連載講座: 高生産並列言語を使いこなす(5)
分子動力学シミュレーション

田浦健次朗

東京大学大学院情報理工学系研究科, 情報基盤センター

目次

1 問題の定義

今回は古典的な分子動力学法シミュレーションを取り上げる [1, 2, 4, 5]. 分子間の相互作用として, 以下の 2 分子間の Lenard Jones ポテンシャルを仮定する.

$$U(r) = 4\epsilon \left\{ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right\}. \tag{1}$$

ここで, ϵ, σ は定数で, r は分子間の距離である. 分子 A が分子 B に及ぼす力は, $\vec{\mathrm{AB}} = \vec{r}$ として,

$$-\nabla U(r) = 24\epsilon \left\{ 2\frac{\sigma^{12}}{r^{14}} - \frac{\sigma^{6}}{r^{8}} \right\} \vec{r} \tag{2}$$

となる. $\{\ldots\}$ 内は r が小さい時に正で, r が大きいと負である. つまり, 力は r が小さい時に斥力, r が大きいと引力になる. 特に, r が大きくなると急激に 0 に近づく. 特にその絶対値が r^{-7} に比例して減衰するので, ほぼ一様な分布では, ある距離以上の分子からの力の寄与は, それらすべてを合計しても, 急激に小さくなる. そのため Lenard Jones ポテンシャルによる相互作用を計算する際は, 適当なカットオフ半径よりも離れた分子間の力は 0 として, 計算量を節約できる.

与えられた分子の初期配置・速度から, 式 (2) で与えられる力に従って, 分子の速度, 位置を更新していくのが解くべき問題である.

また, 周期的境界条件を仮定する. つまりある bounding box (シミュレーション領域) を設けてその中の分子に対してのみ計算を行うが, 実際の分子配置はその bounding box 内の分子配置が繰り返されているものとして計算を行う.

2 逐次プログラム

2.1 分子 (粒子)

各分子は, 質量, 速度, 位置, 加速度を保持する構造体として表現する.

```
typedef struct particle {
  int idx;
  double m;                    /* 質量 */
  double x[3];                 /* 位置 */
  double v[3];                 /* 速度 */
  double f[3];                 /* 力 */
} particle, * particle_t;
```

2.2 セル

素朴な計算方法では分子数 N に対して N^2 に比例する計算量が必要になる. 前述したとおり, 適当なカットオフ半径よりも離れた分子間の力は 0 としてよいが, そもそも分子間の距離を全分子対に対して行っていたのでは, N^2 の計算量は免れない.

実際に計算量を $O(N)$ にするためには, カットオフ半径よりも遠い分子に対して, 分子間の距離を計算せずに済ませる工夫が必要である. そのために空間を**セル**と呼ばれる立方体の箱に分割し, 各セルに, そのセル内の分子を格納する [2].

セルの大きさ (辺の長さ) は原理的には任意だが, 辺の長さをカットオフ半径以上とすることで, あるセル内の分子に計算すべき力を及ぼす分子は, そのセルおよび隣接するセル内にのみ存在するということになる. ここではカットオフ半径を 3σ, セルの一辺の長さを 3.2σ としている.

各セル内の分子は, 上記で定義した構造体 particle の配列に保持する. 全分子数の合計は一定だが, 一つのセル中に含まれる分子の数は, 分子が移動することによって変わりうるため, 各セルごとに別々の配列としている.

```
/* セルを表す構造体 */
typedef struct cell {
  particle_array particles_in_cell; /* セル内の分子 */
} cell, * cell_t;
```

ここで, particle_array は必要に応じて伸長可能な配列で, particle の配列 (ポインタ), そのサイズと, 現在の要素数を保持する.

```
typedef struct particle_array {
  int n;                        /* a の現在の要素数 */
  particle * a;                 /* 分子の配列 */
  int sz;                       /* a の容量 */
} particle_array, * particle_array_t;
```

