短距離古典分子動力学計算の高速化と大規模並列化

渡辺宙志
東京大学物性研究所
hwatanabe@issp.u-tokyo.ac.jp

1 はじめに

近年，計算機の動作周波数はほとんど頭打ちになっており，計算能力の増大はほとんどコア数の増大，及びSIMD化に依っている．現在ではデスクトップPCのマルチコア化も進んでおり，その計算能力を使いこなすためには，普段あまりスパコンを使わないユーザであってもある程度の並列化が必要となる．高速化，とくにメモリ管理関連の最適化を考慮したプログラムは，そうでないプログラムに比べて10倍以上の性能差が出ることがあり，それはそのまま研究能力の差につながる．最適化，並列化はもはや一部の専門家の技術ではなく，計算科学を志す上で必須の技術になりつつある．本テキストは短距離分子動力学法を題材に，最適化，並列化をどのように行えば良いか，またその際に気をつけるべきことは何かをまとめたものである．

2 計算機の仕組み

プログラムを高速化するためには，そもそもそのプログラムが遅いのか速いのかを判断する必要がある．そのためには，ある程度計算機の仕組みを理解しておく必要がある．以下では，分子シミュレーションを行う上で最低限知っておいた方がよいことをまとめる．

2.1 物理的な階層性

計算機は階層構造を持っている(図1)．計算を行う場所はCPUであるが，CPUは複数のコアからなり，MPIやOpenMPの並列化の単位となるのはこのコアである．以下，複数あるコア全体をまとめたものを「CPU」，コア一つに着目したものを「CPUコア」と呼んで区別する．Hyper-Threading TechnologyやSMT (Simultaneous Multithreading)[1]などで，論理コアが物理コアより多い場合もある．複数のCPUと，メモリを組み合わせて一つのノードを構成する．ノード内はメモリ空間が共有されており，どのコアからもノード内のメモリ空間全てにアクセス可能だが，一般には物理的に近いメモリと遠いメモリがあり，物理的に遠いメモリにアクセスしようとすると遅くなる．複数のノードを物理的にまとめたものがラックであり，複数のラックをネットワークで接続したものが計算機システム全体を構成する．ラック内の通信はラック間の通信より速いことが多いが，物理的なラック構成と論理上のネットワーク構成が必ずしも一致しないこともあるので注意が必要である．計算機の性能を出すためには，この階層構造を意識してプログラムを書く必要がある．

2.2 仮想メモリとFirst Touch

近年のOSでは，プログラマ側から見えるメモリ領域はページと呼ばれる単位により仮想的に管理されている．プログラマから見えるメモリ領域(論理メモリ)は一定の大きさのページにより分割され，実メモリに割り付けられたり，場合によってはハードディスクにスワップアウトされている(図2)．仮想メモリを採用すると，不連続なメモリ空間を論理的には連続に見せることができる，スワッピングが可能になることで物理メモリよりも広い論理メモ

[1]いずれもCPUコアを実際の数よりも多くみせる技術．OSから見ると論理コアが物理コアの二倍あるように見える．マルチスレッド環境でパフォーマンスが改善する．

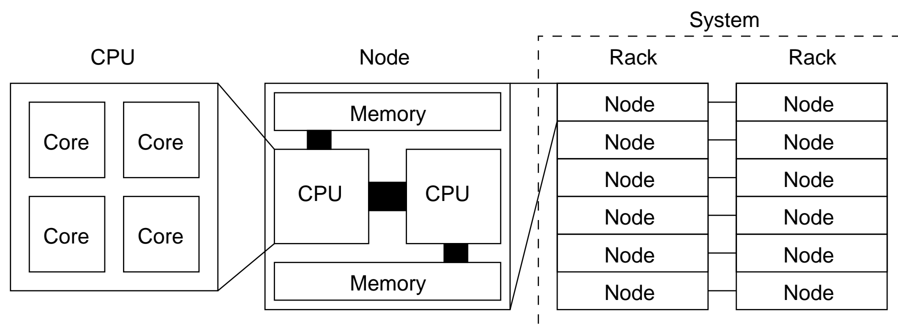

図 1: 計算機の構成．CPUはいくつかのCPUコアからなる．いくつかのCPUとメモリからノードが構成され，複数のノードを物理的にまとめたものがラック，複数のラックをネットワークで接続したものが計算機を構成する．ノード内にはCPUコアから近いメモリと遠いメモリが存在する場合があり，遠いメモリには他のCPUを経由してアクセスしなければならないアーキテクチャもある(NUMA).

リ空間を取ることができるなどのメリットがある．数値計算を行う上で知っておくべき事は,「First Touchの原則」,「NUMA」，そして「TLBミス」の三つである．仮想メモリを採用しているシステムでは，FortranのALLOCATE文や，C/C++のmallocやnewなどでメモリ確保が要求された時にOSはメモリ確保の予約だけを行い，実メモリへの領域確保はされない．プログラムにより，確保した領域にアクセスしようとしたときに初めてページに物理アドレスが割り付けられる．これをFirst Touchの原則と呼ぶ．一般の計算機では，ノード内は共有メモリであっても，あるCPUコアから見て近いメモリと遠いメモリが存在する．このようなアーキテクチャをNUMA (Non-uniform Memory Architecture)と呼ぶ．論理ページは，そのアドレスに触りにいったCPUコアに近い物理メモリに割り付けられる．このため，プログラマはそのメモリ領域を一番使う可能性が高いCPUコアがそのメモリ領域に最初に触るように配慮しなければならない．このようなプログラミングをNUMA最適化と呼ぶ．なお，京やFX10はメモリへのアクセスが全てのコアから対称的であるため，NUMA最適化の必要はない．しかし，メモリの割り付けがページ単位で行われるのは同様であり，もし，大規模計算で初期化が遅く感じる場合，仮想メモリ関連を疑うべきである．例えば，プログラムの最初に巨大な配列を動的に確保している場合，適切な処置を行わないと初期化に時間がかかる場合がある．その場合はページ単位で配列を触るなどの工夫が必要になる．

論理アドレスは物理メモリのアドレスと直接は対応していないため，実際にメモリにアクセスする際には，アドレスの解決を行わなければならない．このアドレス解決処理は時間がかかるため，一度解決した物理アドレスをキャッシュすることで，同じアドレスにアクセスする際に高速にアクセスできるようにする仕組みがTLB (Translation Lookaside Buffer)である．欲しいアドレスがTLB内に存在しない場合，アドレス解決をしなければならない．これをTLBミスと呼ぶ．これも一種のキャッシュミスである．アドレス解決はページ単位で行われるため，ページのサイズが大きければ大きいほど一つのページがカバーする領域が大きくなり，TLBミスが減る[2]．しかし，小規模の領域しか必要がないのにページサイズを大きくすると，未使用の領域が増え，実効的に利用可能なメモリ容量の低下を招くため，プログラムによって最適なページサイズが存在する[3].

