急減圧液体における気泡分布関数の数値的解析

渡 辺 宙 志

東京大学物性研究所

鈴　木　将

九州大学大学院工学研究院

伊 藤 伸 泰

東京大学工学系研究科

1. はじめに

我々は気液混相流の全粒子計算を目指し、大規模な計算が可能な分子動力学法コードの開発、及びそれを用いた研究を行っている。気液混相流の研究とは、典型的には熱いパイプの中を流れる流体の振る舞いを理解しようというものである。外からの熱流入を受け、流体は沸騰をしながら流れることになり、相転移と流動がカップルした現象を起こす。こういった気液混相流は冷却システムの安全確保やガソリンエンジンの効率向上などの工学応用上、非常に重要であるが、熱伝導や流動などの輸送現象に加え、沸騰という相転移も複雑にからみあう強い非平衡現象であることから理論的な解析が難しい。数値計算課題として見た気液混相流は、相転移というミクロな挙動と輸送・流動というマクロな挙動が同居する典型的なマルチスケール問題であり、さらに相転移による大きな密度変化、相界面の移動および生成消滅などが計算を難しくする。これまでこのようなマルチスケールな問題については、現象をそのスケールにより分類し、計算するスケールよりも小さいスケールについては粗視化された構成式を用いることで研究が行われてきた。粗視化に用いるパラメータは実験と比較することで決められるが、気泡と流体の間の摩擦など実験で決定することは難しいパラメータも存在し、それらは経験的な扱いがなされている。それに対し、流体の構成分子を全て陽に扱うことで、(古典的な意味で) 非経験的な知見を得ようとするのが我々が目指す全粒子計算である。構成粒子のミクロな振る舞いは、粒子間のポテンシャルを与えることにより決まるニュートン方程式に支配されている。このニュートン方程式を数値積分することで系の時間発展させ、その振る舞いを調べる手法が分子動力学法 (Molecular Dynamics method, MD) である。気液混相流の全粒子計算とは、流体を構成する粒子を分子動力学法により計算することで、マルチスケールな現象をミクロな相互作用から直接再現しようとする試みであり、これまで人為的に導入されてきた粗視化階層の妥当性や、各階層における構成方程式の検討、そしてミクロな輸送現象がマクロにどのようにつながっていくかをシームレスに研究することを目指している。しかし、マクロな系をミクロから計算するためには必然的に計算は大規模なものとなり、本質的に並列計算が必須となる。並列計算の観点で言えば、短距離力古典 MD は並列化は比較的容易であり、コアあたりの粒子数が多い計算を行えば並列化の粒度も疎となり、良好なスケーリングが期待される。しかし、単なるベンチマークにとどまらず、具体的に相転移を伴う大規模なプロダクトランを実行するためには観測ルーチンも含めた並列化、計算実行カーネルの高速化など、解決すべき問題が多い。本稿では、東京大学情報基盤センターの PRIMEHPC FX10 に向けたカットオフ付き Lennard-Jones 粒子の計算の高速化手法、及び急減圧による多重気泡核生成の計算結果について報告する。

2. アルゴリズムと高速化手法

2.1. 基本アルゴリズム

本研究で用いる粒子間ポテンシャルは、カットオフのある 6-12 型 Lennard-Jones ポテンシャルである。その関数形は以下で与えられる [1]。

$$V(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} + c_2 \left(\frac{r}{\sigma}\right)^{2} + c_0 \right], \tag{1}$$

ただし、r は粒子間、σ は粒子直径、ε はエネルギースケールである。以下、ボルツマン定数 k_{B} を 1 とし、そのほかの物理量も σ、ε を用いて無次元化した単位系を取る。二つの追加係数 c_2, c_0 は、カットオフ距離 r_{c} においてポテンシャルと力がちょうどゼロになる、すなわち $V(r_{\mathrm{c}}) = V'(r_{\mathrm{c}}) = 0$ の条件から定められる。カットオフ距離は、ベンチマークで $r_{\mathrm{c}} = 2.5$、多重気泡核生成過程の計算では $r_{\mathrm{c}} = 3.0$ とした。短距離相互作用 MD シミュレーションのアルゴリズムとして、グリッド探索 [2, 3, 4, 5](もしくはリンクリスト探索 [6, 7]) により相互作用粒子を探索し、ペアリスト構築後は Bookkeeping 法 [8] によりリストの再利用を行い、リストの寿命判定に dynamic upper time cutoff (DUTC) 法 [9] を拡張したものを採用した。詳細については文献 [10] にまとめたので、ここではいくつかのポイントのみ述べる。

メモリアクセスの最適化のため、相互作用する粒子のペアを探索した後に粒子についてソートを行っている。後の便利のため、相互作用している粒子対のうち番号の若い方を Key 粒子、その相手を Partner 粒子と呼ぼう。なお、これはいわゆる i 粒子、j 粒子に対応する。Key 粒子によりソートをすると、同じ粒子と相互作用する粒子がまとめられる。これは、Key 粒子の情報をレジスタに載せることで、必要メモリバンド幅を削減することと、近い番号の Key 粒子と相互作用する Partner 粒子には重なりが多いことによるキャッシュ効率の向上を目的としている。

