連載講座: 高生産並列言語を使いこなす(2)
並列性の記述

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

	ノード内並列	ノード間並列
OpenMP[5]	Y	N
Cilk[2]	Y	N
TBB[6]	Y	N
MPI[4]	Y	Y
UPC[7]	Y	Y
X10[8]	Y	Y
Chapel[1]	Y	Y

表 1: 本連載で扱う言語, ライブラリ

表 2: 連載で取り扱う予定の問題

問題	
自明に並列な処理	並列化構文, 負荷分散, 処理系のスケーラビリティ
木探索	不規則・動的なタスクの記述
分子動力学	大域データ・その分割の記述
疎行列ベクトル積	大域データ・その分割の記述

1 はじめに—本連載について

本連載では, 高生産性を指向した並列処理記述のためのプログラミング言語・ライブラリの, 設計や実装の現状を報告し, 実際の処理系を用いて評価する. 共通の問題をそれらの言語で実際に記述し, 言語間の記述や性能の違いを実際のデータで比較することを目的とする. 東大 HA8000 システムでのインストールや実行の仕方についても記述する.

前回の記事 [9] ではその意義および, 言語の分類軸について述べた. 表 1 に, 取り扱う言語を再掲する. また表 2 に設定している問題とその際の注目点について掲載する.

今回取り上げる問題は表中の「自明に並列な処理」に相当する.

2 今回の問題—Ising モデル

Ising モデルのモンテカルロ法によるシミュレーションを取り上げる.

Ising モデルは強磁性のモデルの一つである. 系は多数 (N) 個の相互作用するスピンから成る. 個々のスピンは 1 または -1 の値をとる. Ising モデルでは, 状態 S にある系のエネルギー $E(S)$ は, i 番目のスピンの値を σ_i としたとき,

$$E(S) = -J \sum_{\langle i,j \rangle} \sigma_i \cdot \sigma_j + H \sum_i \sigma_i \tag{1}$$

で与えられる. J, H は, それぞれスピン間の相互作用エネルギーおよび外部磁場を表す定数で, $J > 0$ である. また, 第一項目の和 $\sum_{\langle i,j \rangle}$ は, 隣接するスピンのペアに渡って取られている. 例えば 2 次元格子点上のスピンであれば, 上下左右に隣接するスピンのペアである.

系の状態はボルツマン分布に従う. つまり, 系がある状態 S にある確率が,

$$q(S) = e^{-E(S)/kT} \tag{2}$$

に比例する. k はボルツマン定数, T は系の温度である. 定性的には, T が大きければ, $q(S)$ は一様分布に近くなり, 小さければ, エネルギーの低い状態に鋭いピークを持つ分布になる. エネルギーの低い状態とは, 定性的には第一項の絶対値が大きい場合—スピンの向きがそろっている場合— および, 外部磁場 H とスピンの向きが反対の場合である.

Ising モデルのシミュレーションではこの確率に従って系の状態をサンプリングし, そこから系の様々な量, 例えば磁化率

$$M = \frac{1}{N}\sum_i \sigma_i \tag{3}$$

の期待値を計算する. そして, H や T を変化させてこの値がどう変化するかを見る.

具体的なサンプリング方法としては, マルコフ連鎖モンテカルロ法の一種である, メトロポリス・ヘイスティング法を用いる. 計算方法は簡単で, 以下によってサンプルを生成する.

$S =$ 適当な初期状態;
repeat {
  $S' = S$ を変化させた状態;
  $r = [0, 1]$ の一様乱数;
  if $(q(S')/q(S) < r)$ $S = S'$;
  if (時々) {
    サンプルとして S を生成;
    S から観測したい状態量を計算;
  }
}

すなわち, ある状態 S から, 異なる状態 S' を生成し, 次の状態を S もしくは S' のいずれからから選択する. 選択にあたっては, $q(S') > q(S)$, すなわち $q(S')$ の確率の方が高ければ, 無条件に S' を選択し, そうでない場合も確率 $q(S')/q(S)$ で S' を選択する. この状態遷移を十分な回数繰り返したあとの S は, $q(S)$ に比例した確率密度からのサンプルと見なせる, というのがメトロポリス・ヘイスティング法である. サンプルを抽出する間隔を十分長くして, それらに相関がないとみなせれば, 上記のループから多数のサンプルを生成することができる.

3 Ising モデルのプログラム

C 言語の形で, 実際の計算をより具体的に示す. $J = 1.0$, $k = 1.0$ としている.

- スピンは $N \times N$ の 2 次元格子状に並んでおり, それを 2 次元配列で表現する. 2 次元配列上で上下左右に隣接するスピンのペアが, エネルギー

$$E(S) = -J\sum_{\langle i,j\rangle} \sigma_i \cdot \sigma_j + H\sum_i \sigma_i$$

の第一項に寄与する. 系全体の状態を表現する構造体として以下を定義する.

```
/* 系の状態 */
typedef struct ising {
  double H;                    /* 外部磁場 */
```

```
    double T;                    /* 温度 */
    double M;                    /* 現在の磁化率 (式 (3)) */
    double E;                    /* 現在のエネルギー (式 (1)) */
    unsigned short RS[3];        /* 乱数の状態 */
    char a[N][N];                /* スピンの状態 */
  } ising;
```

配列の要素 a[i][j] が格子点 (i, j) にあるスピンの状態を表している.

- 状態 S からそれをわずかに変化させた状態 S' を生成するには, S から一様にスピンを一つ選択し, その向きを逆転させたものを S' とする.
- この時, $\Delta E = E(S') - E(S)$ として,

$$q(S')/q(S) = \exp\left(\frac{-\Delta E}{kT}\right)$$

である.

- ΔE は, S と S' で一つのスピン $\sigma_{i,j}$ としか変化していないことを利用して,

$$\Delta E = 2\sigma_{i,j}(J(\sigma_{i+1,j} + \sigma_{i,j+1} + \sigma_{i-1,j} + \sigma_{i,j-1}) - H)$$

で計算できる. ただし $\sigma_{i,j}$ は S における格子点 (i, j) のスピンの値, $\sigma'_{i,j}$ は S' における値である.

以上を総合して, シミュレーションの一ステップを擬似コードの形にしたものが以下である.

```
void do_step(ising * S, double t) {
  // どれを反転させるか?
  int i = NRAND48(S->RS) % N;
  int j = NRAND48(S->RS) % N;
  // 反転した際の M, E の差分
  double dM = - 2 * S->a[i][j];
  double dE = dM * (S->H - (S->a[(i+1) % N][j] + S->a[i][(j+1) % N]
                          + S->a[(i+N-1) % N][j] + S->a[i][(j+N-1) % N]));
  if (dE < 0 || ERAND48(S->RS) < exp(- dE / S->T)) {
    S->a[i][j] = -S->a[i][j]; // 反転
    S->M += dM;
    S->E += dE;
  }
}
```

一つの初期状態から初めて, 上記の do_step を繰り返し呼びながら, サンプルを抽出する部分は, 以下のようになる.

```
sim_result init_and_do_steps(double H, double T, int n_sweeps, int seed) {
  int s;
  ising S[1];    // 系の状態
  sim_result r = { H, T, 0.0, 0.0, 0 };
  init_config(S, H, T, seed); // 乱数でスピン初期化
```

```
  S->M = calc_M(S);
  S->E = calc_E(S);
  // 何度も状態遷移を繰り返す
  for (s = 0; s < n_sweeps; s++) {
    int i;
    for (i = 0; i < N * N; i++) {
      double t = s + (double)i / (N * N);
      do_step(S, t);
    }
    r.n_samples++;
    r.M += S->M;
    r.E += S->E;
  }
  return r;
}
```

seed は乱数の種である. 系が N^2 個のスピンで構成されているため, サンプルは N^2 ステップに一度抽出している. n_sweeps はシミュレーションの長さを調節するパラメータで, 具体的には, (n_sweeps $\cdot N^2$) ステップ実行する.

結果を格納する構造体 sim_result には, パラメータ (H と T), 取り出されたサンプル数, 全サンプルにおける磁化率, エネルギーの合計を格納する.

```
typedef struct sim_result {
  double H;                    /* 外部磁場 */
  double T;                    /* 温度 */
  double M;                    /* 磁化率（全サンプルの合計）*/
  double E;                    /* エネルギー（全サンプルの合計）*/
  int n_samples;               /* サンプル数 */
} sim_result;
```

ここまでの部分はどの枠組みを使った場合も逐次的に実行される.

4 並列性

このシミュレーションには多くの並列性がある.

1. 信頼できる結果を得るためには, 確率密度 $q(S)$ に従ったサンプルを多数生成する必要がある.
2. パラメータ (H や T) を変化させてシミュレーションをする必要がある.

両者とも必要な通信は, 結果の総計や収集のみで, 計算中の通信は必要ない. モンテカルロ法が容易に並列化できると言われる所以である.

ここではまず簡単のため, 前者の並列性のみを抽出するという前提で考える. 逐次の C 言語で書けば以下のようになるループを並列化するというタスクである.

SIMPLE:

```
for (i = 0; i < n_times; i++) {
  seed = ...;
  results[i] = init_and_do_steps(H, T, n_sweeps, seed);
}
```

より現実的な設定では, H や T などの値も変えながら実行する多重ループになる. 例えば, H や T を適当な範囲で刻みながらシミュレーションを行う.

NESTED:

```
for (h = 0; h < n_Hs; h++) {
  for (t = 0; t < n_Ts; t++) {
    for (i = 0; i < n_times; i++) {
      H = ...;
      T = ...;
      seed = ...;
      results[h][t][i] = init_and_do_steps(H, T, n_sweeps, seed);
    }
  }
}
```

この記事の以降では, 各言語でこれらのループを並列化し, 結果を集めるなどの処理がどう記述されるかを見る. 前者を SIMPLE, 後者を NESTED, と呼ぶことにする.