[2]ページサイズを大きく取ることをラージページと呼ぶ．

[3]個人的な意見としては，最初はTLBについては意識しなくて良いと思うが，もしプロファイラなどでTLBミ

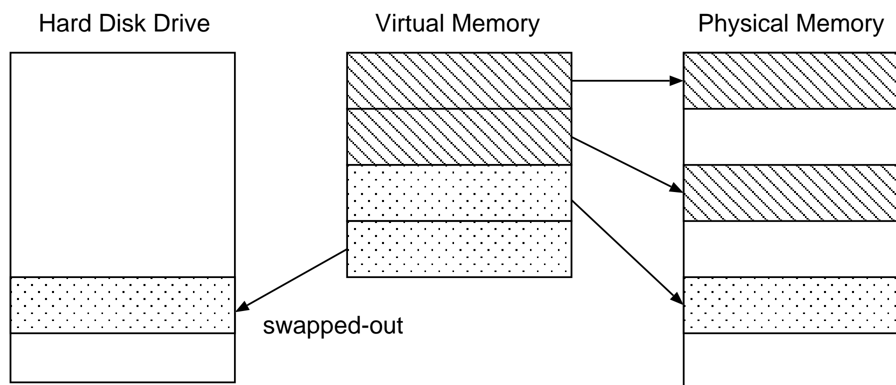

図 2: 仮想メモリ．中央がプログラマから見える仮想的なメモリ空間(Virtual Memory)．論理上ではひとつの連続領域に見えていても，実際にはページ単位で分割され，実メモリ(図右)には不連続に割り当てられているかもしれない．また，一部はハードディスクにスワップされている可能性もある(図左)．

2.3 データ転送能力

記憶媒体にも階層性がある．CPUコアから近い順に，レジスタ，キャッシュ，ノード内のメモリ，他のノードのメモリの区別があり，この順番でアクセスが遅くなる．キャッシュも階層化されており，CPUに近い側からL1(レベル1)キャッシュ，L2キャッシュ，L3キャッシュなどと呼ばれる．計算は全てレジスタ上で行われるため，計算を行う為には，必要なデータをCPUコアにあるレジスタまで持ってこなければならない．計算能力の向上に比べてデータ転送能力はさほど増えておらず，コアから見てメモリが相対的にどんどん「遠く」なっている．メモリとCPU間のデータ転送能力には，要求を出してからデータが届くまでの時間(レイテンシ)と，データが届き始めてから終わるまでに単位時間あたりどれだけのデータを転送できるか(バンド幅)の二種類存在する．レイテンシは小さいほど良く，バンド幅は大きいほど良い．

レイテンシはCPUサイクル数で数えることができる．計算機によって異なるが，典型的にはL1キャッシュまではおよそ数サイクル，L2まで数十サイクル，メモリまでは数百サイクル程度である．注意すべきはメモリアクセスのレイテンシで，キャッシュにのっていないデータにアクセスしようとした場合，CPUはデータが届くまで数百サイクル以上待たされてしまうことになる．広大なメモリ空間をランダムにアクセスするようなプログラムは，計算実行時間のほとんどをレイテンシが占め，CPUはほとんど遊んでいる状態となってしまう．逆に，そこを改善すればプログラムは数百倍速くなる可能性がある．

データの転送能力(バンド幅)の絶対値はByte/sだが，転送能力(Byte/s)と計算能力(FLOPS)との比を使うのが便利である．これをByte/FLOP，通称B/F値と呼ぶ．B/F値が大きいほど計算能力に比較してデータ転送能力が高い．典型的な値は0.5程度である．この値が数値計算にとってどのくらい厳しい値であるか，例を挙げてみてみよう．$A = B \times C$という計算を考える．この計算を実行するには，BとCという倍精度実数をを二つ取って来て，一度掛け算し，Aに書き戻す必要がある．倍精度実数はひとつ8 Byteである．倍精度実数は，ひとつ8 Byteである．2回の読み込み(load)と，1回の書き込み(store)があるから，全体で24 Byteのload/storeが必要となる．一方，計算は1度しかしていない．従って，独立な乗算をひたすら

スが頻発していることがわかった場合には，最適なページサイズを探すなど，なんらかの対策をしなければならないため，頭の片隅には入れておいたほうが良いと思う．

繰り返すような計算がピーク性能を出すためにはB/F値が24以上でなければならない．逆にB/F値が0.5である計算機で$A = B \times C$をただ繰り返すような計算をしようとすると，理論ピーク性能の2%強しか使えず，残りの98%の時間はCPUが遊んでしまうことになる．このようにB/F値がプログラムから要求される値よりも低い場合に計算資源を有効に使う為には，メモリから持って来たデータをCPUに近いところに置いておき，使い回す必要がある．そのためにCPUと主記憶の間に用意されているのがキャッシュである．キャッシュはCPU間と高速に接続されており，キャッシュにあるデータは高速に，かつ低レンテンシでアクセスできる．CPUに近いほど小容量高バンド幅低レイテンシ，CPUから遠くなるほど大容量低バンド幅速高レイテンシである．プログラムを組む際には，いかにCPUに近い記憶媒体に必要なデータを置いておけるかが鍵となる．