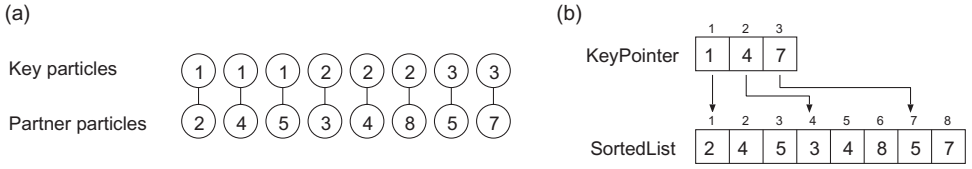

図 1: (a) Key 粒子の粒子番号によりソートしたペアリスト。(b) Partner 粒子のデータフォーマット。ひとつの一次元配列で Partner 粒子の番号を全て格納し、どこからどこまでがどの Key 粒子と相互作用をしているのかをポインタで表現する。例えば、i 番目の Key 粒子と相互作用する最初の Partner 粒子は SortedList[KeyPointer[i]] に、その次の Partner 粒子は SortedList[KeyPointer[i]+1] に格納されている。

並列化アルゴリズムは、単純空間分割法を採用する。また、ロードバランスなどは考慮せず、全ての領域の体積を同じにする。空間分割法は、相互作用距離にくらべて分割された領域が十分大きければ通信時間がほぼ無視できるため、大規模短距離 MD に向いた方法である。通信は基本的には隣接する領域でしか発生しないが、ペアリストの有効期限判定には注意を要する。それぞれの領域において、独立したペアリストを保持するが、そのリストの有効期限は領域により異なる。また、一般にリストの再構築は時間積分 1 ステップに比べて数倍から 10 倍以上時間がかかる。従って、それぞれが独立にペアリストの再構築を行うと、再構築中は別のプロセスからの通信要求に答えることができず、その間は全てのプロセスが待たされてしまう (図 2 (a))。そこで、どこか一つでもペアリストの有効期限が切れたら、他の全てのプロセスのペアリストも同時に再構築を行うことにする (図 2 (b))。これにより、ペアリストの平均再利用回数は同期しない場合よりも少なくなるが、ペアリスト再構築中のアイドル時間がなくなるため、

全体としては効率が良くなる。これは MPI_Allreduce で整数を一つ送ることで実装しており、時間積分の 1 ステップに比べて通信時間はほぼ無視できるが、全体が毎ステップ同期する必要があるため、ロードインバランスの影響を強く受けることになる。

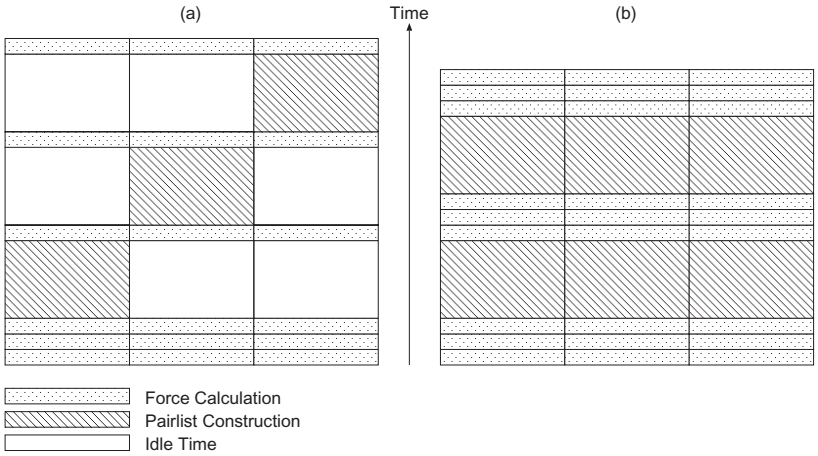

図 2: (a) ペアリストの同期を行わなかった場合。あるプロセスがペアリストの再構築中に別のプロセスが待たされる。これが繰り返されることにより、全体のアイドル時間が増加してしまう。(b) ペアリストの同期を行った場合。どこかひとつでもペアリストの有効期限が切れたら、全てのプロセスのペアリストを同時に再構築する。ペアリストの平均再利用回数は減るが、アイドル時間はなくなる。

2.2. 並列化の実装

並列化は、flat-MPI とするか、MPI/OpenMP のハイブリッドとするかの選択がある。短距離古典 MD においては、ハイブリッドの flat-MPI の方が計算が速い場合が多いが、flat-MPI はメモリ消費が大きく、特に大規模計算においてプロダクトランに必要なメモリ確保が難しくなる場合がある。そこで我々は、MPI/OpenMP のハイブリッドコードを開発した。ハイブリッド並列化を行う際、一般にはノード間を MPI を用いた空間分割、ノード内を OpenMP を用いたループ分割にすることが多いが、今回我々はノード間、ノード内のどちらも空間分割とする、疑似 flat-MPI 法を採用した。これは、OpenMP のスレッド番号を MPI のランク番号のように扱う事で実現する。疑似 flat-MPI 法は、共有メモリを分散メモリとして扱う点で flat-MPI と同様な計算手法であるが、MPI のプロセス数が減る分、消費メモリの低減が期待できる。しかし、環境によってはスレッド間の MPI 通信ができないか、制限を受けることが多いため、通信は一度プロセス単位でまとめて行い、再度スレッドに分配する必要がある (図 3)。全ての物理量の観測ルーチンを、この二段階通信にあわせて用意しなければならないのが、ループ分割によるハイブリッド並列に比べた、疑似 flat-MPI による方法の大きなデメリットである。しかし、ループ分割による方法では、ペアリスト作成などもループ分割しなければ全体の性能が向上せず、かつデータの排他制御も正しく扱わなければならない。何より、計算のホットスポットである力の計算で、OpenMP 並列と SIMD 化を同時に考慮しなければならない点がコードの最適化の大きな障害となる。疑似 flat-MPI では、OpenMP 並列が上位の階層で済んでいるため、力の計算はシングルコアの最適化のみ考慮すれば良い。