5 OpenMP による並列化

5.1 SIMPLE ループの OpenMP による並列化

OpenMP では, OpenMP の最も基本的なプリミティブである, parallel プラグマと, for プラグマを用いて上記のループを並列化できる.

parallel プラグマの文法は,

```
#pragma omp parallel
文
```

で, これによって, 同一の「文」を実行するワーカスレッドを複数生成する. 「文」はもちろん複合文や関数呼び出しであっても良い.

parallel プラグマ自体は「文」を分割して実行する指示ではなく, あくまで同一の文を複数のワーカスレッドが実行する指示, つまり SPMD 実行を始める指示である.

実際の, いわゆる並列実行 (仕事の分割) は, for 構文を代表とする, worksharing 構文で指示する. SIMPLE ループの分割は以下のとおりである.

```
#pragma omp for
for (i = 0; i < n_times; i++) {
  results[i] = init_and_do_steps(H, T, n_sweeps, i);
}
```

とすれば, parallel 構文で作られたワーカスレッド間でこのループの本体が分割されて実行される. プログラムの実行に用いるワーカスレッドの数は, 環境変数 OMP_NUM_THREADS で指定する.

分割の仕方は, static (各ワーカに均等に, 連続した一つの範囲を割り当てる), dynamic (固定幅の範囲を空いているワーカに割り当てる), guided (割り当てる範囲を動的に変化させる) など, いくつかあるが, 無指定時は static となる. これは, 繰り返し間の負荷が均等なループを分割する最も自然な手段である.

並列実行が自明に行えるよう, 許容される for 文にはいくつもの制限がある. 例えば continue や break が出現しない, ループの上限と一回ごとの増分がループを通して一定, などである. 一言で言えば繰り返しの回数及び, 任意回目の繰り返しの際のループ変数 (上記の i) の値が, 簡単に (「開始値 + 定数 × 繰り返し番号」のように) 計算できることである.

OpenMP は, 一部の研究用プロトタイプを除き, ハードウェア共有メモリ上での実行が前提となっており, 結果を格納する配列 results は, ハードウェア共有メモリによってワーカ間で共有される.

5.2 NESTED ループの OpenMP による並列化

OpenMP では for ループが入れ子になっている場合, 完全入れ子ループ (perfectly nested loop) であれば, その入れ子になったループを分割実行できる. 完全入れ子ループとは, 外側のループのすぐ内側に (他の文を挟まずに) 別のループが含まれる, という形の入れ子である. つまり,

```
for (...) {
  S1;
  for (...) {
    S2;
  }
}
```

は完全入れ子ループではないが,

```
for (...) {
  for (...) {
    S3;
    S4;
  }
}
```

はそうである. どのレベルまでを分割の対象とみなすかは, for プラグマに collapse(レベル) という指示を与えることで指定する. 2 重ループを分割したければ, collapse(2) と書く.

我々の NESTED ループは, 幸いにして完全入れ子ループである.

```
#pragma omp for collapse(3)
for (h = 0; h < n_Hs; h++) {
  for (t = 0; t < n_Ts; t++) {
    for (i = 0; i < n_times; i++) {
      H = ...;
      T = ...;
      seed = ...;
```

```
        results[h][t][i] = init_and_do_steps(H, T, n_sweeps, seed);
      }
    }
  }
```

とすることで, 3 重ループ全体を分割して実行できる.

逐次であればより自然な,

```
  for (i = 0; i < n_times; i++) {
    H = ...;
    for (j = 0; j < H_times; j++) {
      T = ...;
      for (k = 0; k < T_times; k++) {
        seed = ...
        results[i][j][k] = init_and_do_steps(H, T, n_sweeps, seed);
      }
    }
  }
```

のような書き方は, 完全入れ子にはならない.

5.3 OpenMP の制限

多くのプログラムが, 計算の中心を占める for ループの入れ子を, OpenMP の parallel プラグマと for プラグマで記述することで, 並列化可能だが, 制限もある. for 文が完全入れ子になっていない場合, ループ中に関数が呼び出されその内部でさらに for 文が出現する場合, 木探索や分割統治のように計算自体が再帰呼び出しで書かれている場合などは, これだけでは並列化できない. 一般に, 入れ子になった並列性を自由に抽出できないところに制限があるとまとめられる.

これは OpenMP の基本的な実行モデルに起因するものである. 具体的には, 完全入れ子ループの並列化は, 個々のタスクをループの添字だけで特定できる. その他の環境 (繰り返しの外で定義されている変数など) は全タスク間ですべて共通になる. 各ワーカは自分が担当する添字の範囲だけを静的または動的に決定して, その範囲の繰り返しを実行するだけで良い.

OpenMP で自由な入れ子並列性サポートに近いものとして, OpenMP 3.0 以降の仕様で導入された task プラグマがある. 次節で述べる Cilk と, 概念としては似ている. しかしその実装の成熟度は大きく処理系に依存し, 特に GCC の現時点での実装はあまり優れていない. ループを単純に分割して実行するという元々の基本的な仕組みとの相互作用も, プログラマにとっては理解しづらい. task プラグマについては次回以降の木探索の並列化の際に紹介・評価する.

現時点で, 自由に入れ子になった並列性を実行するのに優れているのは次節で述べる Cilk であろう.

6 Cilk による並列化

6.1 Cilk の概要

Cilk には MIT のオープンソース版と, その後 Intel の, Parallel Building Blocks の一部としてサポートされている, Cilk plus がある. ここでは前者を用いている.

Cilk では, spawn 構文と sync 構文によって並列化を記述する. 最も簡単な例として, フィボナッチ数列の再帰呼出しは以下のように書ける.

```
cilk int fib(int n) {
  if (n < 2) return 1;
  else {
    int x = spawn fib(n - 1);
    int y = spawn fib(n - 2);
    sync;
    return x + y;
  }
}
```

一般には,

`spawn` 関数呼び出し

ないし,

`spawn` 変数 `=` 関数呼び出し

という構文によって, その関数呼び出しが非同期に実行される. つまり, その関数の実行と, それに引き続く文が並列に実行されうる.

```
sync;
```

は, 非同期に実行されている関数呼出しの終了を待つ. 具体的には, sync を実行した関数が非同期に呼び出しているすべての関数呼び出しの実行が終了するのを待つ. 上記の例であれば, `fib(n - 1)` および `fib(n - 2)` の実行終了を待つことになる.

Spawn 構文は, 機能的には Pthreads などのスレッドライブラリが提供する, スレッド生成機能 (`pthread_create`) を使いやすくしたものという風にも見えるが, その実装は Pthreads のそれよりもはるかに軽量である. オーバーヘッドも少ない上に, 上記の fib 関数のように再帰的に大量に spawn を実行しても問題はなく, 実際良好な台数効果が得られる. 実際のスケジューリングは, spawn によって作られる木構造 (spawn を実行した関数呼び出しと, それによって呼ばれた関数呼び出しを親子関係とする木構造) を, 動的に分割する, work stealing という動的負荷分散手法によってなされる. この方式のルーツは, Multilisp という並列 Lisp 処理系において実装された, Lazy Task Creation [3] という方式である.

Lazy Task Creation では, 一定数 (典型的にはハードウェアのコア数) のワーカスレッドがプログラム開始時に立ち上がり, 最初は main 関数を実行するワーカスレッドだけが実行中 (busy) である. Cilk ではワーカスレッド数は起動時にオプション (`-nproc`) で指定する.

Busy なワーカは, あたかも spawn が普通の関数呼び出しであるかのような順番で, 実行を行う. つまり, あるワーカ A が

```
x = spawn fib(n - 1);
...;
```

という spawn 文に到達した際, A は `fib(n - 1)` の実行を直ちに開始する. A が `fib(n - 1)` を実行している間に, 仕事のないワーカが現れなかった場合, A は `fib(n - 1)` 終了後, そのまま続き (`...` の部分— `fib(n - 1)` の「継続」) を実行する. 言い換えれば, 普通の逐次実行と同じ順番で実行が行

われる．A が `fib(n - 1)` を実行中に仕事のないワーカが現れた場合, `fib(n - 1)` の継続を, そのワーカに与える (work stealing).

このようにタスク生成 (spawn) 時に, あたかもそれが逐次実行であるかのように, 生成されたタスクを直ちに実行するスケジューリング方式を, work-first スケジューリングと呼ぶ. それと反対の方法は, タスク生成時には, 生成されたタスクを後から他のワーカへ分け与える対象としてどこかのデータ構造へ保存「パッケージング」しておいて, ワーカはその継続を実行するというものである (parent-first スケジューリング).

spawn された関数の中でまた spawn が行われ得るので, 一般に busy な各ワーカは, 複数の steal 可能な継続を保持しているが, その場合, spawn の木構造の最も根に近い部分の—そのワーカが保持している中で, 最も過去に作られた—継続を盗む.

この, work-first スケジューリングの原理と, work stealing の際に, 一番外側の継続を盗むという二つの方式は, 非常に多くの場合にうまく動作する.

- 動的負荷分散であるが故に, spawn の木構造がアンバランスな木であっても, 頑健に負荷分散される.
- その際, 無闇に木構造が細分化されない. これは, work stealing 時に, 木構造の根に近い部分から継続を盗むことによる.
- Work-first スケジューリングにより, work stealing がおきていない間の実行順序は逐次のそれと全く同じになる. プログラマは,「局所性の高い逐次プログラム (キャッシュ上のデータの再利用性の高い逐次プログラム)」を元にして, spawn を挿入すれば, その局所性がよく保存される.

動的負荷分散を実装しようと思ったときに, 考え方として一見自然で, 実装も単純なのは, parent-first である. それは, プログラムの文面から, タスクとしてパッケージングする必要のある状態 (spawn 構文の場合であれば, 関数呼び出しの引数と関数名) を抽出しやすく, それに対して継続の方は関数実行の「途中から」実行を再開するような情報をどこかへ保存しなくてはならない. 一方で, 上記にあげたような利点の多くは失われる.