2.3 系の状態

系の状態をすべてまとめたデータ構造が以下の config で, 実質的には上記で定義した cell の配列である. cell は, 概念的には xyz 方向の 3 次元配列だが, C のデータ構造としては一次元配列として表している.

```
typedef struct config {
  int step;                     /* 現在の時刻ステップ */
  int n_steps;                  /* シミュレートすべき時刻ステップ数 */
  int n_particles;              /* 分子数 */
  double side_len;              /* シミュレートする領域全体の一辺の長さ */
  int n_cells_on_side;          /* 一辺に沿ったセルの数 */
  double cell_len;              /* 一セルの一辺の長さ */
  cell_t cells;                 /* セルの配列 */
} config, * config_t;
```

2.4 1 ステップ

シミュレーションの一ステップは,

1. 力の計算 (calc_force_cells)
2. 速度, 位置の更新 (update_cells)
3. 分子のセル間の移動 (migrate_cells)

の 3 フェーズからなる.

```
void step(config_t g) {
  /* 力の計算 */
  double U = 0.5 * calc_force_cells(g);
  /* 速度位置の更新 */
  double K = update_cells(g);
  record_U_and_K(K, U)
  /* セル間を移動した分子を新しいセルへ移す */
  migrate_cells(g);
}
```

力の計算をする過程で全ポテンシャルを計算し, 速度, 位置の更新をする過程で, 運動エネルギーを計算している. 分子の軌跡を得るだけであれば必須の計算ではない.

2.5 力の計算

上記の calc_force_cell はすべての分子に対して力の計算をする関数で, 自然に以下のような, すべてのセルを走査する三重ループになる.

```
double calc_force_cells(config_t g) {
  int nos = g->n_cells_on_side;
  double U = 0.0;
  int i, j, k;
  /* 全セルについて, 力の計算 */
  for (i = 0; i < nos; i++) {
    for (j = 0; j < nos; j++) {
      for (k = 0; k < nos; k++) {
        U += calc_force_cell(g, i, j, k);
      }
    }
  }
  return U;
}
```

calc_force_cell(g, i, j, k) は, (i, j, k) 番目のセル内の分子の力を計算するもので, すべての近隣のセルとの相互作用を計算しつつ, ポテンシャルを計算する.

```
double calc_force_cell(config_t g, int i, int j, int k) {
  int nos = g->n_cells_on_side;
  cell_t c = g->cells[i*nos*nos + j*nos + k];
  double U = 0.0;
  int ii, jj, kk;
  int idx;

  if (c->particles_in_cell.n == 0) return 0.0;
```

```
    /* 自セル内の分子に働く力を 0 に初期化 */
    for (idx = 0; idx < c->particles_in_cell.n; idx++) {
      particle_t p = &c->particles_in_cell.a[idx];
      v_zero(p->f);
    }

    /* すべての近隣セルとの相互作用を計算 */
    for (ii = i - 1; ii <= i + 1; ii++) {
      for (jj = j - 1; jj <= j + 1; jj++) {
        for (kk = k - 1; kk <= k + 1; kk++) {
          U += calc_force_cell_cell(g, i, j, k, ii, jj, kk);
        }
      }
    }
    return U;
  }
```

calc_force_cell_cell(g, i, j, k, i', j', k') は, (i, j, k) 番目のセル内の分子全てに, (i', j', k') 番目のセルの分子全てが及ぼす力を計算する. 端のセルについては, 近隣セルがシミュレーション領域をはみ出すことがある. 適宜, 周期的境界条件を考慮して, 適切なセルとの間で相互作用を計算する.

2.6 速度と位置の更新

速度と位置の更新は以下の更新式で示される, leapfrog 法を用いて行う [5]. つまり実際には particle 構造体内の速度 v は, 時刻 $(t - \Delta t/2)$ における速度を保持している.

$$v(t + \Delta t/2) = v(t - \Delta t/2) + \frac{f}{m}\Delta t \quad (3)$$

$$x(t + \Delta t) = x(t) + v(t + \Delta t/2)\Delta t \quad (4)$$

update_cells の中身は, calc_force_cells 同様, 自然に全セルに渡る三重ループになる.

```
  double update_cells(config_t g) {
    int nos = g->n_cells_on_side;
    int i, j, k;
    double K = 0.0;
    for (i = 0; i < nos; i++) {
      for (j = 0; j < nos; j++) {
        for (k = 0; k < nos; k++) {
          K += update_cell(g, i, j, k);
        }
      }
    }
    return K;
  }
```