既に flat-MPI のコードがあるのなら、疑似 flat-MPI コードを実装するのは比較的容易である。具体的には、計算を行う MDUnit クラスと、通信を行う MDManager クラスにより実装している (開発言語に C++ 言語を用いている)。MDMangaer クラスにおいて、MDUnit クラス

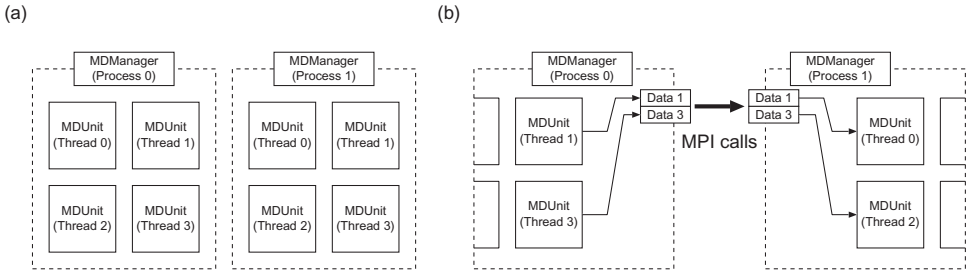

図 3: 疑似 flat-MPI 法。見やすさのため、二次元系で表示してある。2 つの MPI プロセスが、それぞれ 4 つの OpenMP スレッドを持っている。(a) MDUnit と MDManager がそれぞれ MPI プロセスと OpenMP スレッドに対応する。計算領域は MDUnit に割り当てられ、時間積分を行う。MDManager は MDUnit 間の通信を管理する。(b) プロセス間通信。環境によってはスレッドからの MPI 通信が許可されていないか制限を受けるため、一度プロセス単位でデータをまとめてからプロセス間で通信し、その後スレッドに分配する。

List 1: MDUnit クラスの初期化

```
int tid;
MDUnit *mdp;
std::vector <MDUnit *> v;
#pragma omp parallel shared(v) private(tid,mdp)
{
  tid = omp_get_thread_num();
  mdp = new MDUnit();
  #pragma omp critical
  v.push_back(mdp);
}
mdv.resize(num_threads);
for(unsigned int i = 0; i < v.size(); i++) {
  mdv[v[i]->ID] = v[i];
}
```

の初期化を行うサンプルコードは List 1 の通りである。ここで、MDUnit の vector である mdv は MDManager のメンバ変数であるが、OpenMP の制約でクラスのメンバ変数を shared 指定できないため、一度テンポラリ領域に確保してからソートしてコピーしている。MDUnit へのポインタは全てマスタースレッドに確保されるが、実際に計算に使われる配列は MDUnit 内で確保されるため、NUMA 最適化は自動的に満たされている。スレッド数は、MDManager クラスが保持する MDUnit クラスの数である。MDUnit がひとつしかなければ flat-MPI、複数あればハイブリッド実行となる。OpenMP 並列化、例えばたとえば力計算の並列化は以下の様になされる。

```
#pragma omp parallel for schedule(static)
for(int i = 0; i < num_threads; i++) {
  mdv[i]->CalculateForce();
}
```

すなわち、flat-MPI と疑似 flat-MPI の違いは、MDUnit クラスのメソッドが MPI プロセスから呼ばれるか、OpenMP スレッドから呼ばれるかだけしかない。従って、適切な領域配置を行えば、flat-MPI とハイブリッドで (足しあげの順序等も含めて) 全く同じ計算を実行することができる。