2.4 SIMDとパイプライン

現在のCPUでは，命令はパイプライン処理によって実行される．パイプライン処理とは，一つの命令(たとえば加算)を複数のステージにわけ，一度に複数の作業を同時にこなすことで高速に命令を実行する仕組みである．これは，ベルトコンベア式生産ラインと発想が似ている．ベルトコンベアに次々と部品(入力データ)が流れて来て，複数の人が自分の担当の作業をすることで一つの製品(演算結果)を生み出す．命令の実行開始から，その結果が返ってくるまでのサイクル数をレイテンシと呼ぶ．これはコンベアに部品が入ってから，出てくるまでの時間と考えればよい．これがコンベアに隙間無く詰め込まれていれば，毎サイクルなにか結果が出力されることになる．乗算のレイテンシが6であったとしよう．100個の独立な乗算があったとすれば，理想的には106サイクルで計算が完了する．1000個あれば1006サイクルである．このように，十分な数の独立な乗算があれば，平均的に1サイクルで1つの命令が実行できているように見える．これを「スループットが1」と呼ぶ．多くのCPUコアが，二つの命令パイプラインを持っている．従って，スループットが1である命令を1サイクルに2つ実行できる．動作周波数を上げるか，命令パイプラインを増やせばその分CPUコアの性能は上がることになるが，技術，コスト面や電力消費の問題からそれは難しい．そこで一つの命令で複数の計算を実行することで，命令パイプライン数も動作周波数も上げずに理論性能を上げる方法がSIMD (Single Instruction Multiple Data)である．

倍精度実数の加算を考える．倍精度実数は64bitであるから，128bitのレジスタを用意すれば，倍精度実数二つを格納可能である．この状態で128bitレジスタ同士で，上位64bitと下位64bitで独立な加算を実行すれば，一つの命令で二つの独立な加算が実行できることになる[4]．さらに，行列などの計算では$X = A \times B + C$，$Y = A \times B - C$という積和，積差演算が頻出するため，その演算を実行する命令が用意されていることが多い．この積和演算をSIMD化すれば，一つの命令パイプラインで最大4つ，二つのパイプラインで1サイクルあたり8つの倍精度浮動小数点演算が可能となる．京コンピュータのCPU，SPARC VIIIfxは，CPUコアの動作周波数は2 GHzであり，積和のSIMD命令があるため，理論ピーク性能は2 GHz $\times$ 8 FLOP = 16 GFLOPSである．さらに，CPUひとつに8つのCPUコアがあるため，1CPUあたりのピーク性能は128 GFLOPSとなる．

なお，理論ピーク性能が「十分多くの独立なSIMD積和演算が実行されたとき」にのみ達成可能である．命令がSIMD化されていなければ，それだけで性能は半分に，さらに積和演算が実行されず，乗算のみ，加減算のみの場合にはさらに性能が半分になる．それぞれの計算が独立でない場合は，直前の計算の結果が出るまで次の計算を開始できないため，レイテ

[4]IntelのSandy Bridgeマイクロアーキテクチャでは，256bitのレジスタを用意することで一度の4つの倍精度演算，8つの単精度演算が実行可能となった．

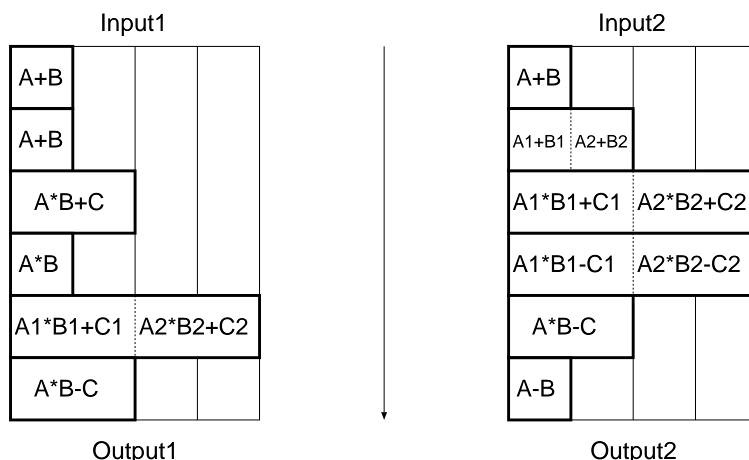

図3: パイプラインとSIMDの実行イメージ．幅が4のベルトコンベアが2ラインあり，それぞれに幅1から4の荷物(命令)を一つずつ流す事ができる．全て幅4の荷物でコンベアを埋めた時がピーク性能である．また，荷物を乗せてから製品として出てくるまでの時間がレイテンシに対応する．

ンシの分だけ遅くなり，パイプラインはほとんど埋まらず，性能がでないことになる．独立な演算でなければSIMDにより同時に演算することはできないため，いかに独立な演算を多数用意するかが性能向上の鍵となる．

3 メモリ最適化

既に述べたように，CPUとメモリ間のデータ転送が多くのプログラムのボトルネックになっている．単体チューニングのキモはほとんどメモリ最適化にある．しかし，メモリの保持の仕方，及びその最適化は研究対象によって異なり，一般論は難しい．以下では，分子シミュレーションを例にキャッシュ効率向上とレジスタ活用の二つの事例を紹介する．

3.1 キャッシュ効率向上

時間発展により，シミュレーションの空間的には近い位置にある粒子が，メモリ上では遠い位置に格納されてしまう場合がある(図4 (a))．このまま相互作用を計算すると，アルゴリズムによってはキャッシュミスが頻発し，ほとんどの時間がメモリからのデータ転送待ちとなり，性能が極端に劣化する．この状況を改善するため，空間的に近い粒子の情報が，メモリ上でも近い位置に格納されるようにソートを行う．具体的には系を相互作用範囲程度の大きさのセルに分割し，その中で連続した粒子番号を持つように番号を振り直す．すると，計算に必要な情報がキャッシュに存在する確率が高くなり，性能が大きく向上する．粒子をランダムに配置してから計算をした場合と，ソートしてから計算した場合の計算速度の粒子数依存性を図5に示す．粒子が少ない場合は全てがキャッシュにのっているために性能差はでないが，粒子数が増えてキャッシュから溢れるようになると，キャッシュヒット率がそのまま計算速度の差としてあらわれる．ソートを行っていない場合，L2キャッシュとL3キャッシュの容量から溢れた時にそれぞれ性能が大きく劣化していることがわかる．逆にソートをしている場合では計算速度があまり粒子数に依存せず，キャッシュが有効に使えていることがわかる．キャッシュの有効利用は必要メモリバンド幅を下げるため，その意味においても有用である．