Work-first スケジューリングを実装するには, spawn に引き続く計算 (継続) をパッケージングして, 他のタスクがそれを盗んで実行できなくてはならない. それには関数中で今後使われる変数 (生きている変数) の解析が必要で, 処理系が実装するならばプログラムを解析できる言語処理系が必要で, コンパイラから独立したライブラリとして実装するのは難しい.

6.2 SIMPLE ループの Cilk による並列化

MIT からリリースされているオープンソース版の Cilk には並列 for のようなループ並列化のための特別な構文は提供されていないが, OpenMP の worksharing for 構文で許容されているループであれば, 2 分割を再帰的に繰り返すことで, 形式的に書き換えることができる. つまり, 我々の SIMPLE ループは, 元々の区間 $[0, n\_times)$ の一部の範囲 $[a, b)$ を, 実行する関数を以下のような分割統治法を用いて書くことで並列化できる.

すなわち,

```
/* [i0,i1) の範囲を計算する関数 */
cilk run_sims_rec(int i0, int i1, ...) {
  if (i0 + 1 == i1) {
    /* ループの本体に相当 */
```

```
    seed = ...;
    results[i0] = init_and_do_steps(H, T, n_sweeps, seed);
  } else {
    /* 与えられた範囲を 2 分割 */
    int ic = (i0 + i1) / 2;
    spawn run_sims_rec(i0, ic, ...);
    spawn run_sims_rec(ic, i1, ...);
    sync;
  }
}
```

を作った上で, 全体を,

```
run_sims_rec(0, n_times, ...);
```

で実行する. ただし, ... 部分には, ループの本体で最終的に必要になる変数を渡す. 例えば上記であれば, results, H, T, n_sweeps などの変数が (大域変数でない限り) 渡される.

再帰呼び出しを行うかどうかを,

```
if (i0 + 1 == i1)
```

で判定しておりこれは, 2 つ以上の要素が含まれる区間はことごとく分割するという方針での分割である. もちろん繰り返し一回の仕事がもっと小さければ区間がもっと大きなうちに, 分割をやめて, 逐次のループに移行することもでき, 実際各繰り返しの処理が少ない場合はある程度必要になる.

例えば大きさ 10 未満の区間は分割しないということであれば,

```
if (i1 - i0 < 10)
```

とすればよい.

どこまで分割するのが最適であるかは難しいが, ここで注意としては決して, spawn される関数呼び出しの数を, コア数程度にする, などの難しい調整は求められていないという点である. Spawn そのものにかかるオーバーヘッドは小さく, 関数呼び出しのコストの数倍程度である. そのオーバーヘッドと, 「実質的な仕事」である再帰呼び出しのリーフにおける処理量の比が, ある一定以上小さくなればよい (例: 1% 以下).

今我々が対象としている計算では, 繰り返し一回自身が, 系全体のシミュレーション一回に相当するもので, spawn のオーバーヘッドに比べて遥かに大きな計算なので, 範囲が 1 になるまで細分化しても何も問題はない.

6.3 分割統治の重要性

なお, 以下のようにループの各繰り返しを spawn する記述:

```
for (i = 0; i < n_times; i++) {
  seed = ...;
  results[i] = spawn init_and_do_steps(H, T, n_sweeps, seed);
}
sync;
```

は, プログラムの書き換えとしてはより単純で, 実際このように記述することも可能だが, 並列化の方法としてはあまり好ましくない. この記述では, すべての spawn が一つのタスクによってなされてしまう. つまりすべての spawn は直列におこり, 複数の spawn が並列に起こることはありえない. 従って, 複数の work stealing が並列に起こることもない.

実装レベルの言葉で言えば, work stealing が可能な継続を持っているワーカは常に一つである. したがってすべてのワーカが実行可能な仕事を手にするまでに, O(ワーカ数) の時間がかかる. 一方先に述べた, 2 分割を再帰的に繰り返す方法であれば,

```
spawn run_sims_rec(i0, ic, ...);
spawn run_sims_rec(ic, i1, ...);
```

の始めの `run_sims_rec(i0, ic, ...)` を実行しているワーカと, その継続を実行しているワーカは, ともにさらなる work stealing の対象となる. 従ってタスクはより迅速に伝搬される.

前者が望ましいことの, もっと実装に中立な理解の仕方がある. それは, 後者の計算のクリティカルパス (直列にしか実行できない一連の計算の最大長) があきらかに, `n_times` に比例しているのに対し, 後者は log(`n_times`) に比例しているというものである. 言い換えれば前者は無限のプロセッサを持ってきても, 得られる台数効果は高々定数倍:

$$\frac{\text{計算 } \texttt{spawn init\_and\_do\_steps}(\texttt{H}, \texttt{T}, \texttt{n}_\texttt{s}\texttt{weeps}, \texttt{seed}); \text{一回の実行時間}}{\text{代入 } \texttt{results}[\texttt{i}] = \text{一回の実行時間}} + 1$$

である. 後者の場合, 最大で理論的には $O(\texttt{n\_times}/\log(\texttt{n\_times}))$ までの台数効果が得られる.

形式的な変換とはいえ, ループの並列化を行うたびにそれを補助関数を作って再帰呼び出しに書き換えるというのは面倒である. 形式的な変換であればこそ, 処理系にやって欲しいものである. Intel からリリースされている Cilk Plus という処理系では, 並列 for 文がサポートされている.

Cilk の実行方式は, 常に動的に負荷分散をするというもので, OpenMP のように, ループ開始時点で静的に分担を決めるということはしない. 従って, 繰り返し間の負荷が均等でないループ, 外側の繰り返し回数が十分でなく, 内側での並列性も抽出することが必須なループ, 再帰呼び出しなどで威力を発揮する.

6.4 NESTED ループの Cilk による並列化

我々の NESTED ループを並列化する場合はどうなるであろうか? 概念的には, 辺の長さが `n_Hs` × `n_Ts` × `n_times` の直方体を 2 分割して行くのと同じである. どの辺を先に 2 等分するかによって方針が分かれるが, この例題のように, 繰り返し間でのデータの共有が全くないような場合であれば, どのように分割しても大差ない.

例えば外側のループをこれ以上細分できなくなるところまで細分し, そこから内側のループを分割する, という方針をコードにしたものが以下である.

```
cilk void run_sims_rec(int h0, int h1, int t0, int t1, int i0, int i1,
                       double H0, double dH, double T0, double dT,
                       int n_sweeps) {
  if (h1 - h0 > 1) {
    spawn run_sims_rec(h0, (h0+h1)/2, t0, t1, i0, i1, H0, dH, T0, dT, n_sweeps);
    spawn run_sims_rec((h0+h1)/2, h1, t0, t1, i0, i1, H0, dH, T0, dT, n_sweeps);
    sync;
```

```
  } else if (t1 - t0 > 1) {
    spawn run_sims_rec(h0, h1, t0, (t0+t1)/2, i0, i1, H0, dH, T0, dT, n_sweeps);
    spawn run_sims_rec(h0, h1, (t0+t1)/2, t1, i0, i1, H0, dH, T0, dT, n_sweeps);
    sync;
  } else if (i1 - i0 > 1) {
    spawn run_sims_rec(h0, h1, t0, t1, i0, (i0+i1)/2, H0, dH, T0, dT, n_sweeps);
    spawn run_sims_rec(h0, h1, t0, t1, (i0+i1)/2, i1, H0, dH, T0, dT, n_sweeps);
    sync;
  } else {
    /* リーフの場合 */
    double H = H0 + h0 * dH;
    double T = T0 + t0 * dT;
    int seed = h0 * n_Ts * n_times + t0 * n_times + i0;
    results[h0][t0][i0] = init_and_do_steps(H, T, n_sweeps, seed);
  }
}
```

変数の多さにやや頭痛がするものの, 妙な技工はない, わかり易いコードといえなくもない. 元々の三重ループの美しさ・単純さとは比べるべくもないが, ここでのポイントはこれを Lazy Task Creation で動的負荷分散することにより,

- プロセッサ数とパラメータ設定に応じた適切なレベルまでの並列度の抽出
- プロセッサ数が 2 の冪でない場合の細分
- ある程度の OS ジッタに対する耐性
- 少ないオーバーヘッド

などが自然に達成されるということである. 今後どんどんコア数の増えていくマルチコアプロセッサに, よく対応したコードであると言うことができる.

なお, 変数の多さ, 特に 3 次元の矩形 (直方体) を表す 6 つの変数を何とかしたい場合は, 3 次元の矩形を表す構造体を作るのが自然であろう. 矩形を二つに分割するという部分を関数としてまとめれば, 上記の長い条件分岐も 1 つにまとめられる. まさしく次節以降で述べる, TBB, Chapel, X10 などは, それを陽にサポートしている.

ところでこれを多数のワーカで実行すると, 個々のワーカが work stealing の結果手に入れるタスクは多くの場合, 扁平な—(`h1 - h0`) が小さいか, すでに 1 であるような—直方体になるだろう. 今回の例題では必要なのは計算負荷の均等化のみであり, それで全く問題ない. これが望ましくなく, 直方体をなるべく均整のとれた形に—立方体に近く— 保ちたい場合も多い. 各領域が, 隣の領域のデータを少しだけ参照し, 隣同士の計算を同じプロセッサ上で実行すると計算/通信比が高まるという場合である.