2.3. FX10 向けのチューニング

今回の計算に向け、力の計算ルーチンの FX10 向けのチューニングを行った。要点は、作用反作用を使わないことによるソフトウェアパイプライニングの促進、SIMD 化、逆数近似命令の利用とその精度補正である。準備的計算により、Intel Xeon などのアーキテクチャに比べて、FX10 での実行効率が悪いことがわかっていたが、その理由の一部は間接参照を伴う運動量の書き戻しにあることがわかった。通常、力の計算を行う際、作用反作用を用いて、i 粒子から j 粒子への力積を計算した後、i 粒子だけでなく、j 粒子の運動量にもその力積を反映させる。しかし、i 粒子と相互作用する j 粒子番号は間接参照となっており、これがコンパイラによるソフトウェアパイプライニングを阻害していただけなく、store 命令そのもののレイテンシが強く見えているようであった。そこで、作用反作用を使わないことで間接参照を伴う運動量の書き戻しを削除した。これにより計算量は二倍となるが、速度向上は二倍以上となり、トータルで計算速度が大きく向上した。また、コンパイラが効率的に SIMD 命令を出力できていないようだったため、intrinsic 命令により SIMD 命令を明示的に記述した。基本的にはループを二倍展開することで独立な命令を作る、自明 SIMD 化を行ったが、除算の SIMD 命令が見つからなかったため、代わりに逆数近似命令を利用した。逆数近似命令 (frcpad) は、高速に逆数を計算できる代わりに、その相対誤差は 8bit、すなわち最大で 1/256 程度に達する。通常、コンパイラがこの命令を出力する場合は、精度補正命令も同時に発行してくれるのだが、明示的に frcpad を実行したため、精度補正も明示的に行う必要がある。具体的には、以下のような手順で、二次の補正を行った。粒子間距離を r としよう。実際に計算するのはその二乗、r^2 である。その逆数近似値を $\tilde{r}^{-2}$ とする。すなわち、$\tilde{r}^{-2} = \mathrm{frcpad}(r^2)$ である。逆数近似と、倍精度で計算した場合の精度差、すなわちエラー ε は以下で表される。

$$\tilde{r}^{-2}r^2 = 1 + \varepsilon, \qquad (|\varepsilon| < 1/256) \tag{2}$$

力の計算には r^{-8}、および r^{-14} の計算が必要となるが、以下に r^{-8} の精度補正を示す。式 (2) を以下のように変形する。

$$\begin{aligned} r^{-8} &= \frac{(\tilde{r}^{-2})^4}{(1+\varepsilon)^4} && (3)\\ &= (\tilde{r}^{-2})^4(1 - 4\varepsilon + O(\varepsilon^2)) && (4)\\ &= (\tilde{r}^{-2})^4(5 - 4(1+\varepsilon) + O(\varepsilon^2)) && (5)\\ &= (\tilde{r}^{-2})^4(5 - 4\tilde{r}^{-2}r^2 + O(\varepsilon^2)). && (6) \end{aligned}$$

すなわち、r^{-8} の逆数近似値 $(\tilde{r}^{-2})^4$ に対し、$(\tilde{r}^{-2})^4(5 - 4\tilde{r}^{-2}r^2)$ と補正項をかけてやれば、その精度は $O(\varepsilon^2)$ となる。同様に、r^{-14} への補正項は $(8 - 7\tilde{r}^{-2}r^2)$ と求められる。どの程度の精度を行う必要があるかは現象により異なるが、今回の計算では二次の補正で十分であることを確認している。実際のコードではさらに二倍に展開し、一部のソフトウェアパイプライニングを手で行う事で、コンパイラによるソフトウェアパイプライニングを促進している。最終的にループは 4 倍展開されているが、2 倍が自明 SIMD 化のために、2 倍がソフトウェアパイプライニングのために行われている。

3. ベンチマーク

作成したコードを用いて、ベンチマークシミュレーションを行った。計算条件は以下の通りである。

- コアに割り当てる領域のサイズは $100 \times 100 \times 100$ の立方体。
- 数密度は 0.5、すなわちコアあたり 50 万粒子、ノードあたり 800 万粒子。

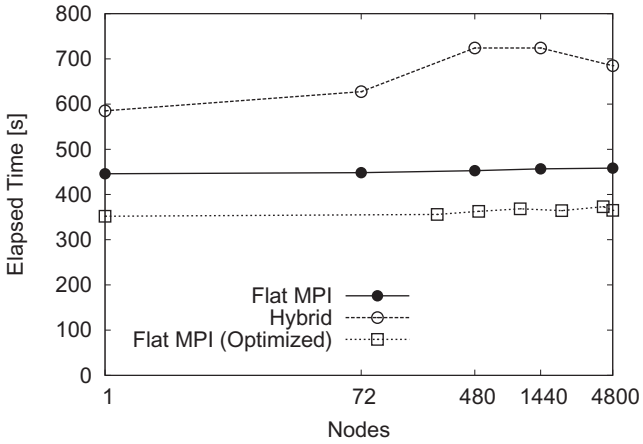

図 4: ベンチマーク結果。1000 ステップの時間積分にかかった時間。Intel Xeon 向けのコードの flat-MPI、ハイブリッド実行の結果、及び FX10 向けコードの flat-MPI の結果。

- 初期条件は FCC 格子とし、全方向に周期的境界条件。
- 時間積分は二次のシンプレクティック積分 (陽解法)。
- 時間刻みは 0.001。カットオフは 2.5。
- 初期速度は絶対値 0.9 とし、方向はランダム。
- 150 ステップのあと、1000 ステップの時間積分にかかった時間を計測。
- ノード数あたりの粒子数を固定したままノード数を増やす、ウィークスケーリング解析。