なお，ここでのソートは完全なソートではなく部分ソート (partial sort)となっている．完全

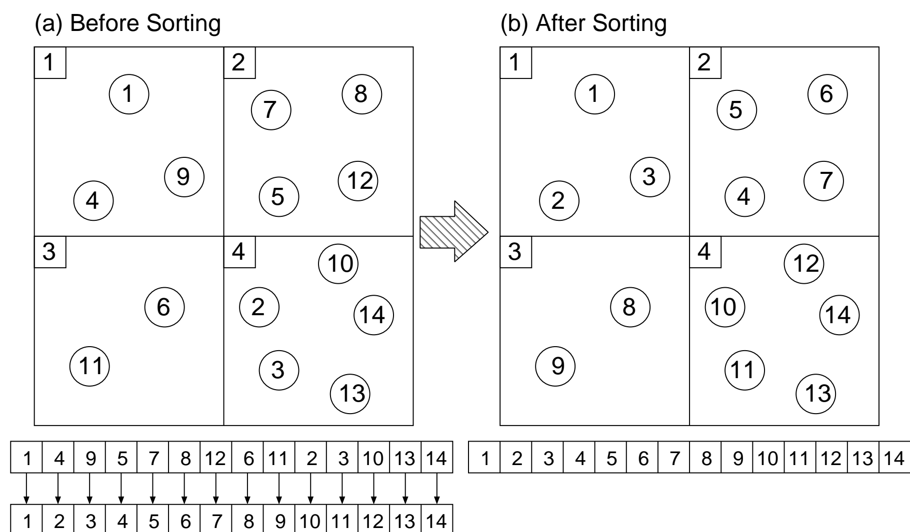

図 4: 粒子番号の空間的なソート．(a) 時間発展により，空間的に近い粒子がメモリ上では遠い位置に格納される場合がでてくる．(b) 空間的に近い粒子がメモリ上でも近くなるように，粒子番号を振り直す．

なソートの計算量は要素数をNとして$O(N \log N)$であるが，部分ソートであれば$0(N)$で可能である．粒子の配置に限らず，キャッシュ効率の向上はある種のソートになっていることが多く，特に計算量が小さくて済む部分ソートは有効であることが多い．

3.2 レジスタの活用例

メモリ階層の一番CPU側に存在する記憶媒体がレジスタである．全ての演算はレジスタを通して行われる．一般に，レジスタを効率的に使うコードを人の手で書くのは困難で，コンパイラに任せた方が良い結果がでることが多い．それでも，たまにレジスタを意識した方が良い場合もある．List 1は，全ての粒子間において12-6型のLennard-Jonesポテンシャルによる力積を計算するコードを単純に書き下したものである．ここで，外側のループのインデックスがi，内側のループがjであるので，それぞれのインデックスに関わる粒子をそれぞれi粒子，j粒子と呼ぶことが多い．計算の詳細を追うと，まずi粒子とj粒子の距離を計算し，距離から力積を計算し，力積を運動量に書き戻している．ここで，i粒子の運動量も，j粒子ごとに和を取っているが，実際にはここでデータを書き戻す必要はない．一度，テンポラリ変数にj粒子からの力積の和を集めておき，j粒子のループの最後に和だけi粒子の運動量に書き戻した方が，メモリアクセスの面で効率的である．そのような修正をしたのがList 2である．これは，i粒子の情報を全てレジスタに載せておくことを意識したコードとなっている．この効果を調べるため，いくつかの環境にて力の計算を行うのにかかった時間を表1に示す．環境はコンパイラに強く依存するが，おおむね5%程度，場合によっては2倍近い性能向上が得られている．

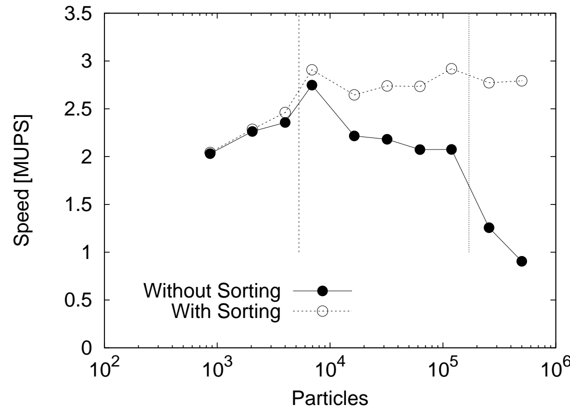

図 5: 性能の粒子数依存性．横軸が粒子数，縦軸が計算速度で，単位時間あたり何個の粒子を更新できたかを表す．L2とL3キャッシュ容量に対応する粒子数を点線で示す．キャッシュをうまく使えていない場合(黒丸)は，キャッシュからデータが溢れる度に性能が急激に劣化していくのが分かるが，キャッシュを有効に使えている場合(白丸)では性能の粒子数依存性がほとんどなく，キャッシュが有効に使えていることがわかる．

環境及びコンパイラ	工夫無し [s]	レジスタ意識 [s]
Mac OS X (Xeon 2.93GHz + gcc)	2.92	2.78
ISSP システム B (Xeon X5570, 2.93GHz) + gcc	3.00	2.82
ISSP システム B (Xeon X5570, 2.93GHz) + icc	3.03	1.55
Linux (Core i7 3.5GHz) + gcc	1.81	1.60
Linux (Core i7 3.5GHz) + icc	0.99	0.94

表 1: 2万粒子を3次元空間にランダムに配置し，全ての粒子対についての力を計算するのにかかった時間．コンパイラはgccとIntel Compiler (icc)を，コンパイルオプションはどちらも-O3を指定している．

List 1: LJポテンシャルの計算．何も工夫しない場合．

```
void
calcforce(void){
  for(int i=0;i<N-1;i++){
    for(int j=i+1;j<N;j++){
      const double dx = q[j][X] - q[i][X];
      const double dy = q[j][Y] - q[i][Y];
      const double dz = q[j][Z] - q[i][Z];
      const double r2 = (dx*dx + dy*dy + dz*dz);
      const double r6 = r2*r2*r2;
      const double df = (24.0*r6-48.0)/(r6*r6*r2)*dt;
      p[i][X] += df*dx;
      p[i][Y] += df*dy;
      p[i][Z] += df*dz;
      p[j][X] -= df*dx;
      p[j][Y] -= df*dy;
      p[j][Z] -= df*dz;
    }
  }
}
```