2.7 セル間の分子の移動

分子の位置が更新されたら, 必要に応じて分子を適切なセルに移し替える (migrate_cells). その全体の構造は, calc_force_cells や update_cells と全く同様の三重ループになる.

```
void migrate_cells(config_t g) {
  int nos = g->n_cells_on_side;
  int i, j, k;
  for (i = 0; i < nos; i++) {
    for (j = 0; j < nos; j++) {
      for (k = 0; k < nos; k++) {
        migrate_cell(g, i, j, k);
      }
    }
  }
}
```

migrate_cell(g, i, j, k) では, (i, j, k) 番のセル内の各分子について, 必要ならばその分子を移動 (migrate_particle) させる.

```
void migrate_cell(config_t g, int i, int j, int k) {
  int nos = g->n_cells_on_side;
  cell_t c = &g->cells[i*nos*nos + j*nos + k];
  int idx = 0;
  while (idx < c->particles_in_cell.n) {
    particle_t p = &c->particles_in_cell.a[idx];
    if (migrate_particle(g, p, i, j, k) == 0) {
      idx++;
    }
  }
}
```

migrate_particle では, 分子の位置から所属すべきセルを求め, それが (i, j, k) でなければ新しいセルへ移動する. つまり, セル (i, j, k) からセルを削除して, 新しいセル (i', j', k') へ挿入する.

```
int migrate_particle(config_t g, particle_t p, int i, int j, int k) {
  int nos = g->n_cells_on_side;
  int new_i, new_j, new_k;

  translate_to_inside(g, p);

  new_i = (int)floor(p->x[0] / g->cell_len) % nos;
  new_j = (int)floor(p->x[1] / g->cell_len) % nos;
  new_k = (int)floor(p->x[2] / g->cell_len) % nos;
  if (new_i < 0) new_i += nos;
  if (new_j < 0) new_j += nos;
```

```
  if (new_k < 0) new_k += nos;

  if (i == new_i && j == new_j && k == new_k) return 0;
  cell_t cell = &g->cells[i*nos*nos + j*nos + k];
  cell_t new_cell = &g->cells[new_i*nos*nos + new_j*nos + new_k];
  /* 新しいセルへ分子を挿入 */
  particle_array_add(&new_cell->particles_in_cell, p);
  /* 現セルから分子を削除 */
  particle_array_remove(&cell->particles_in_cell, p);
  return 1;
}
```

以降, 今号は OpenMP, Cilk, Intel TBB を用いた共有メモリ並列化を取り上げる.

3 OpenMP による並列化

各タイムステップを構成する 3 つの関数 (`calc_force_cells`, `update_cells`, `migrate_cells`) はどれも, OpenMP の parallel for プラグマを用いて, 全セルに対して並列実行できる. `calc_force_cells` であれば前述の三重ループに pragma を一行追加するだけで良い.

```
double calc_force_cells(config_t g) {
  int nos = g->n_cells_on_side;
  double U = 0.0;
  int i, j, k;
#pragma omp parallel for collapse(3) reduction(+:U)
  for (i = 0; i < nos; i++) {
    for (j = 0; j < nos; j++) {
      for (k = 0; k < nos; k++) {
        U += calc_force_cell(g, i, j, k);
      }
    }
  }
  return U;
}
```

collapse(3) は, pragma 以下にある 3 つのループ (完全入れ子ループであることが要求される) を, 分割実行することを指示している.

`update_cells` や `migrate_cells` もほぼ同様だが, `migrate_cells` の並列実行を行うと, ひとつのセルに対する分子の挿入や削除が並行に行われることになる. 今回は簡単のため, 可変長配列への挿入, 削除それぞれを, 排他制御で保護することで実現している.