計算速度は MUPS (millions of updates per second) という単位を用いて定義する。これは百万粒子を 1 秒間で 1 ステップ計算できたら 1 となる単位である。flat-MPI と、ハイブリッド両方の計算を行った。ハイブリッド計算では、ノードあたり 16 スレッドとした。flat-MPI の計算は、Intel Xeon 向けのチューニングを施したコードと、FX10 向けのチューニングを施したコードの両方について、ハイブリッドは Intel Xeon 向けのチューニングを施したコードのみについて計算を実行した[1]。それぞれの実行結果を表 1、表 2、表 3、及び図 4 にまとめた。最大で 4800 ノード、76800 プロセス/スレッドの計算で、最大 384 億粒子の計算を行った。flat-MPI のスケーリングはほぼ完璧であるにもかかわらず、ハイブリット実行の結果は揺らぎが大きく、かつ flat-MPI の計算より二倍近く遅い。また、FX10 向けのチューニングを行ったコードは、Intel Xeon 向けのチューニングを行ったコードを FX10 で実行した場合に比べて 25%程度速度が向上している。なお、4800 ノードの実行においてノードあたりの消費メモリは flat-MPI の場合で 14GB、hybrid 実行で 10GB であり、ほぼ同じ計算を行っているにもかかわらず、flat-MPI の方が 4GB 余計にメモリを消費していた。

最大の計算 (4800 ノード) における計算効率は、Intel Xeon 向けコード+flat-MPI で 56.5 TFLOPS (ピーク性能比 5%)、FX10 向けコード+flat-MPI で 193 TFLOPS (ピーク性能比 17%) であった。FX10 向けのコードは作用反作用を利用していないために FLOPS の絶対値には意味がないが、二倍の計算量を、二倍以上の効率で計算していることから、トータルの計算実行速度は 25%程度向上している。

[1]FX10 向けのチューニングではスレッドローカルな配列を利用する必要があったが、そのハイブリッド版の用意が実行に間に合わなかったため。

ノード数	粒子数	実行時間 [s]	速度 [MUPS]	並列化効率
1	8,000,000	445.815	17.9447	1.0
72	576,000,000	448.275	1284.93	0.99
480	3,840,000,000	452.781	8480.93	0.98
1440	11,520,000,000	456.641	25227.7	0.98
4800	38,400,000,000	458.399	83769.9	0.97

表 1: Intel Xeon 向けチューニング+flat-MPI の実行結果。並列化効率はシングルノードの実行時間と比較した値。

ノード数	粒子数	実行時間 [s]	速度 [MUPS]	並列化効率
1	8,000,000	585.160	13.6715	1.00
72	576000000	627.449	918.003	0.93
480	3840000000	724.139	5209.34	0.79
1440	11520000000	724.139	15908.5	0.81
4800	38,400,000,000	684.871	56068.9	0.85

表 2: Intel Xeon 向けチューニング+ハイブリッドの実行結果。

4. 多重気泡生成シミュレーション

4.1. 多重気泡生成

気液混相流の全粒子計算への第一歩として、急減圧による多重気泡生成シミュレーションを行った。液相からの気泡生成には、温度上昇による沸騰 (boiling) と、減圧によるキャビテーション (cavitation) に区別される。キャビテーションは船のスクリューの腐食などに関与する。シミュレーションで実現が容易なのは、定積条件下において急減圧することである。これは、ある瞬間に系を一様膨張させることで実現される。減圧の度合いが弱いと、系は準安定状態となり、一定時間後に気泡が生成するが、減圧の度合いが強い場合には、系はスピノーダル分解を起こし、減圧後直ちに多数の気泡が生成する。今回は、強い急減圧を行う事で多重気泡生成を再現する。多重気泡生成では、初期ステージにおいて多数の気泡が生成した後、圧力を介した気泡間相互作用により、大きな気泡がより大きく、小さい気泡がより小さくなる Ostwald 的成長を起こす。その後、十分な時間の後に単一気泡へと収束する。同様な現象に、液滴核生成がある。これは純粋な気相状態から条件を変えることで過飽和状態とし、液滴が成長する過程であ

ノード数	粒子数	実行時間 [s]	速度 [MUPS]	並列化効率
1	8,000,000	351.821	22.7389	1.00
256	2,048,000,000	355.829	5755.57	0.99
512	4,096,000,000	362.775	11290.7	0.97
1024	8,192,000,000	368.275	22244.2	0.96
2048	16,384,000,000	364.272	44977.4	0.97
4096	32,768,000,000	373.007	87848.3	0.94
4800	38,400,000,000	364.920	105229.0	0.96

表 3: FX10 向けチューニング+Flat-MPI の実行結果。

る。どちらも気液相転移による緩和過程であるが、まわりが液体であるが故に気泡間相互作用は液滴間相互作用よりも強く、かつ気泡が成長する際に液体にする仕事を考慮しなければならないなど、気泡生成過程の方が液滴生成過程よりも考慮すべき項目が多く、解析が難しくなる。

4.2. 計算の詳細

まず平衡状態にある液相を用意する。これは、相図上で気液共存線に近い液相 (温度 0.9、密度 0.7) において、NVT シミュレーションを行うことで実現する。温度制御には Nose-Hoover 法を用いた [11]。温度を制御している間の時間積分は、二次の指数分解法 (reversible system propagator algorithm, RESPA) を用いた [12]。カットオフ距離は 3.0、時間刻みはシミュレーションを通して 0.005 とした。系の大きさはコアあたり一辺 30 の立方体とし、全体の大きさは $1440 \times 1200 \times 1200$、数密度が 0.7 であることから粒子数は 14 億 5 千万である。温度制御付きで 10000 ステップ計算して緩和させた後、密度が 0.65 となるように系を一様膨張させ、以後は温度制御をせず NVE シミュレーションにより時間発展を行った。計算は緩和も含めて 17800 ステップ実行し、減圧後は 1000 ステップに一度、以下に述べる局所密度情報を保存した。なお、計算は flat-MPI ではメモリ不足で失敗したため、ノードあたり 1MPI プロセス ×16 OpenMP スレッドのハイブリッドで実行した。実行時間は 15 時間であった。