List 2: LJポテンシャルの計算．レジスタを意識した場合．

```
void
calcforce(void){
  for(int i=0;i<N-1;i++){
    double pix = p[i][X];
    double piy = p[i][Y];
    double piz = p[i][Z];
    for(int j=i+1;j<N;j++){
      const double dx = q[j][X] - q[i][X];
      const double dy = q[j][Y] - q[i][Y];
      const double dz = q[j][Z] - q[i][Z];
      const double r2 = (dx*dx + dy*dy + dz*dz);
      const double r6 = r2*r2*r2;
      const double df = (24.0*r6-48.0)/(r6*r6*r2)*dt;
      pix += df*dx;
      piy += df*dy;
      piz += df*dz;
      p[j][X] -= df*dx;
      p[j][Y] -= df*dy;
      p[j][Z] -= df*dz;
    }
    p[i][X] = pix;
    p[i][Y] = piy;
    p[i][Z] = piz;
  }
}
```

4 CPUチューニング

計算に必要なデータがほぼキャッシュにのっていると思われるにも関わらず性能がでない場合，CPUチューニングを行うことになる．しかし，一般的にはCPUチューニングは手間のわりに効果が低いことが多く，自分でチューニングするより，コンパイラに任せた方が性能が出る場合が多い．以下では分子シミュレーションで有用ないくつかの最適化技術を紹介する．

4.1 条件分岐削除

一般に，プログラムの開発はIntel系のCPUを搭載したPCで行うことが多いであろう．PCに比べ，京コンピュータやIBM POWERなどのCPUを搭載したスパコンでは性能が出ない，ということはよくあるが，その原因の一つが条件分岐である．分子動力学計算ではカットオフを設けることが多いため，粒子間距離を計算し，カットオフ距離以上であればループの次へジャンプ(continue)させることが多い(Algorithm 1)．この条件分岐によるジャンプはIntel系以外のアーキテクチャではペナルティが高く，性能劣化を招きやすい．この問題の簡単な対処法は，カットオフにかまわず力を計算してしまい，その後で粒子間距離をチェックして，もしカットオフ距離以上であれば計算した力をゼロにクリアしてしまうことである(Algorithm 2)．粒子間距離をチェックした後に力の計算を行う場合に比べて計算量には無駄が生じるが，ループがジャンプフリーになることで様々な最適化が効くようになる．さらに，条件により代入する値を選ぶ代入命令[5]がアセンブリレベルで用意されていることが多く，場合によっては数倍以上高速化される．Intel系のCPUにおいても，SIMD化を考慮すると無駄な計算を増やしてでも条件分岐を削除したほうが高速になる場合もある．

[5]Floating-point Selectionの略で，IBM POWER系ではfsel，京やFX10ではfselmovd命令が用意されている．C言語の三項演算子を用いた代入命令が一命令で実行できる．

Algorithm 1 条件分岐ジャンプのある力計算ルーチン

1: **for all** (i, j) in pairlist **do**
2: $r \leftarrow |\mathbf{q}_i - \mathbf{q}_j|$
3: **if** $r >$ cutoff length **then**
4: **continue**
5: **end if**
6: $f \leftarrow$ force between particles i and j.
7: Update momenta with f.
8: **end for**

Algorithm 2 条件分岐削除をした力計算ルーチン

1: **for all** (i, j) in pairlist **do**
2: $r \leftarrow |\mathbf{q}_i - \mathbf{q}_j|$
3: $f \leftarrow$ force between particles i and j.
4: **if** $r >$ cutoff length **then**
5: $f \leftarrow 0$
6: **end if**
7: Update momenta with f.
8: **end for**

4.2 除算削除

Intel系のPCに比べて，そうでないアーキテクチャでプログラムが遅くなるもう一つの原因は除算にある．一般に除算は乗算，加算に比べて数倍以上遅い．ループ内で変化しない数での除算であれば逆数の乗算に変形することが可能であるが，例えばLennard-Jonesポテンシャルやクーロンポテンシャルの計算では，粒子対ごとに最低一度は除算を実行する必要がある．このとき，アーキテクチャによっては以下の変形による除算削除が有用である．B_1とB_2の二つの変数の逆数を計算したいとしよう．

$A_1 \leftarrow 1/B_1,$
$A_2 \leftarrow 1/B_2.$

この計算は，一度B_1とB_2の積を考えることで，以下の計算に変形できる．

$C \leftarrow 1/(B_1 \times B_2),$
$A_1 \leftarrow C \times B_2,$
$A_2 \leftarrow C \times B_1.$

除算が一つ減り，乗算が三つ増えているため，除算が乗算の3倍以上「重い」場合には計算速度の向上が見込める[6]．実際の計算では，力の計算ループを二倍展開することで独立な除算を二つ作り，上記の変形により除算を一つ減らすということを行う．煩雑になるため，興味のあるかたは文献[1]を参照されたい．

4.3 SIMD化

SIMD命令は独立な計算を一度に行うことでサイクルあたりの演算数を稼ぐ．SIMD命令を実行するためには，独立な演算を多数抽出し，それらをレイテンシが隠れるようにうまく配置してやる必要がある．独立な演算を増やす方法には，大きく分けてループアンローリング

[6]アーキテクチャにも依存するが，加減算，乗算に比べて除算は5倍以上のサイクル数がかかることもある．

とソフトウェアパイプライニングの二種類が存在する．

ループアンローリングとは，ループ数を減らして，ループ内の命令数を増やす方法である．例えば以下のような単純ループを考える．

```
1: for i = 1 to N do
2:     a[i] = b[i] + c[i]
3: end for
```

このループは，以下のループと等価である(簡単のため，Nは偶数であるとする)．

```
1: for i = 1 to N/2 do
2:     a[2i-1] = b[2i-1] + c[2i-1]
3:     a[2i] = b[2i] + c[2i]
4: end for
```

ここで，二倍に増えた演算の左辺は独立に実行することができるため，SIMD化命令にまとめることができる．これは自明SIMD化(通称「馬鹿SIMD化」)と呼ばれる方法であり，コンパイラメッセージで「Loop unrolled ×× times」などと出力されたら，この種の最適化が行われている可能性が高い．この方法ではSIMD命令率は増やすことができるが，もしループ内の演算に依存関係がある場合，そのレイテンシは隠すことができない．例えば，

```
1: for i = 1 to N do
2:     a[i] = b[i] + c[i]
3:     d[i] = a[i] × a[i]
4: end for
```