そのような場合は, 3 辺のうち最も大きい辺を分割するという方法がしばしば有効であり, 領域分割などでよく用いられる (Orthogonal Recursive Bisection; 直交再帰二分割). それは以下のように書ける.

```
cilk void run_sims_rec(int h0, int h1, int t0, int t1, int i0, int i1,
                       double H0, double dH, double T0, double dT,
```

```
                        int n_sweeps) {
    if (h1 - h0 > 1 && h1 - h0 >= t1 - t0 && h1 - h0 >= i1 - i0)) {
      spawn run_sims_rec(h0, (h0+h1)/2, t0, t1, i0, i1, H0, dH, T0, dT, n_sweeps);
      spawn run_sims_rec((h0+h1)/2, h1, t0, t1, i0, i1, H0, dH, T0, dT, n_sweeps);
      sync;
    } else if (t1 - t0 > 1 && t1 - t0 >= i1 - i0) {
      spawn run_sims_rec(h0, h1, t0, (t0+t1)/2, i0, i1, H0, dH, T0, dT, n_sweeps);
      spawn run_sims_rec(h0, h1, (t0+t1)/2, t1, i0, i1, H0, dH, T0, dT, n_sweeps);
      sync;
    } else if (i1 - i0 > 1) {
      spawn run_sims_rec(h0, h1, t0, t1, i0, (i0+i1)/2, H0, dH, T0, dT, n_sweeps);
      spawn run_sims_rec(h0, h1, t0, t1, (i0+i1)/2, i1, H0, dH, T0, dT, n_sweeps);
      sync;
    } else {
      /* リーフの場合 */
      double H = H0 + h0 * dH;
      double T = T0 + t0 * dT;
      int seed = h0 * n_Ts * n_times + t0 * n_times + i0;
      results[h0][t0][i0] = init_and_do_steps(H, T, n_sweeps, seed);
    }
  }
```

本質的には, 直方体を, その左下隅と右上隅の座標 (合計 6 個の変数) で表現していることに相当する. 従って, このコードの見た目を改善したければ, それらの変数を矩形領域を表すデータを一つの構造体にまとめることが自然であり, 実際次節以降で述べる TBB や, Chapel, X10 やそれらの概念を陽にサポートしている.

7 インテルスレッディングビルディングブロック (TBB) による並列化

7.1 TBB の概要

インテルスレッディングビルディングブロック (TBB) は, 並列化構文を C++のテンプレートを駆使してライブラリにしたものである. OpenMP のプラグマや他の言語拡張のような方式に比べると, 記述の自然さや, コンパイル時エラーメッセージの不可解さなどの難があるが, コンパイラに依存せずに動作するという利点がある.

機能は

- 並列ループ
- タスク並列
- 並列データ構造 (コンテナ)
- 並列メモリ管理

など, 逐次コンパイラそのものには備わっていない並列化に必要な機能がそろっている.

タスク並列については次号以降の木探索で紹介することとし, ここでは, 今回の例題に必要な並列ループを説明する. OpenMP の for プラグマに相当するのは, parallel_for という関数である. 構文を拡張していないので, 記法としては関数呼び出しの記法になる. その関数に, 繰り返しの全範囲を指定するオブジェクトと, 範囲を与えられて, その範囲の繰り返しを実行するオブジェクトを渡すのが基本である.

```
parallel_for(全範囲を指定するオブジェクト,
                一部の範囲を受け取りその範囲を実行するオブジェクト)
```

前者の, 「全範囲を指定するオブジェクト」としては, TBB が提供する組み込みクラスとして, $i = a, \ldots, b-1$ のような一次元の区間を表す blocked_range と, 2 次元の区間 (矩形) を表す blocked_range2d などがある. それ以外のものを作りたければ, 決められたインタフェース (実質的には範囲を 2 分割するメソッドを持つクラス) を書けばそれを使うこともできる.

後者の, 与えられた範囲を実行するオブジェクトは, 任意のクラスで良く, その演算子 operator() を定義する. その演算子は, 添字の範囲を伴って TBB フレームワークから呼ばれたら, 渡された範囲の計算を実行するように定義する.

7.2 SIMPLE ループの TBB による並列化

従って我々の SIMPLE ループを並列実行する TBB のコードは以下のようになる.

```
parallel_for(blocked_range<int>(0, n_times),
             apply_init_and_do_steps(H, T, n_sweeps, results));
```

ここで, apply_init_and_do_steps は以下のように定義されたクラスである.

```
class apply_init_and_do_steps {
  double H;
  double T;
  int n_sweeps;
  sim_result_t results;
 public:
  apply_init_and_do_steps(double H, double T,
                   int n_sweeps, sim_result_t results) {
    this->H = H;
    this->T = T;
    this->n_sweeps = n_sweeps;
    this->results = results;
  }
  void operator()( const blocked_range<int>& r ) const {
    for( int i=r.begin(); i!=r.end(); ++i ) {
      seed = ...;
      this->results[i] = init_and_do_steps(H, T, n_sweeps, seed);
    }
  }
};
```

実質的なのは唯一, operator() の定義であり, 範囲を受け取ってその範囲に対して繰り返し本体を実行する.

TBB も Cilk と同様, 動的負荷分散を行うスケジューラを基本として設計されている. parallel_for で実行されているコードの中に再び parallel_for が表れてもよい.

一方で単純なループの並列化をするのにいちいちクラスを定義しなくてはならず, それもループ本体と parallel_for の呼び出しがコード上に分散し, 可読性の低いコードになる. コードが分散する原因は, parallel_for が関数呼び出しであるため, ループの本体を渡す方法はオブジェクトを作るしかなく, そのためにはクラスを書かねばならず, そのためにはそのクラスをどこか別の場所 (ソースコードのトップレベル) に書かなくてはならない, ということである. また, ループの外側で定義され, 繰り返し本体で用いる変数を, そのオブジェクトのコンストラクタの引数を通じて渡したり, あまり本質的でない作業に多くの行数を費やす. OpenMP などの言語処理系と一体化した方式では, それらをコンパイラが自動化していた.

C++0x という C++の次期仕様では, 関数型言語ではお馴染みの, クロージャーが定義されており, それをサポートする処理系では以下のように書けるそうである.

```
#include "tbb/tbb.h"
using namespace tbb;
parallel_for(blocked_range<int>(0, n_times),
             [=] (const blocked_range<int>& r ) const {
               for( int i=r.begin(); i!=r.end(); ++i ) {
                 seed = ...;
                 this->results[i] = init_and_do_steps(H, T, n_sweeps, seed);
               }
             });
```

[=] 以降 4 行目から 8 行目までが, クロージャという無名のオブジェクトを表しており, 実質的には先の例で明示的にクラスを作りそのメソッドを定義していた部分を, インラインに書けるようになっている.

ループの構造と本体が同じ場所に書けているので, 理屈の上では可読性が向上しているというのはその通りだが, これを見てまた C++の悪ノリが始まったと思うのは筆者だけだろうか.

7.3 NESTED ループの TBB による並列化—3 次元領域の表現

TBB において, 我々の NESTED ループを並列化する場合はどうなるであろうか?

すでに述べた parallel_for を 3 重にネストする必要があるとすると, 個々のループに対して一つクラスを定義して, 最終的なループ本体はそのうち一つの operator() 定義の中に埋もれることになる. かなり絶望的な思いに駆られるところだが, 前節で述べたように, 幸いにして TBB では, parallel_for の呼び出しにおいて, 「添字の全範囲を指定するオブジェクト」のところに, 二次元や三次元の矩形を指定することができる. それぞれ blocked_range2d, blocked_range3d というクラスを使う. 従って三重ループへの変更は, まず以下のように blocked_range3d を用いて並列 for を記述し,

```
parallel_for(blocked_range3d<int>(0, n_Hs, 0, n_Ts, 0, n_times),
             apply_init_and_do_steps(H0, dH, T0, dT, n_sweeps));
```

繰り返し本体の方も, 3 次元の部分領域に対して操作をするように書き直せばよい. 以下では operator() 本体だけを示す.

```
void operator()( const blocked_range3d<int>& r ) const {
  for( int h=r.pages().begin(); h!=r.pages().end(); ++h ) {
    for( int t=r.rows().begin(); t!=r.rows().end(); ++t ) {
      for( int i=r.cols().begin(); i!=r.cols().end(); ++i ) {
        double H = H0 + h * dH;
        double T = T0 + t * dT;
        int seed = h * n_Ts * n_times + t * n_times + i;
        results[h][t][i] = init_and_do_steps(H, T, n_sweeps, seed);
      }
    }
  }
}
```

3 次元でも足りない場合や, また, 自分で好きな「繰り返し領域」を定義することもできる. そのクラスが満たすべきインタフェースのうち本質的なものは, 領域を 2 分割する操作である. blocked_range2d (または blocked_range3d) の実装ではそれに対して, 長方形 (または直方体) の一番長い辺を 2 等分する, という分割を行っている. 実際に定義すると有用な例として, 3 角形領域, Adaptive Mesh などの場所によって解像度の異なる領域, 非構造格子で区切られた領域などがあげられよう.

8 MPI による並列化

8.1 SIMPLE ループの MPI による並列化

さてここからが分散メモリ計算機をサポートする処理系である. まずその代表格である MPI である. 言うまでもなく最も広く使われているライブラリで, 言語処理系ではなく C, C++, Fortran に対応したライブラリである.

SPMD モデルに基づいており, 通信手段はメッセージ送受信およびその集合通信のみを基本とする (MPI 2.0 以降の仕様では遠隔 PUT/GET も提供されている).

今回の例題を並列化する際の自然な作戦は, OpenMP の for 文の static スケジューリングと同じものである. OpenMP との違いは, 各プロセッサの担当範囲を自分で明示的に計算する必要があるという点と, 得られた結果を, 結果を集計するプロセス (ここではランク 0 のプロセス) に明示的に送信する必要がある, という点である. 今回はこれに MPI_Gather を用いる.

SIMPLE ループの分割実行は以下のようになる.

```
int n_per_proc = (n_times + THREADS - 1) / THREADS;
int begin = n_per_proc * MYTHREAD;
int end   = begin + n_per_proc;
for (i = begin; i < end; i++) {
  int seed = ...;
  my_results[i] = init_and_do_steps(H, T, n_sweeps, seed);
}
MPI_Gather(my_results + begin,
           sizeof(sim_result) * n_per_proc, MPI_BYTE,
           results, sizeof(sim_result) * n_per_proc, MPI_BYTE,
           0, MPI_COMM_WORLD);
```

THREADS, MYTHREAD はそれぞれ, 自プロセスのランクと, 全プロセス数を表している (次節で述べる UPC との統一のためにこの用語を用いている).