気相、液相の判定のため、空間を小さなセルに区切り、一定密度以下であるセルを気相、それ以上の密度を持つ系を液相と定義した [13]。隣接する気相セルは同じ気泡と定義し、その体積を計算した。今回の計算ではセルの大きさを一辺 3.0 の立方体とし、気相判定の密度の閾値を 0.2 とした。なお、気泡判定アルゴリズムは原理的にパーコレーションにおけるクラスタリングアルゴリズムと同じものであるが、これを全系で行うと FFT 型の全体通信が必要となり、通信路の輻輳により極めて時間がかかる。そこで、各プロセスにおいて局所密度情報を計算した後、その情報をルートノードに集約し、クラスタリング処理はルートノードのみで行うことにした。各コアの担当領域が一辺 30 の立方体であるため、各コアあたり 1000 個のセルを持つ。もし密度を倍精度実数で表現すると全体で 614MB の情報が必要となるが、通信量、およびスナップショットのデータ量削減のため、それぞれのセルについて何個の粒子が含まれるかを 1 バイト (unsigned char) で表現することとした。各セルの体積が 27 であり、液相密度が 0.7、液相がほぼ非圧縮であることから局所数密度が 1 を超える可能性は極めて低く、密度情報を精度を全く失うことなく保存可能である。これにより、密度スナップショットに必要な容量は 76.8MB となった。スナップショットは 169 回保存したため、計算全体で利用したストレージは 13GB 程度であった。

4.3. 計算結果

図 5 に圧力の時間発展の様子を、図 6 に気泡の可視化のスナップショットを示す。液相状態で緩和させたあと $t = 50$ において急減圧を行った結果、系は負圧となる。減圧直後から気泡が生成することで系の圧力が上昇していく様子がわかる。また、最初は気泡が多数生成するが、気泡成長にともない圧力が上がることでその圧力下で成長できる気泡、できない気泡にわかれる。成長できない気泡は圧力に負けてつぶされていき、大きな気泡はより大きく、小さな気泡はより小さくなっていく (Ostwald 成長)。

系に存在する気泡の数の時間発展を図 7 (a) に示す。減圧後に急激に増えた気泡は、途中から減少に転じる。これは気泡成長により圧力が上昇し、新たな気泡生成が抑制された上に、小さな気泡がどんどんつぶされて減って行くことにより、大きな気泡がより大きく成長していることによる。減圧後、1 万ステップ後の気泡のサイズ分布関数を図 7 (b) に示す。分布関数が十分な精度で観測できていることがわかる。

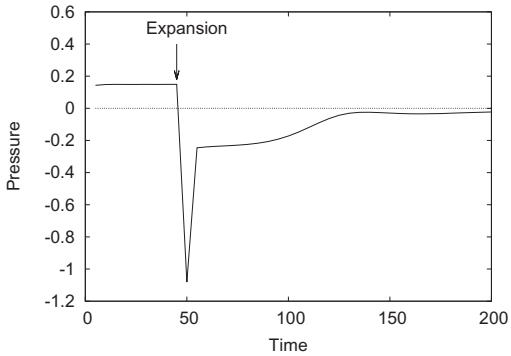

図 5: 圧力の時間発展。最初は純粋な液相 (温度 0.9、密度 0.7) に保たれているが、$t = 50$ において急減圧を行った。圧力は一度負圧となり、その後気泡生成により徐々に上昇する。

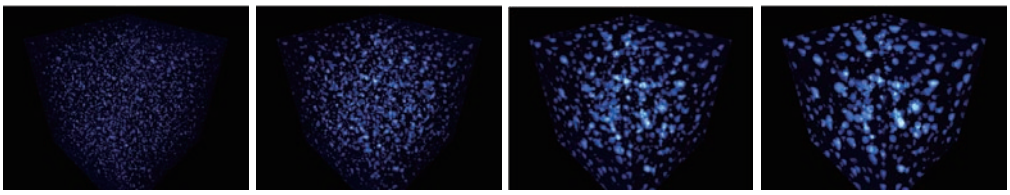

図 6: 気泡生成過程。左から右に時間発展する。減圧直後から多数の気泡が発生し、その後 Ostwald 的成長が見られる。なお、可視化には POV-Ray を用いた [14]。