という計算があった場合，二行目の計算は一行目の計算が終わるまで実行できず，これはSIMD化しても同様である．

ここで，依存関係は添字iを共有していることに注意したい．このようなループインデックス内(同じi間)では依存関係があるが，ループインデックス間(異なるi間)には依存関係がないようなループに有効なループ変換がソフトウェアパイプライニング (Software pipelining)である．具体的には以下のような変換をする．

```
1: a[1] = b[1] +c[1]
2: for i = 1 to N − 1 do
3:     a[i + 1] = b[i + 1] + c[i + 1]
4:     d[i] = a[i] × a[i]
5: end for
6: d[N] = a[N] × a[N]
```

ループ内のインデックスを「ひとつずらす」ことで，ループ内の一行目と二行目は独立に実行できるようになっていることがわかる．ここではループ内に二行しか演算がないので二段にずらしているが実際にはより多段にずらすことで，独立な演算を大幅に増やすことができる．その上で独立な演算をうまく組み合わせてSIMD化することで，サイクルあたりの命令数を増やし，かつレイテンシを隠すことができる．コンパイラメッセージで「Loop software pipelined」と出力されたらこの種の最適化が行われている．

パイプラインの段数が深く，かつアウトオブオーダー実行があまり得意でないアーキテクチャでは，ループアンローリングよりも，ソフトウェアパイプライニングを行った方が性能がでることが多い．簡単な場合には上記のループ変換はコンパイラが自動で行ってくれるため，通常はプログラマはあまりループ変換を意識する必要はない．しかし，コンパイラが自

動でソフトウェアパイプライニングできず，そのために性能が劣化していると思われる場合には，テンポラリ領域を用意してループのインデックス間の依存関係をなくすなど，なるべくコンパイラがソフトウェアパイプライニングしやすいようなコードを用意するなどの最適化が必要となる場合もあるだろう．その種の最適の中で簡単なものの一つは，作用反作用の法則を使わないことである．通常，力の計算は作用反作用の法則を用いて，片方の粒子から受ける力積を計算したら，符号を反転してもう片方の粒子の運動量に加えるということを行う．

```
1: for i = 1 to N do
2:   for all j such that interacting with i do
3:     f ← force between i and j
4:     p_i ← p_i + f dt
5:     p_j ← p_j − f dt
6:   end for
7: end for
```

このとき，内側のループにおいて同じjの値は二度出現しないが，コンパイラはそれを判断できずに積極的な最適化ができない場合がある．さらに，p_jへの書き戻しは一般に間接参照となるため，メモリアクセスのレイテンシのペナルティが大きい．そこで，作用反作用の法則を使わず，二倍計算することにする．

```
1: for i = 1 to N do
2:   p_tmp ← p_i
3:   for all j such that interacting with i do
4:     f ← force between i and j
5:     p_tmp ← p_tmp + f dt
6:   end for
7:   p_i ← p_tmp
8: end for
```

一番内側のループでは，メモリへの書き戻しが無いことに注意したい．メモリアクセスは読み込みだけとなり，書き込みはレジスタにのみ行われ，メモリへの書き込みは最後にまとめて行われる．これにより計算量は倍になるが，計算時間が倍以上短くなることがある．特にメモりへの書き込みが遅いアーキテクチャでは有用である[7]．

5 並列化

並列化ではプログラム技術にばかり注意が向きがちであるが，そもそもなぜ並列化をしたいのか，なんのために並列化をしたいのかを明確にしておかないと思わぬ落とし穴にはまることがある．また，並列化プログラムが完成し，シングルノードや研究室のクラスタでのテストも終わり，いざ大規模計算，となったときに初めて出会う，大規模計算特有の問題がいくつか存在する．これらは大規模計算で初めて現れるため，知らなければ事前の対策が難しい．以下では，大規模計算を行う上で考えておいた方が良いこと，知っておいた方が良いことをまとめる．

[7]我々のコードでは作用反作用を使わない方が計算が速かった．このとき，FLOPS値は二倍以上の値になるが，もちろん意味のある指標とはなっていない．重要なのは自分の行いたい計算がどれだけ速く実行できるかである．

5.1 プログラムの粒度

並列化には空間を稼ぐ並列化と時間を稼ぐ並列化の二種類が存在する．シングルノードのメモリには乗り切らない大きな問題を扱いたい場合には，より大きなメモリ空間を使うために並列化が必要となる．並列化を行うことで，1ステップにかかる時間を減らし，計算の実行時間を短くしたい場合もあるだろう．これは，後述のウィークスケーリングとストロングスケーリングに対応する．並列化の性能を特徴づける量として並列化効率があるが，この並列化効率の測定には，二種類の方法がある．ひとつが，ノードあたりの計算量を固定したまま，ノード数を増加させ，計算時間を測定するもので，ウィークスケーリングと呼ばれる．もうひとつは，全体の計算規模を固定したままノード数を変えるもので，ストロングスケーリングと呼ばれる．ノード数がNの時，計算時間がt_Nであったとしよう．ウィークスケーリングでは並列化効率はt_1/t_Nで定義される．理想的な場合には計算時間はノード数に依存しないため，$t_1 = t_N$の時に並列化効率1，すなわち100%の効率となる．ストロングスケーリングの場合は，理想的には計算時間はノード数に反比例するため，並列化効率はNt_N/t_1で定義される．ストロングスケーリングでは，ノード数が増えるほどノードあたりの計算量が小さくなり，結果として通信の比重が大きくなるため，性能を出すのが難しくなる．

並列プログラムで重要なのは，プログラムの実行時間に占める計算時間と通信時間の比である．計算時間に比べて通信時間が小さい時，このプログラムの粒度(Granularity)が疎であると呼ぶ．逆の場合は粒度が密である．粒度が疎であるプログラムはノード数が増加しても性能が劣化しにくく，大規模計算向きである．逆に粒度が密である場合には大規模計算が難しくなる．シングルノードでは疎であっても，ストロングスケーリングによってノード数が増えた場合には粒度が密になることで通信時間が支配的になり，並列化効率が落ちることが多い．