なお, このように多重ループの並列化を行った際の, 繰り返しの分割は, まずすべての繰り返しを, 逐次実行する順番—この例では (`i`, `j`, `k`) の辞書順—に一列に並べ, それに対して通常のループに対するスケジューリングポリシーが適用される. 例えばデフォルトのスケジューリングポリシー (static) で, `nos` がスレッド数よりも十分大きければ, `i` のループだけを分割した場合と似た様な分割になる.

4 Cilk による並列化

本連載で用いている MIT 版の Cilk でサポートされている並列構文は spawn のみであり, 三重ループの並列化も再帰呼び出しに書き直す必要がある. つまり, シミュレーション空間 (セルの集合) を, x,y,z 軸方向に再帰的に 2 分割を繰り返していく. この際, 一番長い辺に沿って 2 分割を繰り返せば, 個々の CPU が担当する領域の形が立方体に近く保たれ (直交再帰 2 分割; Orthogonal Recursive Bisection), 体積/表面積の比を高く保つことができる. それは, 隣接セルのデータを必要とする計算では, 計算/通信比を高く保てるということを意味する.

pragma を挿入するだけで並列化できるのと比べると面倒ではあるが, どのような言語であろうと, 計算/通信比を高く保つために必要になることがある方法である.

このような空間分割の再帰呼び出しをきれいに書くためには, 3 次元の矩形領域を表すデータ構造を作れば良く, 実際 Chapel, X10, TBB などですでに備わっている. といっても自作も簡単で, 以下ではこれを部品として実現する.

```
/* 3 次元の区間 [i0,i1]x[j0,j1]x[k0,k1] */
typedef struct dom3 {
  int i0;
  int i1;
  int j0;
  int j1;
  int k0;
  int k1;
} dom3;
```

最も長い辺に沿って矩形 d を 2 分割し, 結果を p[0], p[1] に格納するのが以下の split_dom3 である. これ以上分割できない場合は 0 を返す.

```
int split_dom3(dom3 d, dom3 p[2]) {
  int i0 = d.i0, i1 = d.i1;
  int j0 = d.j0, j1 = d.j1;
  int k0 = d.k0, k1 = d.k1;
  if (i1 - i0 == 1 && j1 - j0 == 1 && k1 - k0 == 1) return 0;
  else {
    p[0] = p[1] = d;  /* d のコピーを二つ */
    /* どの辺が長いかに応じて中点で分割 */
    if (i1 - i0 >= j1 - j0 && i1 - i0 >= k1 - k0) {
      p[0].i1 = p[1].i0 = (i0 + i1) / 2;
    } else if (j1 - j0 >= k1 - k0) {
      p[0].j1 = p[1].j0 = (j0 + j1) / 2;
    } else {
      p[0].k1 = p[1].k0 = (k0 + k1) / 2;
    }
    return 1;
  }
}
```

これをもとに矩形領域 d を再帰的に分割して並列に力を計算する関数は以下のようになる.

```
cilk double calc_force_cells_aux(config_t g, dom3 d) {
  dom3 p[2];
  if (split_dom3(d, p) == 0) {
    /* これ以上分割できない (1 セル) */
    double K;
    spawn K = calc_force_cell(g, d.i0, d.j0, d.k0); sync;
    return K;
  } else {
    double K0, K1;
    K0 = spawn calc_force_cells_aux(g, p[0]);
    K1 = spawn calc_force_cells_aux(g, p[1]);
    sync;
    return K0 + K1;
  }
}
```

もちろん分割関数 split_dom3 は力の計算, 位置と速度の更新, 分子のセル間の移動すべてで再利用できる. あとは逐次プログラムや OpenMP プログラムの関数とインタフェースを統一するために, 領域全体への呼び出しを行う関数を書けば全体が完成する.

```
cilk double calc_force_cells(config_t g) {
  int nos = g->n_cells_on_side;
  dom3 d = { 0, nos, 0, nos, 0, nos };
  double K;
  K = spawn calc_force_cells_aux(g, d); sync;
  return K;
}
```

update_cells や, migrate_cells も全く同様である. なおもちろん, この書き方は Cilk に限った書き方ではなく, 前号のゲーム木探索問題のところで取り上げた OpenMP の task 構文, Intel TBB の taskgroup を用いても同様の記述が可能である.

