。今回の提案アーキテクチャ評価では、基本的にはメモリはオンチップのチップ当り 32MB、ボード当り 128MB のみとしている。

各ベンチマークについての詳細は既に述べた通りである。 FD では、大規模計算ではホストとの通信バンド幅が性能をリミットするため、計算領域全体をボード側における程度のメモリをつけると大きく性能が向上する。しかし、以外では加速ボード側に大きなメモリがあっても役に立つことはない。 NICAM ではあったほうがコーディングは単純になるが必須ではない。それ以外では全く不要である。

# 6 変更記録

## 6.1 2006/7/28

- ParaMD の通信量推定を詳しくした
- RSDFT の HPsi の部分について、全ノードを使っていないことを反映した数字にした

以上