なお, MPI_Gather の制限で, 転送元領域と転送先領域が重なってはいけないので, 最終的に結果を集計する配列 (results) とは別に, 中間結果を書き出す配列 (my_results) を用意している. さらに, 明らかにメモリの無駄遣いではあるが, 添字をずらすなどの汚いコードを避けるために, my_results 自身も全結果を格納できる大きさとしている.

8.2 NESTED ループの MPI による並列化

NESTED ループの場合はどうか?

個々の繰り返しにおける計算は完全に独立しており, 負荷も一定と仮定しているので, 目標は各プロセス毎の繰り返し数の均等化—P 個のプロセスがそれぞれ約 $(\mathtt{n\_Hs} \times \mathtt{n\_Ts} \times \mathtt{n\_times})/P$ の繰り返しを実行する—のみである. それは一般には矩形領域にならない. あまり紙の上で計算をしなくてよい概念的にわかり易い記述法は, 全プロセスが上記の三重ループをまわって, 自分の担当しない領域はスキップするというものである.

```
int n_per_proc = (n_Hs * n_Ts * n_times + THREADS - 1) / THREADS;
for (h = 0; h < n_Hs; h++) {
  for (t = 0; t < n_Ts; t++) {
    for (i = 0; i < n_times; i++) {
      int thread = (h * n_Ts * n_times + t * n_times + i) / n_per_proc;
      if (thread != MYTHREAD) continue;
      double H = H0 + h * dH;
      double T = T0 + t * dT;
      int seed = ...;
      my_results[h][t][i] = init_and_do_steps(H, T, n_sweeps, seed);
    }
  }
}
MPI_Gather(((sim_result *)my_results) + MYTHREAD * n_per_proc,
             sizeof(sim_result) * n_per_proc, MPI_BYTE,
             results, sizeof(sim_result) * n_per_proc, MPI_BYTE,
             0, MPI_COMM_WORLD);
```

しかしさすがにこれでは無駄な空回りが大きい. 台数効果としても,

$$\frac{\text{ループ本体を実行する時間}}{\text{ループを空回りする時間}} + 1$$

が上限となる.

これを解消したければ, 外側のループを適切な上限・下限で区切る必要がある. しかしその計算は結構な頭痛を伴う. それをするくらいなら, 最初から三重ループ全体を手動で平坦化してしまう方が考え易い. 三次元の繰り返し空間全体に一続きの通し番号 (下記の fi) をつけ, fi から各次元の添字を計算する.

```
int fi;
```

```
int n_per_proc = (n_Hs * n_Ts * n_times + THREADS - 1) / THREADS;
int begin = n_per_proc * MYTHREAD;
int end   = begin + n_per_proc;
for (fi = begin; fi < end; fi++) {
  int h = fi / (n_Ts * n_times);
  int t = (fi % (n_Ts * n_times)) / n_times;
  int i = fi % n_times;
  double H = H0 + h * dH;
  double T = T0 + t * dT;
  int seed = h * n_Ts * n_times + t * n_times + i;
  my_results[h][t][i] = init_and_do_steps(H, T, n_sweeps, seed);
}
MPI_Gather(((sim_result *)my_results) + MYTHREAD * n_per_proc,
             sizeof(sim_result) * n_per_proc, MPI_BYTE,
             results, sizeof(sim_result) * n_per_proc, MPI_BYTE,
             0, MPI_COMM_WORLD);
```

なお, ここで結果の集計が MPI_Gather だけですんでいる—したがってコードがそれほど醜くならない—のは, 以下の条件による.

- results, my_results がともにグローバルに宣言された 3 次元配列 (したがってそのアドレスは連続している) である. C では, 配列の大きさを実行時に決めようと思うと, 動的に確保しなくてはならず, その場合 my_results[h][t][i] のような簡便なアクセス記法を諦めるか, もしくは配列全体を連続領域に確保することをあきらめなくてはならない (ポインタの配列を作る). 後者を選択すれば, もちろん通信は一度の MPI_Gather では済まなくなる.
- 各プロセスが担当するのも, 3 次元配列上でアドレスが連続している領域である. さもなければ, 各プロセスが送る結果も飛び飛びのアドレスに対するものになり, 当然 MPI_Gather では済まない.

いずれの場合も送信側でデータをまとめ, 受信した側でデータを戻すという操作が必要になる.

9 UPC による並列化

9.1 UPC の概要

UPC (Unified Parallel C) は, SPMD に基づく実行モデルに, OpenMP の for プラグマに似たセマンティクスを持つ worksharing for 構文 (upc_forall), 全プロセッサで共有できる変数, 配列, そこへのポインタなどを加えた言語であり, C 言語の拡張として定義されている.

まず, 全プロセスで共有される変数, 配列を定義するには, 以下のように大域変数の定義の前に, shared を付ければよい.

```
shared double x;
shared double a[6 * THREADS];
```

ここで THREADS は実行スレッド数 (並列度) を表す UPC の予約語である. MPI のプロセスに相当するものを UPC ではスレッドと呼ぶ.

Shared 変数 (x) は, ランク 0 のスレッドに割り当てられる. Shared 配列 (a) は, 特に指定しなければ一要素ごとのラウンドロビン (サイクリック) に割り当てられる. つまり, ID 0 のスレッドが a[0], a[THREADS], a[2*THREADS], ..., a[5*THREADS], ID 1 のスレッドが a[1], a[THREADS+1], a[2*THREADS+1], ..., a[5*THREADS+1], ... という具合に割り当てられ, 各スレッドが都合 6 要素を保持することになる.

いわゆるブロックサイクリック分割のブロックサイズを指定するには,

```
shared[3] double a[17 * THREADS];
```

のように, shared の後ろに指定する. これで, 3 要素ずつの固まりがサイクリックに割り当てられる.

UPC では異なるスレッド数で実行できるようなプログラムへコンパイルしたい場合 (dynamic translation 環境), 各スレッドが同じだけの要素を持つことが明らかであるような shared 配列だけが許されている. もちろんスレッド数固定でプログラムをコンパイルすることもでき, その場合にはこのような制限はない.

Dynamic translation 環境では, 構文的な制限として, shared 配列の要素数が THREADS の値によらず, その定数倍になることが明らかであるような式だけが, shared 配列の要素数として許されている. 上記の 6 * THREADS, 17 * THREADS などが合法な要素数の例である. このルールの例外は, ランク 0 のスレッドにすべてを割り当てる配列で, それは,

```
shared[] double a[100];
```

のように, ブロックサイズ部分を空にして定義する. 我々の例題で, 結果をランク 0 に集約するための配列は, これを使って定義できる.

```
shared[] sim_result results[n_times];
```

Shared な変数や配列のアドレスをポインタ変数に保持することもできる. そのポインタは通常のポインタとは, コンパイル時に異なる (shared ポインタ) 型として認識され, ローカルな計算に対するオーバーヘッドが少ないように設計されている. また, shared なメモリ領域を動的に割り当てることも可能である. それらについては次号以降で述べる.

Worksharing 構文の upc_forall は, 構文的には for 文を少し拡張したもので, 実行モデルも OpenMP の for プラグマと似ている.

upc_forall(*init*; *condition*; *increment*; *affinity*) 文

という構文で, 最初の *init, condition, increment* の役割は C の for 文のそれらと同じである. またそこに書ける式も, OpenMP の for プログマとほぼ同様の制限がある. 最後の *affinity* は, その繰り返しを実行するスレッド ID を指定する. 正確には, (*affinity % THREADS*) の ID を持つスレッドがその繰り返しを実行する. OpenMP と異なり, static, dynamic のようなキーワードではなく, 各繰り返しを実行するスレッドを直接指定する. 特に, dynamic なスケジューリングはサポートされていない. 分散メモリ計算機を対象としているため, 自然な設計と言える.

9.2 SIMPLE ループの UPC による並列化

以上をまとめると, 我々の SIMPLE ループに, OpenMP の static スケジュールに相当する分割を適用するには,

```
n_per_proc = (n_times + THREADS - 1) / THREADS;
upc_forall (i = 0; i < n_times; i++; n_times / n_per_proc) {
  seed = ...;
  results[i] = init_and_do_steps(H, T, n_sweeps, seed);
}
upc_barrier;
```

とする．shared 配列 `results` への代入を実行するだけで, 明示的な通信のコードが不必要になっている．

upc_barrier は, upc_barrier 以前に行われた shared な領域への更新が, upc_barrier 以降, 他のスレッドから観測されることを保証するための構文である．

上記では OpenMP の static 相当の, いわゆるブロック分割を記述したが, この例題のように負荷の均等化だけに興味がある場合には, より単純に,

```
upc_forall (i = 0; i < n_times; i++; i) {
  seed = ...;
  results[i] = init_and_do_steps(H, T, n_sweeps, seed);
}
upc_barrier;
```

とすることもできる．いわゆるサイクリック分割に相当する．

9.3 NESTED ループの UPC による並列化

多重ループを直接分割する方法はないが, 同じ事を達成するのに affinity 式として, キーワード continue を指定することができる．こうすると, すべてのスレッドがその繰り返しを実行する．このようにして最内ループ以外のループを全スレッドが実行し, 最内ループだけを affinity を用いて分割すればよい．

以下は, 我々の NESTED ループ全体を均等に分割する．

```
int n_per_proc = (n_Hs * n_Ts * n_times + THREADS - 1) / THREADS;
upc_forall (h = 0; h < n_Hs; h++; continue) {
  upc_forall (t = 0; t < n_Ts; t++; continue) {
    upc_forall (i = 0; i < n_times; i++;
                (h * n_Ts * n_times + t * n_times + i) / n_per_proc) {
      double H = H0 + h * dH;
      double T = T0 + t * dT;
      int seed = h * n_Ts * n_times + t * n_times + i;
      results[h][t][i] = init_and_do_steps(H, T, n_sweeps, seed);
    }
  }
}
upc_barrier;
```

これがどう実装されているか, UPC コンパイラの出力結果を見ると, 少なくとも現状の実装は, 各スレッドが全繰り返しを実行した上で, affinity 式と自分のスレッド ID (MYTHREAD) とを比較し

ているようである. ちょうど, 前節で MPI による NESTED ループの分割を述べた時の方法である. 従って繰り返し一回あたりの計算量が少ないとスケーラビリティの足かせとなる.