4.4. ロードインバランス

多重気泡生成シミュレーションでは、減圧後に気泡が成長する過程で密度揺らぎが大きくなる。気泡が領域をまたぐレベルに成長すると、各コアの担当する領域が気相のみ、液相のみとなる場合があり、大きなロードインバランスが発生する。毎ステップ、グローバルな同期を取っているため、ロードインバランスは全体の計算効率に直結する。ここではロードインバランスによる計算速度低下の程度を見積もる。短距離 MD において、計算負荷は大雑把に粒子数に比例する。今回は全ての分割領域の体積が等しいため、計算負荷は密度に比例する。理想的なロードバランスが達成された場合、計算負荷は平均密度 ρ_{ave} に比例する。また、気泡生成が始まったあと、最も計算負荷が重いのは、領域内がすべて液相となる場合である。液相はほぼ非圧縮であることを考慮すると、もっとも計算負荷が重い領域の計算負荷は液相密度 ρ_{l} に比例し、これが全体の計算速度を決める。今回の計算では、初期密度 0.7 から 0.65 に減圧したため、平均密度 $\rho_{\mathrm{ave}} = 0.65$ である。また、減圧後の温度は 0.88 となった。この温度での気液共存密度は、気相が $\rho_{\mathrm{g}} = 0.0344$、液相が $\rho_{\mathrm{l}} = 0.687$ である [15]。ロードインバランスによる性能劣化は、平均密度 ρ_{ave} と液相密度 ρ_{l} で決まるため、今回の計算では $1 - \rho_{\mathrm{ave}}/\rho_{\mathrm{l}} \sim 0.05$、すなわち 5%程度しか劣化しない。これは気泡が成長しきった、最終ステージでの最悪の値であるため、実際の計算ではロードインバランスはほとんど問題とならない。気相と液相の共存密度の比は $\rho_{\mathrm{l}}/\rho_{\mathrm{g}} \sim 20$ であるため、計算が軽い領域と重い領域の計算負荷の差は最大で 20 倍近くになるが、計算が軽くなった領域の分の計算負荷を、多数の別の領域が分担することで全体の計算時間はほとんど増えていないという構図になっている。ただし、これは減圧シミュレーションによる特徴であるため、例えば過飽和気相から液滴生成を計算する場合にはロードインバランスは極めて深刻な問題となる。

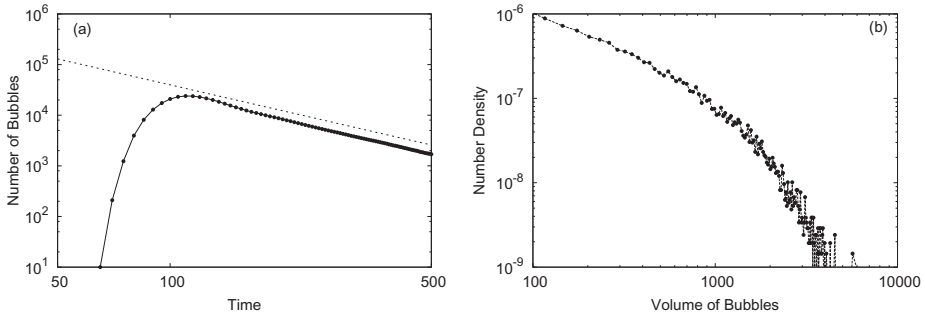

図 7: (a) 気泡数の時間発展 (両対数プロット)。点線は傾き-1.7。(b) $t = 100$ (減圧後 1 万ステップ後) の気泡数密度分布関数 (両対数プロット)。

5. ハイブリッド実行時の速度向上

HPC チャレンジ後、ハイブリッド実行時の速度が flat-MPI よりも遅い件について富士通側に調査を依頼したところ、STL の std::vector が原因であることが判明した。具体的には、隣接粒子探索をするルーチンにおいて std::vector を利用していたのだが、そこがマルチスレッド実行で遅くなっていた。問題となった std::vector は以下のように関数内のテンポラリバッファとして用いられていた。

```
void
SomeClass::SomeFunction(void){
  std::vector<int> v;
  // v を用いた処理
}
```

ただし、SomeFunction はそれぞれのスレッドから独立に実行される関数であり、この領域では排他制御の必要はない。この変数宣言を以下のようにスタックからヒープに確保した上で、スレッドローカルな変数となるように変更した。

```
void
SomeClass::SomeFunction(void){
  static __thread std::vector<int> v;
  // v を用いた処理
}
```

なお、変更箇所は一カ所のみである。以上の変更による実行速度の変化を表 4 及び図 8 にまとめる。1 ノードの計算で、flat-MPI の場合は 16 プロセス、ハイブリッドの計算は 1 MPI プロセス 16 OpenMP スレッド、粒子数は 100 万粒子 (コアあたり 6 万 2500 粒子)。1000 ステップ計算するのにかかった時間。flat-MPI では、vector をスタックに取ってもヒープに取っても実行速度に影響はなかった。しかし、1 MPI プロセス、16 OpenMP スレッド実行をした場合では、vector をスタックに取ると flat-MPI に比べて 40%程度遅くなり、ヒープに取ると flat-MPI と遜色ない実行速度となった。

6. まとめ

疑似 flat-MPI による並列化がなされた短距離古典分子動力学法コードを開発し、最大 4800 ノード、384 億粒子の計算を実行、193 TFLOPS の性能 (ピーク性能比 17%) を達成した。FX10 向けのチューニングを行うことで、Intel Xeon 向けのコードを利用した場合よりも 25%程度の性能向上が得られた。既にキャッシュ効率の最適化が進んでいるコードであったため、最適化による速度向上はさほど大きいとは言えないが、同様な計算を Intel Xeon と実行した場合、コ

	実行時間 [s]
flat-MPI (stack)	47.73
flat-MPI (heap)	47.28
hybrid (stack)	65.29
hybrid (heap)	47.78