メモリアクセスの場合と同様に，通信においてもレイテンシとバンド幅の両方に注意しなければならない．通信を開始しようとしてから通信が始まる時間をレイテンシ，通信が始まってからどれだけのデータをながせるかがバンド幅である．扱うシステムサイズが小さければレイテンシが，大きければバンド幅がボトルネックになりやすい．通信部分を最適化する際には，まずレイテンシとバンド幅のどちらが律速となっているかを判断する必要がある．

なお，並列化効率に似た用語で,「並列化率」という単語がある．これは，プログラム全体のうち，どれだけが並列実行可能であるかを示す割合である．例えば並列化率が99%のプログラムがあった場合，100並列にて実行すると，実行時間は$0.99/100 + 0.01 = 0.0199$，つまり50倍にしかならない．並列化率が100%でないプログラムは，並列数が増えた時に急速に性能が悪くなる．1000並列を超えるような大規模計算を志すのであれば，並列化率は原則として100%でなければならない．

5.2 系のサイズと計算時間

並列化を行う前に，まず興味ある対象の時間スケールを決める要因とシステムサイズとの関係を考える必要がある．いま，一辺の長さがLであるような三次元系をシミュレーションしたいとする．密度を固定すれば，系内の要素数NはL^3に比例する．いま，アルゴリズムが$O(N)$であり，並列化効率が完璧なプログラムを持っているとしよう．すると，1ステップの計算量はL^3に比例する．しかし，系が熱平衡状態に緩和するまでの時間は，システムサイズが大きくなるにつれて長くなる．一般に緩和時間はL^2に比例して長くなるため，系の平衡状態に興味があるのであれば全体の計算量はL^5に比例して大きくなる．ジョブの実時間には制限があるから，これは系が大きいほど1ステップあたりの計算時間を短くして，長いステップ数を実行する必要があることを意味する．すなわち，平衡状態に興味がある場合には，プログラムがストロングスケーリングにおいて性能がでることが要求される．なお，緩

和時間がL^2に比例して伸びるのは，系が拡散に支配されている場合である．ガラス系や，相転移点近傍の臨界緩和がある系では，さらに緩和時間が長くなる．

興味ある対象が爆発，気泡生成などの非平衡非定常系である場合，音速や気泡生成率など系のサイズにあまり影響されないものが系のダイナミクスを支配するため，(初期条件の準備以外では)系を平衡化する必要がない．この場合は計算量は$O(L^3)$で良い．これは非平衡非定常系の大規模計算に使うプログラムはウィークスケーリングで性能が出れば良いことを意味する．シェアがかかった系など，非平衡定常系においては緩和時間が系のサイズに応じて長くなるため平衡系と同様にストロングスケーリングが要求される．

5.3 Flat-MPIか，ハイブリッドか

超並列計算で知っておくべきことは，MPIプロセスはOpenMPスレッドに比べてメモリ利用量が大きいことである．まったく同じ規模の計算を行った場合でも，flat-MPI(全てMPIプロセスによる並列化)の場合とハイブリッド(MPI+OpenMPによる並列化)ではメモリの使用量が大きく異なる．東大物性研のSGI Altix ICE 8400EXの1024コアにおいて，flat MPI (1024プロセス)とハイブリッド(128プロセス×8スレッド)の計算を行った結果を表2に示す．Flat-MPIの方が実行速度は早いが，69TBだけ余計にメモリを使っていることがわかる．詳細は実装に依存するが，一般に並列数が大きくなればなるほどMPIの「プロセスあたり」のメモリ使用量が大きくなる傾向にある．従って，100並列では大丈夫だった計算が，ウィークスケーリング(ノードあたりの計算規模を固定してノード数を増やす方法)であるにもかかわらず1000並列ではメモリ不足で失敗する，ということがおきる．その場合は環境変数を修正することでMPIの利用メモリを減らしたり，ハイブリッド化によりMPIのプロセス数を減らすなどの工夫が必要となる．

並列化手法	実行時間 [s]	利用メモリ [TB]
Flat-MPI	249.36	268
Hybrid	257.09	199

表 2: Flat-MPI並列(1024プロセス)とHybrid並列(128プロセス×8スレッド)の性能比較．カットオフ付きLennard-Jonesポテンシャルの計算．直径を1として$800 \times 800 \times 1600$，密度0.5で5億1千万粒子を1000ステップ計算するのにかかった時間と利用実メモリ量．

また，MPIはメモリ以外の資源も大量に使うため，大規模実行時に資源の枯渇がよく起きる．通信時間の隠蔽のため，MPI_Isendなどのノンブロッキング通信を使うことも多いだろう．しかし，一般にノンブロッキング通信はブロッキング通信よりも多くのメモリ，多くの種類のバッファを使うため，大規模並列時に問題を起こしやすい．例えば小規模並列では問題なかったプログラムを大規模並列実行した時に「MPI_REQUEST_MAXが足りない」といったエラーメッセージとともにジョブが失敗することがある．MPI_REQUEST_MAXは同時に行うことができるノンブロッキング通信の最大数であり，これが不足する場合には，通信を小分けにするか，一部をブロッキング通信に置き換えるなどの工夫が必要となる．また，REQUEST_MAX以外にも「MPI_○○○_MAXが足りない」というタイプのエラーはよく出現する．その際は対応する環境変数を上げる必要があるが，その背景にはメモリ不足が疑われるため，ハイブリッド化によりノードあたりのプロセス数を減らしたり，問題サイズを小さくしたりする必要がある．

6 分子動力学法の並列化

分子動力学法による計算を並列化する際，多くの場合空間分割による並列化を採用するであろう．以下では，空間分割により並列化された分子動力学法特有の事情についていくつか紹介する．

6.1 境界領域の情報の通信方法

対象によって様々な分割の方法があり得るが，もっとも単純には三次元的に直方体的で分割するであろう．その際，隣接する領域は26個存在するが，通信は26回しなくても良い．面を接する領域と情報を交換した後，受け取った粒子情報を転送することで，通信回数は6回で済む(図6参照).