なお, Intel Cilk Plus や, GCC 4.7 以降でサポートされる Cilk には, cilk_for という並列 for 構文があり, for 文を再帰的に 2 分割する構文が存在する. ただし cilk_for が三重にネストしたループは, 直交再帰分割になるわけではないので, 直交再帰分割を明示的に行うのにある程度自力での記述が必要なのは, 今のところどの言語でも同じのようである.

5 Intel TBB による並列化

Intel TBB には 3 次元までの矩形領域を表すデータ構造 (blocked_range, blocked_range2d, blocked_range3d) が備わっており, それに対して並列に操作を適用する, parallel_for, parallel_reduce という関数がある.

calc_force_cells は全ポテンシャルエネルギーを計算するので, 縮約 (reduction) が必要になる. そこで parallel_reduce を使う [3]. parallel_reduce の基本的なインタフェースは,

`parallel_reduce(範囲, 縮約オブジェクト);`

「範囲」としては `blocked_range`, `blocked_range2d`, `blocked_range3d` などのインスタンスを指定する. 「縮約オブジェクト」はユーザが定義する, 「縮約型 (reduction type)」のインスタンスである. 縮約型は以下のメソッドとシグナチャを備えたクラスであれば自由に定義できる. 以下ではその型の名前を C とする.

通常のコンストラクタ:

splitting コンストラクタ: シグナチャは,

`C(const C& c, tbb::split);`

で, c という既存の縮約オブジェクトを元に自分 (this) を初期化する. 典型的には, 計算に必要な共通の変数などを c からコピーする.

これは, 既存の縮約オブジェクトを二つに分ける際に `parallel_reduce` から呼ばれるコンストラクタで, splitting コンストラクタであることを明示する (他のコンストラクタと区別する) ために `tbb::split` を引数に受け取っている.

`void operator()`**:** シグナチャは,

`void operator(const blocked_range3d<int>& r);`

などで, 与えられた範囲 r に対する計算を行う. r の型は `parallel_reduce` に渡したのと同じ物を指定する. 縮約の結果は通常どこかのインスタンス変数に記録する.

`join` **メソッド:** 二つの縮約の結果をさらに縮約 (マージ) する. シグナチャは

`void join(const C& c);`

で, c および `this` の `operator()` の計算が終了しているという前提で, それら二つの縮約結果を `this` に統合する.

例えば以下は, 配列の和を計算するための縮約型である.

```
class array_sum {
  double * a;
public:
  double val;
  /* 通常のコンストラクタ */
  array_sum(double * a) {
    this->a = a;
    val = 0.0;
  }
  /* splitting コンストラクタ */
  array_sum(const array_sum &c, tbb::split) {
    a = c.a;
    val = 0.0;
  }
  /* 与えられた範囲の和を取る operator() */
```

```
  void operator()(blocked_range<int> r) {
    int i;
    for (i = r.begin(); i != r.end(); i++) {
      val += a[i];
    }
  }
  /* 二つの部分和を統合する */
  void join(const array_sum &other) {
    val += other.val;
  }
};
```

そしてこれを以下のように呼び出せば配列の和を取る関数が完成する.

```
double sum(double * a, int n) {
  sum_chunk s(a);
  parallel_reduce(blocked_range<int>(0, n), s);
  return s.val;
}
```

そして, 同じことを C++のクロージャ(lambda 式) の機能を用いて, クラスを明示的に作らずに書くこともできる.

```
#include <tbb/tbb.h>
using namespace tbb;

double sum_without_class(double * a, int n) {
  return
    parallel_reduce(blocked_range<int>(0, n),
                    0.0, //
                    [=](blocked_range<int> r, double ps)->double {
                      int i;
                      for (i = r.begin(); i != r.end(); ++i) {
                        ps += a[i];
                      }
                      return ps;
                    },
                    std::plus<double>());
}
```

parallel_reduce を用いて, 全セルに対する力の計算は以下のように並列化できる.

```
double calc_force_cells_iter(config_t g) {
  int nos = g->n_cells_on_side;
  return
   parallel_reduce(blocked_range3d<int>(0, nos, 0, nos, 0, nos),
```