もちろん UPC は SPMD モデルなので, worksharing for 構文を用いずに, MPI と同様のループに書き換えることもできる. 一方で上記のループから, 各スレッドの担当領域だけを効率的に実行するようなコードへの変換は, 興味深いテーマではないかと思われる.

10 Chapel による並列化

10.1 Chapel の概要

前節までで述べた処理系のうち, 分散メモリマシンをサポートする MPI および UPC は, どちらも SPMD 実行モデルに基づくものであった. 本節と次節で述べる Chapel および X10 は, 分散メモリ環境であっても自由なタスク並列性をサポートする言語であると, 特徴付けることができる.

Chapel は CRAY が開発中の言語で, 特徴を要約すると以下のようになる.

1. Java/C 風だが, 独自の構文を持つ言語で, ポインタ安全なオブジェクト指向言語である.
2. 任意の場所での新しいタスクの生成—いわゆるタスク並列— をサポートする. それとともに, 並列ループ構文 (forall) もサポートしている.
3. UPC に見られるような, 位置を意識しないデータや配列へのアクセスをサポートしている. UPC と異なり, 同じノード内のデータへの参照と, 別のノードにある (かもしれない) データへの参照は, 文法上は区別されない. 極めてデータの分散透明性が高い設計になっている.
4. 他のノードに配置されたデータへの参照は透明に行える一方で, 実際のデータや計算の配置は明示的に制御できる, というよりもしなくてはならない. それを記述するため, プログラム中で, そのプログラムの実行に参加している各計算ノードを表す, 「場所 (locale)」というオブジェクトを参照できる.
5. 計算の配置は, 任意の文を特定の locale に移動して実行できる構文 (on 構文) を用いて指定する. データ (変数) も, 類似の構文で指定した locale 上に割り当てることができる.
6. 代入や関数呼び出しの引数として渡される時に, 参照でなく値がコピーされるデータ構造 (record やタプル), 関数からもどる際に変更が反映される引数など, データをコピーすることで細かい通信やヒープ割り当てを減らす言語仕様が取り入れられている.
7. TBB と同様に, 一次元の範囲 (range), 多次元の矩形を表すデータ構造 (domain) を陽にサポートし, これと for 文を組み合わせて, 多次元矩形に対するデータ並列操作や, 多次元矩形の分割などを記述できる. さらにそれを一般化して, 矩形以外の一般的な domain も定義することができる.
8. 配列は一般に domain から値への写像として定義されており, 一次元配列, 多次元配列, 連想配列などがすべて, 配列として統一的に扱える.
9. Domain を多数の locale にマッピング (分割) したものとして mapped domain という, データ構造がサポートされており, それを domain とする配列がすなわち, 分散配列である.

今回の例題で使うのはさしあたり, タスク並列構文を用いたタスクの生成, on 構文を用いたそのタスクの分散, データ並列構文を用いた locale 内の並列処理である.

10.2 Chapel のタスク並列構文

まずタスク並列にはいくつかの構文がある．一番 primitive なものが begin 構文で, Cilk の spawn を, 関数呼び出しに限らず任意の文に対して行える構文と言えば, 大体の説明になっている．begin によって作られたタスクの終了を待ち合わせるには, sync 変数もしくは single 変数を用いる．そして, その変数への代入が行われるのを待つことができる．例題として, まず逐次の fib 関数を Chapel で記述したものが以下である．

```
proc fib(n : int) : int {
  if (n < 2) {
    return 1;
  } else {
    var x = fib(n - 1);
    var y = fib(n - 2);
    return x + y;
  }
}
```

まず文法について, C 風の構文ではあるがいくつかの特徴をあげておく．

- 関数の引数は型を, 変数名：型名で指定する (1 行目の `n : int`).
- 局所変数宣言, キーワード var で始めるが, 型を毎回宣言する必要はなく, 右辺から推論される (例: 5 行目の `var x = fib(n - 1)`). 宣言あるいは強制したい場合は, 関数の引数と同様の構文で宣言する．初期化式を伴わない変数宣言においては, 型宣言が必須である．
- 関数呼び出しの返り値の型も通常は内部の return 文から推論してくれるが, fib のように再帰呼び出しを伴う関数は宣言が必須である．

さて, それを primitive な begin と sync を用いて並列化したものが以下になる．

```
proc fib(n : int) : int {
  if (n < 2) {
    return 1;
  } else {
    var x : single int;
    begin x = fib(n - 1);
    var y = fib(n - 2);
    return x + y;
  }
}
```

違いは,

- 変数 `x` を int ではなく, single int という型で定義している
- 代入文 `x = fib(n - 1);` の前に begin が追加されている

だけである．Cilk の明示的な sync 文の代わりに, `return x + y` における `x` の参照が, 代入文が実行されるのを待ち合わせる効果を持つ．Chapel のマニュアル [1] では, sync 変数に$ を付けるのを慣例としているようだが, 実際の処理系では$ は必ずしも必要ではないようである．

sync 変数は single 変数と似ているが, 容量が 1 の有限バッファとして機能する．一度値を読むとそのデータは変数から「消費」され, 再びデータが代入されるまで, データの読み出しがブロックするようになる．一方書き込まれたあと, まだ「消費」されていないデータがある変数に代入が行われると, それが消費されるまで代入文がブロックする．この関数 fib においては, single でも sync でもどちらでもよいが, single の方が自然であろう．

多数のタスクを作り, それらすべての終了を待つというパターンは典型的なので, いくつかのショートカットがある．一つは sync 文で,

```
sync {
  ...
}
```

という構文で, ... 内で begin で作られたタスク全てが終了したら終了する．個別のタスクを待つ必要がない場合に簡便な記法である．

似た記法として cobegin がある．cobegin 中に複数の文を並べればそれらを並行に実行して, すべてが終了したところで cobegin が終了する．Fib を cebegin を使って書いたものが以下である．

```
proc fib(n : int) : int {
  if (n < 2) {
    return 1;
  } else {
    var x, y : int;
    cobegin {
       x = fib(n - 1);
       y = fib(n - 2);
    }
    return x + y;
  }
}
```

現状 (ver 1.3.0) で Chapel の実装は, 1 タスクを 1 つの Pthread として実装しており, 上記の関数は実際には Pthread の作りすぎになり, 実際的ではない．それ以外のスレッドパッケージの実装も提供されているが, 現状では 1 ノード (locale) の実行のみがサポートされている．

10.3 ノードの抽象化—Locale

計算を実行する locale は, on 構文で指定する．Locales という組み込みの変数は, 計算に参加している locale の配列であり, 例えば Locales[0] で 0 番目の locale を表すオブジェクトが得られる．Locales の要素数は numLocales という変数に格納されている．

on 式 do 文

または,

on 式 {
  文 ...
}

という構文で, 「式」を評価した結果の locale 上で「文」を実行する. 従って 1 番の locale へ移動するには,

```
on Locales[1] { ... }
```

でよい.

10.4 データ並列 for 構文

次に Chapel では, 通常の for 構文は,

for 変数 in イテレータ do
  文;

もしくは

for 変数 in イテレータ {
  文;
}

という形をしている. for を forall に変えたものが並列 for 文で, 繰り返しが適当なタスクに分割されて実行される.

「イテレータ」部分には配列や範囲を表す式など様々なものが指定できる. 例えば, A が 0 から 99 までの整数を添字とする配列であるとする. まずそれは以下のように宣言する

```
A : [0..99] int;
```

この上で以下はどれも配列の各要素に関数 f を適用する並列 for 文である.

```
forall i in 0..99 {
  f(A[i]);
}

forall i in A.domain {
   f(A[i]);
}

forall a in A {
   f(a);
}
```

10.5 SIMPLE ループの Chapel による並列化

我々の SIMPLE ループを並列実行するプログラムを考える．まず, ノード (locale) 間で分割して実行させる方法を考える．SPMD ではないので出発点は 1 スレッドが実行している状態である．ここからタスク並列によってタスクを作りそれらを on 構文で移動させる．これには明らかに, Cilk の節で述べた方法と似た, 分割統治法が有効である．今回は, 再帰呼び出しに, 計算をすべき繰り返しと共に, それらを実行すべき (実行して良い)locale の範囲も同時に渡す．再帰呼び出し時には, 両方の範囲を分割する．Locale が一つになったら与えられた範囲全体を, その locale で実行するということなので, これ以上再帰呼び出しをする代わりに並列 for 文を使う．ここで locale 内でさらなる分割統治を続けても良いのだが, Chapel の現状の実行時システムは, Cilk や TBB のように, タスク並列を効率的に (Lazy Task Creation を使って) 実装しているわけではない．故に, 実践的には locale 内にタスクを作りすぎない方が良い.

```
// [a, b) の範囲を, locale 番号 [p, q) の locale で実行する
proc run_sims_rec(a : int, b : int, p : int, q : int,
                  H : real, T : real, n_sweeps : int,
                  results : [] sim_result) {
  if (p + 1 == q) {
    // 1 locale だけになったのでノード内で並列 for 実行
    forall i in a..(b-1) {
      var isng = new ising(H=H, T=T, n_sweeps=n_sweeps, seed=i);
      results[i] = isng.init_and_do_steps();
    }
  } else if (a + 1 == b) {
    // 1 要素になったのでそのまま実行
    var isng = new ising(H=H, T=T, n_sweeps=n_sweeps, seed=a);
    results[a] = isng.init_and_do_steps();
  } else {
    // locale の範囲 [p, q) を二つに分割
    var r = (p + q) / 2;
    // それに比例して, [a, b) も分割
    var c = (a * (q - r) + b * (r - p)) / (q - p);
    // そして再帰呼び出し
    cobegin {
      run_sims_rec(a, c, p, r, H, T, n_sweeps, results);
      on (Locales[r]) do
        run_sims_rec(c, b, r, q, H, T, n_sweeps, results);
    }
  }
}
```

最後の再帰呼び出し部分がポイントで,

- Locale の範囲 $[p, q)$ を 2 分割して $[p, r)$ および $[r, q)$ とし,
- それにほぼ比例するように繰り返しの範囲 $[a, b)$ を分割して $[a, c)$ および $[c, b)$ とし,

- $[a,c)$ を locale$[p,q)$ 上で, $[c,b)$ を locale $[r,q)$ 上で評価するよう再帰呼び出しする.
- 再帰呼び出しには cobegin を用い, $[r,q)$ に対する呼び出しを locale r に移動する.