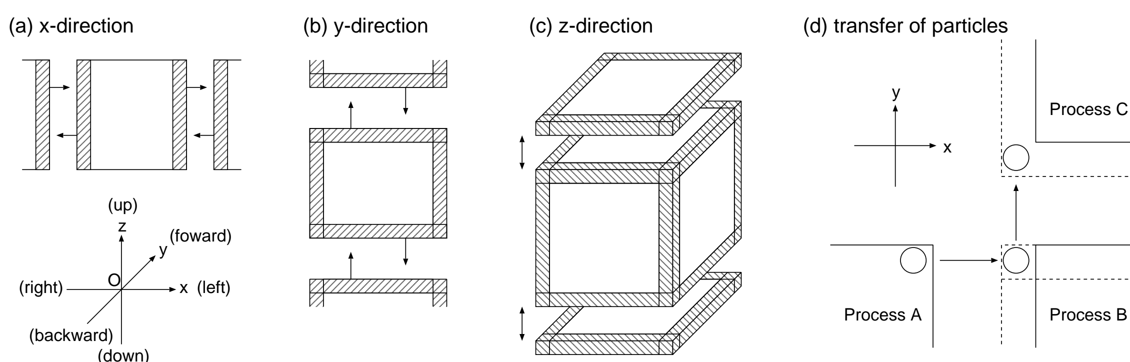

図 6: 境界領域にいる粒子の通信方法．(a) まず，x方向に隣接する領域と粒子情報を交換する．(b) 次に，y方向に隣接する領域と粒子情報を交換するが，その際にx方向の通信でもらった粒子も送る．(c)同様にx，y方向でもらった粒子も含めてz方向に隣接する領域と粒子情報を交換する．(d) 面を接する領域同士(図ではプロセスA-B間)で交換した情報を転送することで，$x-y$の斜め方向(図ではプロセスA-C間)の情報交換ができたことになる．

6.2 ペアリスト更新のタイミング

相互作用にカットオフのある粒子系を用いる場合，Bookkeeping法と呼ばれる最適化アルゴリズムがよく用いられる．力の計算をするためには，相互作用範囲内にいる粒子のペアのリスト(ペアリスト)を作成する必要がある．ペアリストの探索を全探索で行うと$O(N^2)$の計算量がかかるため，通常は空間をセルに分割し，粒子を住所登録した上，住所から逆引きして相互作用相手を探すことで計算量を$O(N)$に落とす．しかし，この方法はメモリ上をランダムアクセスすることになるため，極めて時間がかかる．そこで，ペアリストを作る際に相互作用距離より長い距離にいる粒子ペアを登録し，そのマージンの分だけペアリストを再利用する．これがBookkeeping法である．マージンが長ければペアリストが再利用できる回数が増えるが，その分ペアリストの構築に時間がかかり，力の計算でも無駄が増えるため，最適な長さが存在する．詳細については文献[1]と，その中の引用文献を参照されたい．ペアリストをある回数だけ再利用したら，ペアリストに登録されていないにも関わらず相互作用範囲内に入ってくる粒子対があらわれる可能性があるため，そのチェックを毎ステップ行わなければならない．ペアリストは各並列プロセスで独立に保持し，それぞれについて有効期限チェックを行うが，一つでも有効期限が切れたものがあれば全てのプロセスでペアリストの再構築を行う必要がある．

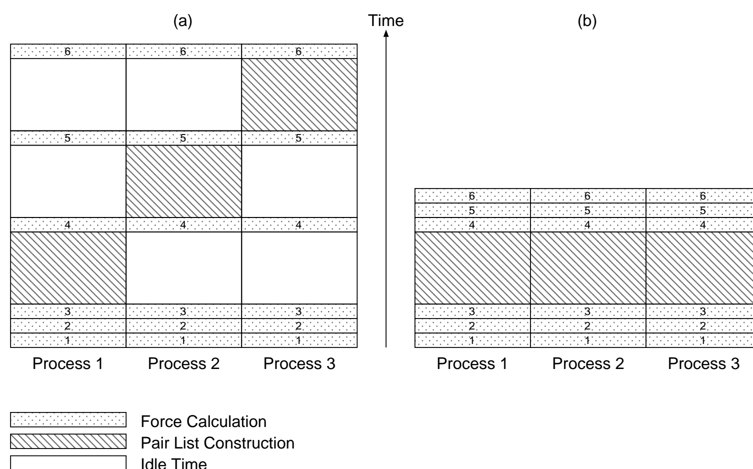

図 7: ペアリストの同期更新．プロセスが3つあり，それぞれ3,4,5ステップ目でペアリストの有効期限が切れるとする．(a) もしプロセスそれぞれで独立にペアリストの更新を行うと，プロセス1がペアリストの更新中は残りのプロセスが待たされる．同様にプロセス2，3のペアリスト更新中も他のプロセスが待たされるため，全体として待ち時間(Idle Time)が増えることになる．(b) もし一つでもペアリストの有効期限が切れた場合，全体でペアリストの更新を行うと，ペアリストの平均寿命は短くなるかわりにペアリストの更新に伴う待ち時間はなくなるため，性能は大きく向上する．

7　おわりに

計算機の仕組みから，分子動力学法における高速化，並列化において知っておくと有用な知識をまとめた．ここで強調しておきたいのは，高速化や並列化は手段であって目的でないということである．あくまでも計算物理の主眼はサイエンスにあり，重要なのは研究着手開始から論文出版までの時間であって，高速化により得られた速度向上より高速化にかけた時間が長ければ意味がない．もちろん高速化，並列化は研究を遂行する上での武器である．他の誰よりも性能の良いコードが書ければ，他の誰にもできない計算，研究をできる可能性も広がる．しかし，個人的な経験から言えば，高速化，並列化はハマると深遠な闇が待ち受けている．研究自体もそうだが，あくまでも自分が楽しいと思える範囲で取り組むのが良いだろう．本稿で紹介した最適化を施した分子動力学法コードのソースも公開しているので，興味のあるかたは参照されたい[2].

謝辞

本稿で紹介した内容は，理化学研究所計算科 学研究機構の京速コンピュータ「京」の試験利用，東京大学情報基盤センター PRIMEHPC FX10における「大規模HPCチャレンジ」の採択結果，及び東京 大学物性研究所のSGI Altix ICE 8400EXにおける大規模キューを使わせていただいた結果を含みます.

参考文献

[1] H. Watanabe, M. Suzuki, and N. Ito, Prog. Theor. Phys. **126** 203–235 (2011).

[2] http://mdacp.sourceforge.net/