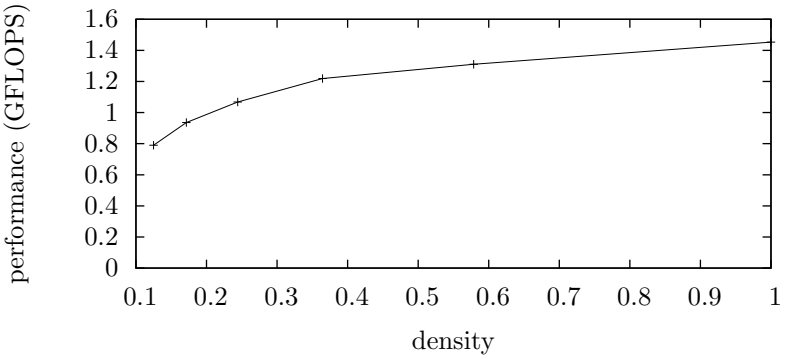


図 1: ベースとなる C コードの演算性能. 横軸は分子の密度で, 1.0 は全格子点に分子がある場合

```
                    0.0,
                    [=](blocked_range3d<int>& r, double U)->double {
                      int i, j, k;
                      for (i = r.pages().begin(); i != r.pages().end(); ++i) {
                        for (j = r.rows().begin(); j != r.rows().end(); ++j) {
                          for (k = r.cols().begin(); k != r.cols().end(); ++k){
                            U += calc_force_cell(g, i, j, k);
                          }
                        }
                      }
                      return U;
                    },
                    std::plus<double>());
}
```

6 実験

6.1 評価環境

評価環境は前号までも用いていた 24 コア (48 ハードウェアスレッド) のマシンである.

- CPU: Intel Nehalem-EX (E7540) 2.0GHz (6 core/12 スレッド × 4 ソケット)
- L3 cache: 18MB
- memory: 24GB DDR3

6.2 初期条件

初期条件は以下のように設定した. まず各分子の位置は面心立方格子上の点を中心とし, x, y, z それぞれに $\pm 0.05\sigma$ の範囲の一様乱数を足している. 面心立方格子は立方体の頂点及び面の中心を点とする格子点である. 格子の大きさは, ある格子点とその最近隣の格子点間のポテンシャルが, ポテン

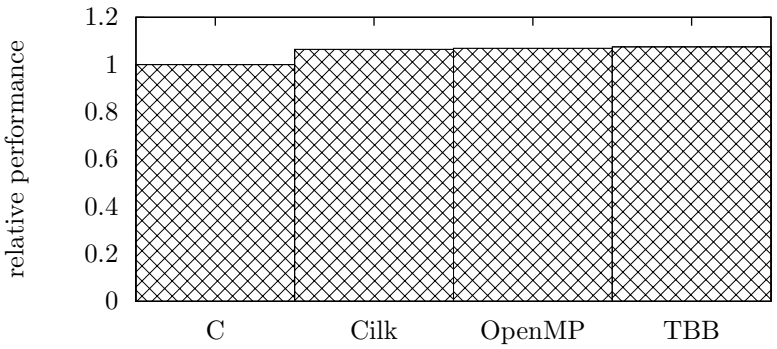


図 2: OpenMP, Cilk, TBB の, C を基準とした相対性能

シャルの底 ($\sqrt[6]{2}\sigma$) となるようにしている. セルの一辺の長さは 3.2σ としている. この設定で, 一つのセルに含まれる面心立方格子上の格子点の数は 35-40 程度である.

密度を可変として, それらの格子点から一様に点を間引いて生成している. 合計の分子数がほぼ 500,000 で一定となるように, シミュレーション領域の大きさ (セルの数) を調整する.