表 4: flat-MPI、hybrid のそれぞれについて、vector をスタックに取った場合 (変数宣言無指定) と、ヒープに取った場合 (static 指定)。

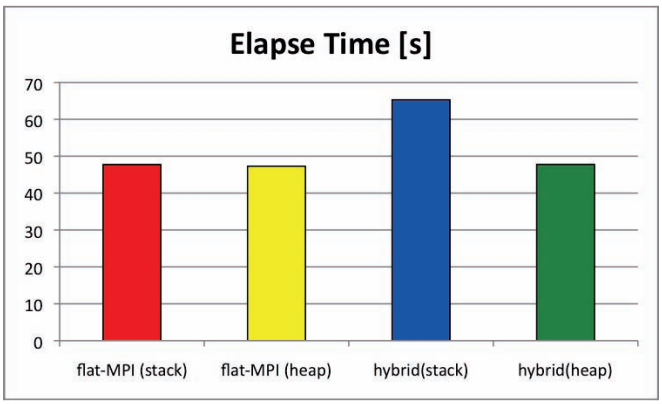

図 8: flat-MPI、hybrid のそれぞれについて、vector をスタックに取った場合 (変数宣言無指定) と、ヒープに取った場合 (static 指定)。

アあたりの性能は FX10 の方が 2 倍近く低く [10]、メモリバンド幅、キャッシュヒット率、力の計算アルゴリズム等から予想される理論ピーク性能に比べて実行性能が低いため、まだ性能向上の余地があると思われる。

また、flat-MPI 実行とハイブリッド実行の比較も行った。コアあたりの計算量及びノード間通信のデータ量は同じまま、ノード間通信回数はむしろハイブリッド実行の方が少ないにもかかわらず、flat-MPI の方がハイブリッド実行よりも二倍近く速かった。その後の調査により、std::vector に問題があることが判明し、コードを修正することで現在では flat-MPI と遜色のない実行速度が得られている。並列化のスケーリングに関しては、1 ノード実行を基準とした 4800 ノード実行の並列化効率は 96%と、計算規模を考えれば極めて良好であった。他の計算資源においては、システム由来と思われるノイズにより、大規模計算実行時に並列化効率の劣化が見られたが、本システムにおいてはシステム由来のノイズはほぼ観測されなかった。これは FX10 のシステムノイズ対策が効果的になされていることを示唆する。

FX10 の 4800 ノード全体を用いたプロダクトランとして、14 億粒子を用いた多重気泡生成シミュレーションを行った。これは気液相転移を伴う粒子計算としては (筆者の知る限りにおいて) 世界最大の計算である。粒子間相互作用の結果として気泡が生成し、気泡間相互作用の結果として Ostwald 成長が観測された。これは、少なくとも二階層にわたるマルチスケール構造が、ミクロなモデルから直接計算できたことを意味する。また、数億粒子を超える計算を行うと、気泡の数が 1 万を超えるため、気泡分布関数の時間発展を十分な精度で追うことができる。これにより、ミクロな相互作用から推定した分布関数の時間発展と直接比較することが可能となった。今後は、気泡間相互作用の直接推定や、系内の圧力分布の解析などにより、多数の気泡が存在する系の物性を解明し、気液混相流の解析へとつなげていきたい。

なお、本研究に用いたコードはオープンソースとして公開しているため、興味のあるかたは参照されたい [16]。

謝辞

本研究は King Abdullah University of Science and Technology (KAUST) による Global Research Partnership (Award No. KUK-I1-005-04)、及び科研費 若手研究 (B) (課題番号: 23740287) の支援を受けて行われました。

参 考 文 献

[1] S. D. Stoddard, and J. Ford, Phys. Rev. A **8**, 1504 (1973).

[2] M. P. Allen and D. J. Tildesley, Computer Simulation of Liquids (Clarendon Press, Oxford, 1987).

[3] D. E. Knuth, The Art of Computer Programming (Addison–Wesley, Reading, MA, 1973).

[4] G. S. Grest and B. Dünweg, Comput. Phys. Commun. **55** (1989) 269 – 285.

[5] D. M. Beazley and P. S. Lomdahl, Parallel Comput. **20** (1994) 173 – 195.

[6] B. Quentrec and C. Brot, J. Comput. Phys. **13** (1975) 430 – 432.

[7] R. W. Hockney and J. W. Eastwood, Computer Simulation Using Particles (McGraw-Hill, New York, 1981).

[8] L. Verlet, Phys. Rev. **159** (1967) 98 – 103.

[9] M. Isobe, Int. Mod. Phys. C **10** (1999) 1281–1293.

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

[11] W. G. Hoover, Phys. Rev. A **31** (1985) 1695 – 1697.

[12] M. Tuckerman, B. J. Berne, and G. J. Martyna, J. Chem. Phys. **97** (1992) 1990 – 2001.

[13] H. Watanabe, M. Suzuki, and N. Ito, Phys. Rev. E **82** (2010) 051604.

[14] ⟨http://www.povray.org⟩

[15] H. Watanabe, N. Ito, and C.-K. Hu J. Chem. Phys. **136** (2012) 204102.

[16] コードは MDACP という名前で公開している。今回の計算で用いたバージョンは 2.00。最新版は以下の URL から取得可能。⟨http://mdacp.sourceforge.net⟩.