10.6 NESTED ループの Chapel による並列化

NESTED ループに対する変更は, Cilk におけるものを踏襲すればよい. また, Chapel がサポートする domain を使って, 実行すべき添字の集合を表せばすっきりとしたコードになる. まず再帰呼び出し自身のコードは以下だけになる.

```
proc run_sims_rec(dom : domain, locales : range, results : [] sim_result) {
  if (locales.length == 1 || !divisible(dom)) {
    // locale 1 つだけになるか繰り返し領域が分割不能ならば実行
    forall x in dom {
      var (h,t,i) = x;
      var H = H0 + h * dH;
      var T = T0 + t * dT;
      var seed = h * n_Ts * n_times + t * n_times + i;
      var isng = new ising(H=H, T=T, n_sweeps=n_sweeps, seed=seed);
      results[h,t,i] = isng.init_and_do_steps();
    }
  } else {
    // まず locales を二つに分ける
    var p = locales.low;
    var q = locales.high;
    var r = (p + q + 1) / 2;
    // それに比例して領域を分割
    var (sdom_0,sdom_1) = divide_domain(dom, r - p, q + 1 - r);
    cobegin {
      run_sims_rec(sdom_0, p..r-1, results);
      on r run_sims_rec(sdom_1, r..q, results);
    }
  }
}
```

引数として, 繰り返しを行うべき領域 (dom : domain) と, それを実行すべき (locales : range) を, それぞれ一つの引数として受け取っている. Domain は, 矩形領域を含む任意の集合を表すことが可能なデータ構造である. TBB の blocked_range, blocked_range2d などをさらに一般化したものといえる.

Chapel では 1 次元の区間 $[a,b] = \{x \mid a \le x \le b\}$ を表現する range を $a..b$, 矩形 $[a,b] \times [c,d] = \{(x,y) \mid a \le x \le b, c \le y \le d\}$ を表現する domain を, `[`$a..b$`,`$c..d$`]` という簡便な式で記述することができる.

従って上記の関数を呼び出す式は,

```
run_sims_rec([0..n_Hs-1,0..n_Ts-1,0..n_times-1],
             [0..numLocales-1], results);
```

のように書くことができる.

補助関数として divisible という, domain が分割可能かを判定する関数と, そのような domain を実際に分割する関数を記述している. それぞれ以下のように書ける.

```
proc divisible(d : domain) {
  // d.dim(1), d.dim(2), d.dim(3) は矩形の 3 辺の長さ
  if (d.dim(1).length > 1) { return true; }
  if (d.dim(2).length > 1) { return true; }
  if (d.dim(3).length > 1) { return true; }
  return false;
}

proc divide_domain(d : domain, a : int, b : int) {
  var l1 = d.dim(1).length;
  var l2 = d.dim(2).length;
  var l3 = d.dim(3).length;
  if (l1 > 1) {
    // 1 次元目の辺を分割
    var l = d.dim(1).low;
    var h = d.dim(1).high;
    var m = (a * l + b * (h + 1)) / (a + b);
    return ([(l..m-1),d.dim(2),d.dim(3)],[(m..h),d.dim(2),d.dim(3)]);
  } else if (l2 > 1) {
    // 2 次元目の辺を分割
    var l = d.dim(2).low;
    var h = d.dim(2).high;
    var m = (a * l + b * (h + 1)) / (a + b);
    return ([d.dim(1),(l..m-1),d.dim(3)],[d.dim(1),(m..h),d.dim(3)]);
  } else if (l3 > 1) {
    // 3 次元目の辺を分割
    var l = d.dim(3).low;
    var h = d.dim(3).high;
    var m = (a * l + b * (h + 1)) / (a + b);
    return ([d.dim(1),d.dim(2),(l..m-1)],[d.dim(1),d.dim(2),(m..h)]);
  } else {
    assert(false);
  }
}
```

これを直交再帰 2 分割に変更するには, divide_domain だけをわずかに変更すればよい.

11　X10 による並列化

11.1　X10 の概要

X10 は IBM が開発中の言語で, おおまかな特徴は Chapel と共通部分が大きい.

1. Java 風だが, 独自の構文を持つ言語である.
2. 任意の場所での新しいタスクの生成—いわゆるタスク並列— をサポートする. 並列ループ構文は, 現在のバージョン (2.1.2) では削除されている. 並列ループに似た構文としては, 配列に対する map 関数がある.
3. 遠隔データへのアクセス方法は独特で, 遠隔データへのポインタや, 分散配列をサポートするが, ノード外のデータを透明にアクセスすることはできない. そのような参照はコンパイル時または実行時にエラーとなる. ここは Chapel や UPC と大きく異なる点である.
4. Chapel 同様, 実際のデータや計算の配置は明示的に制御できる, というよりもしなくてはならない. それを記述するため, プログラム中で, そのプログラムの実行に参加している各計算ノードを表す, 「場所 (place)」というオブジェクトを参照できる.
5. 計算の配置は, 任意の文を特定の place に移動して実行できる構文 (at 構文) を用いて指定する.
6. 変数を一度だけ代入されて以降 2 度と代入されない変数 (キーワード val で宣言) と, 任意回の代入を許す変数 (var で宣言) に分け, val の変数は place 間を移動する際にコピーされる (遠隔参照が暗黙的に作られることはない). これによって, 別の place へデータをコピーして持ち運ぶコードを簡単に書くことができる.
7. TBB や Chapel と同様に, 一次元の範囲, 多次元の矩形を表すデータ構造 (Region) を陽にサポートし, これと for 文を組み合わせて, 多次元矩形に対する操作や, 多次元矩形の分割などを記述できる. 多次元の点は Point と呼ばれる.

明らかに Chapel と X10 には共通点が多い. 同じ概念に対してことごとく異なる名前がついている. 共通点と相違点を表 3 にまとめた.

11.2　X10 の遠隔参照の扱い

Chapel と X10 の言語設計上の決断で, 決定的に異なっているのは, 遠隔参照の扱いである. 通常のデータ (配列やオブジェクト) が, Chapel では自然に分散透明に異なるノード (locale) から参照され得るのに対し, X10 では通常の配列やオブジェクトは異なるノードから参照されることはない. 具体的な構文で違いを示す. 以下はどちらもノード p で作った配列 a を, 別のノード q に移動して代入し, その後再び p で参照する例である.

Chapel:

```
on (p) {
  var a : [0..9] int; // 配列 a を locale p に作成
  on (q) {
    a[3] = 100;       // q から, p にある配列に書き込み
  }
  writeln("a[3] = %d\n", a[3]); // a[3] = 100
}
```

	Chapel	X10
タスク並列構文	begin	async
タスクの待ち合わせ	sync { ... }	finish { ... }
その他のタスク並列	cobegin, coforall	N/A
並列ループ	forall	N/A
ノードの抽象化	Locale	Place
ノード間移動	on	at
ノード数	numLocales	Place.MAX_PLACES
ノード集合	Locales	Place.places()
多次元配列の添字	タプル	Point
一次元の区間	range	Region
多次元の矩形	domain	Region
不規則な領域	domain	N/A
遠隔参照	透明	明示的

表 3: Chapel と X10: 共通点, 語彙の違い, 相違点

X10:

```
at (p) {
  val a = new Array[Int](10);
  at (q) {       // ここで a はまるごとコピーされる
    a(3) = 100;  // この a は p の a とは別物
  }
  Console.OUT.printf("a(3) = %d\n", a(3)); // a(3) = 0
}
```

Chapel での挙動はいわゆる共有アドレス空間のそれであり, q で代入した値 100 が, p でも観測される. X10 では, at 構文を実行する際, a はコピーされて, 実行される. したがって p の配列は初期状態 (0) のままである. X10 の at 構文のルールは, at (p) S という文において, S の外側で定義されている変数のうち, val を用いて定義されているものは p にコピーされて実行される, というものである. 従って p と q 内でデータを共有するわけではない. var で定義された変数 (任意回変更可能な変数) は, それが定義された place 内でしか参照できないよう, コンパイル時にチェックされる. 例えば上記の a を var で定義すると, 以下のようなコンパイル時エラーになる.

```
Local variable "a" is accessed at a different place, and must be declared final.
```

X10 で, 異なる place にデータへの「参照」を渡したければ, そのデータを明示的に GlobalRef という大域参照で包む必要がある. 以下は, 配列 a を作り, それへの参照を大域参照で包むコードである.

```
val a = new Array[Int](10);
val ga = new GlobalRef[Array[Int](1)](a);
```

大域参照は,

```
new GlobalRef[型](参照されるオブジェクト)
```

という表記で作る. 型の部分にある, `Array[Int](1)` は, 1 次元の Int の配列の型を表している.