6.3 逐次性能

図 1 に, 元にした C プログラムの性能を示す. 演算数は, 力・ポテンシャルの計算内の浮動小数点演算のみを数え, GFLOPS 値はそれを実行時間全体で割って算出している. 加減乗除すべてを 1 演算と数えており, 距離の計算 (カットオフの判定まで) に対して 12 演算, それ以降の実際の力・ポテンシャルの計算に対して 17 演算とカウントしている.

横軸が密度で, 密度 = 1.0 の場合に, 格子点と同じ数だけの分子が生成されている. 密度が大きくなるほど性能が出ているが, これは当然の結果であり, 二つのセル間の力の計算においてそれぞれに約 n ずつ分子が含まれているとすると, そこでの計算量は約 n^2 に比例し, メモリ参照量は n に比例する. 二つのセルの分子のデータは高々80 分子程度でこれは, 1 次キャッシュに収まる.

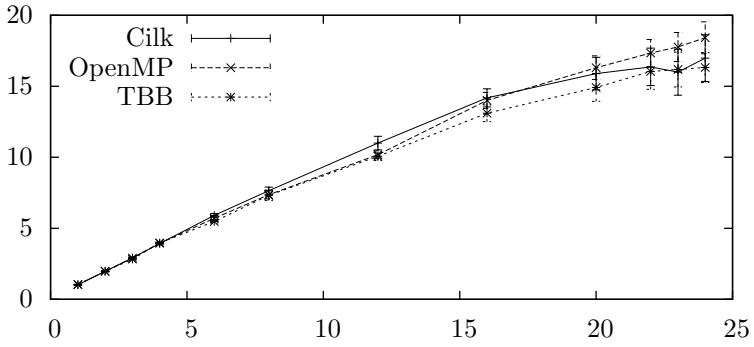
絶対値としては最大でも 1.5GFLOPS 程度でピークに比べれば程遠いが, SIMD 演算なども用いておらず, あくまで気軽に書いた C コードの性能と理解してください.

図 2 は, OpenMP, Cilk, TBB を 1 スレッドで実行した際の, 性能を示したもので, どれもほぼ C と同様 (実際にはわずかに上回っている) の速度を達成している.

6.4 台数効果

最後に図 6.4 に台数効果を示す. 密度を 1 とした場合の台数効果である. 8 スレッドまでは 5 回, 12 スレッド以上は 10 回平均して, 台数効果の平均と, 95%の信頼区間を表示している.

いずれにおいても 24 コアでの台数効果が 17-18 倍程度であった. この理由や, Cilk の台数効果が 22, 23 コアで大きく落ち込んでいることなど, いくつか精査すべき点があるものの今回は時間の関係上, そこまで至らなかった.



7 まとめ

分子動力学法の初歩である Lenard Jones ポテンシャルを用いたシミュレーションを, OpenMP, Cilk, TBB を用いて並列化し, その評価結果について述べた. 三重ループを, OpenMP では parallel for プラグマで, Cilk は直交再帰分割を spawn 構文で, TBB は parallel_reduce, parallel_for を用いて小さい変更で並列化し, いずれも 24 コアで 17-19 倍程度の台数効果を得られることが確認された.

8 今後の予定

毎号の連載記事としては今回をもって一区切りとさせていただきますが, 今後も定期的にこの内容に沿った記事を掲載する予定です. 近々, 探索プログラムおよび分子動力学シミュレーションの, 分散メモリ並列化 (Chapel, X10, UPC など) について掲載する予定です.

参考文献

[1] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation. From Algorithms to Applications.* Academic Press, 1996.

[2] D. C. Rapaport. *The Art of Molecular Dynamics Simulation.* Cambridge University Press.

[3] Intel Threading Building Blocks design patterns. available from http://threadingbuildingblocks.org/documentation.php. Last accessed Nov. 9, 2011.

[4] 吉井範行 岡崎進. **コンピュータ・シミュレーションの基礎** *(***第** *2* **版***)*. 化学同人, 2011.

[5] 佐藤明. *HOW TO* **分子シミュレーション—分子動力学法**, **モンテカルロ法**, **ブラウン動力学法**, **散逸粒子動力学法**. 共立出版, 2004.