大域参照 ga から, 元のオブジェクトへの参照を取り出すには, 演算子 `()` を用いる.

```
val orig_a = ga();
orig_a(3) = 100;
```

ただし, この参照 ga() が成功するのは a が存在する place においてのみである. 従って以下のコードはコンパイル時にエラーになる.

```
at (Place.places()(0)) { // 0 番の place で...
  // 配列 a とそれへの大域参照 ga を作り...
  val a = new Array[Int](10);
  val ga = new GlobalRef[Array[Int](1)](a);
  // place 1 で参照
  at (Place.places()(1)) {
    val a = ga();
    a(3) = 100;
  }
  Console.OUT.printf("a(3) = %d\n", a(3));
}
```

参照 ga() が成功することを「コンパイル時に」保証するには, ga の存在する place に明示的に移動する必要がある. 以下でようやく, place 0 で作られた配列を place 1 から書き換えることが可能になる.

```
at (Place.places()(0)) { // 0 番の place で...
  // 配列 a とそれへの大域参照 ga を作り...
  val a = new Array[Int](10);
  val ga = new GlobalRef[Array[Int](1)](a);
  // place 1 で参照
  at (Place.places()(1)) {
    at (ga) {
      val a = ga();
      a(3) = 100;
    }
  }
  Console.OUT.printf("a(3) = %d\n", a(3));
}
```

分散配列の場合も, その要素を保持する place においてのみ参照や代入が成功する. しかし分散配列の場合それをコンパイル時に保証することはない. これについては次回以降で述べる.

11.3 SIMPLE ループの X10 による並列化

さて, SIMPLE ループを並列化する際の作戦は Chapel と殆ど同様である. つまり, 繰り返しの添字をまず place 間で再帰的に分割して行く. これにはタスク並列構文 async と, place 間の移動 at を組み合わせる. 1 place に割り当てられた部分の並列実行は, Chapel と異なり並列ループに相当するものはサポートされていないので, それにもタスク並列 async を用いる.

```
static def run_sims_rec(a : Int, b : Int, p : Int, q : Int,
                        H : Double, T : Double, n_sweeps : Int,
                        results : GlobalRef[Array[sim_result](1)]) {
  if (a + 1 == b) {
    val isng = new ising(H, T, n_sweeps, a);
    isng.init_sigma();
    val res = isng.init_and_do_steps();
    at(results) {
      results()(a) = res;
    }
  } else if (p + 1 == q) {
    val c = (a + b) / 2;
    finish {
      async run_sims_rec(a, c, p, q, H, T, n_sweeps, results);
      run_sims_rec(c, b, p, q, H, T, n_sweeps, results);
    }
  } else {
    val r = (p + q) / 2;
    val c = (a * (q - r) + b * (r - p)) / (q - p);
    val there = Place.places()(r);
    finish {
      async at(there) {
        run_sims_rec(c, b, r, q, H, T, n_sweeps, results);
      }
      run_sims_rec(a, c, p, r, H, T, n_sweeps, results);
    }
  }
}
```

特徴的なのは結果を書き込みに行く部分

```
at(results) {
  results()(a) = res;
}
```

である. 大域参照が指している場所へ移動 (`at(results)`) し, そこでその大域参照の本体を取り出し (`results()`), ようやく答えを代入する.

NESTED もほぼ, Chapel の場合と同様である. Chapel と違うのは言葉遣いくらいである.

```
def run_sims_rec(reg : Region, places : Region,
                 results : GlobalRef[Array[sim_result](3)]) {
  if (reg.size() == 1) {
    val [h,t,i] = [reg.min(0), reg.min(1), reg.min(2)];
    val H = H0 + h * dH;
    val T = T0 + t * dT;
    val seed = h * n_Ts * n_times + t * n_times + i;
```

```
      val isng = new ising3(H, T, n_sweeps, seed);
      isng.init_sigma();
      val res = isng.init_and_do_steps();
      at (results) {
        results()(h,t,i) = res;
      }
    } else if (places.size() == 1) {
      val [reg_0,reg_1] = divide_region(reg, 1, 1);
      finish {
        async run_sims_rec(reg_0, places, results);
        run_sims_rec(reg_1, places, results);
      }
    } else {
      val p = places.min(0);
      val q = places.max(0);
      val r = (p + q + 1) / 2;
      val [reg_0, reg_1] = divide_region(reg, r - p, q + 1 - r);
      val there = Place.places()(r);
      finish {
        async at(there) {
          run_sims_rec(reg_0, p..(r-1), results);
        }
        run_sims_rec(reg_1, r..q, results);
      }
    }
  }
```

12 性能測定

ここで紹介した処理系に加え, 逐次の C 言語を加えた合計 8 種類の処理系で, 今回述べた問題を用いた性能評価を行った. HA8000 を用いて, 16 ノード ×16 コア (256 並列) までの評価を行っている.

12.1 パラメータ

- 系の大きさ (スピンの数): 64×64
- 一シミュレーションのステップ数: $1000 \times 64 \times 64$
- シミュレーションの回数: 1024
- 外部磁場 (H): 0.0
- 相互作用エネルギー (J): 1.0
- 温度 (T): 1.0

大雑把に行って, 一回のシミュレーションが HA8000 上の C プログラムで 0.4 秒程度の処理である.

また, どのプログラムも並列度によらず完全に同じ結果を返し, プログラム間でも結果は完全に一致する. 言い換えれば乱数の種は 1 回のシミュレーション毎に決められたものを用い, シミュレーション間では (同一スレッド内でも) 一切共有していない.

C, OpenMP, Cilk, TBB, MPI, UPC では erand48, nrand48, jrand48 相当のもの (なぜ「相当」かは下記参照) を用いている. それらは以下のインタフェースを持つ (unsigned short の 3 要素の配列を受け取る) C の関数である.

```
long int nrand48(unsigned short xsubi[3]);
long int jrand48(unsigned short xsubi[3]);
double erand48(unsigned short xsubi[3]);
```

Chapel では, 同一の方を並べたタプル (homogeneous tuple) は, C の配列と同じデータ表現を持つ. 従って, 以下の宣言でそれらの関数を, データ変換なしに呼び出すことができる. [1]

```
_extern proc jrand48(inout x : 3*uint(16)) : int(64);
_extern proc nrand48(inout x : 3*uint(16)) : int(64);
_extern proc erand48(inout x : 3*uint(16)) : real;
```

この, C の配列に相当するデータが Chapel でそのままアクセスできるという性質は, 様々な場面で有用となるであろう.

X10 にも外部関数呼び出し機能があるが, X10 のデータ型で, C の 3 要素配列に相当するものはない. そこで, X10 においては X10 で C の jrand48, nrand48, erand48 に相当する関数を自作している. それを行ったところ, 逐次性能は C 言語を上回った. おそらく C 言語でそれらの関数を動的リンクライブラリ関数から呼び出すよりも, X10 で同一プログラム内に書かれた関数を呼び出す (あるいはインライン展開される) 方が速いということである. そこで最終的な評価プログラムでは, C 言語でもそれらの関数をプログラム内に記述して, インライン展開可能とした. それにより再び C 言語の性能が X10 を上回った. なお, Chapel においても同様のことを試みたが, Chapel においては C 言語の外部関数を呼び出すものと比較して性能向上は見られなかった.

次節の結果はこれをおこなったあとのものである.

12.2 逐次性能

まずそれぞれの言語で書かれたプログラムを並列度なしで実行したものと, C 言語で書かれた純粋な逐次プログラムとを比べたものが図 1 である.

高水準言語というと多くの場合逐次性能が問題になるが, X10 のオーバーヘッドは C と比較して 25%程度, Chapel も 2 倍以内のオーバーヘッドであり, 多くの処理に対して十分実用的な速度である. その他の処理系は型システムも C 言語のそれと同じであり, ある意味では当然の結果と言えるが C 言語と比べたオーバーヘッドはほとんどない.

なお, Chapel, X10 とも, 最適化オプションで配列の添字チェックなどを外すオプション (付録 H 参照) をつけないと, 性能は数倍悪くなる. それらの言語のマニュアルにも, 性能評価はそれらのオプションを必ずつけて実行するよう, 推奨されている. X10 には C++へのコンパイルと Java へのコンパイルの二つの処理系があるが, ここで用いたものは前者で, 性能評価をするなら前者を用いるよう推奨されている.

[1] 配布物中の, chapel-1.3.0/chapel/doc/technotes/README.extern

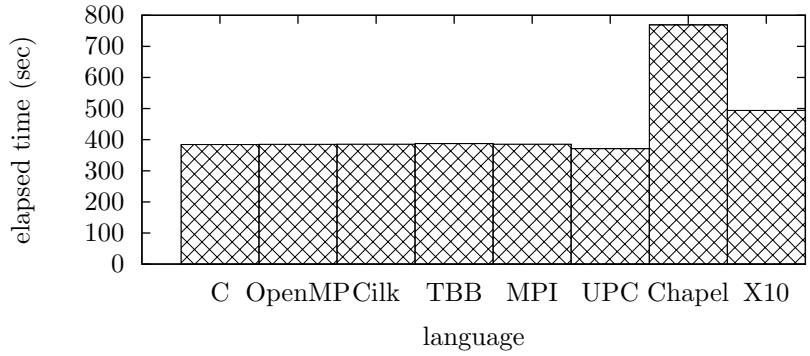

図 1: 逐次 (1 ノード, 1 コア) での実行時間

12.3 1 ノードでの台数効果

図 2 は, 1 ノード内でスレッドを増やした際の台数効果を示したものである. MPI の場合はプロセスを増やし, それ以外の処理系では 1 プロセス内のスレッドを増やしている (起動の仕方の詳細については付録 B, . . . , H を参照).

プログラムの性質を考えればあまり驚くことではないが, 多くのケースでほぼ 100% 近い台数効果が出ている.

- Chapel は 15 台まではほぼ 100%だが, 16 台のときのみ急激に 50%程度まで落ち込んでいる.
- X10 は 16 台で 12 倍弱. 効率がだいたい 60%程度

前者について, Chapel が実行中のシステムの負荷を観測すると, 常に指定した数の並列度に加えて, システム負荷が 1 コア分近く上昇するという現象が観測された. この根本的な理由についてはまだ特定できていない. 15 コアまでほぼ完全に台数効果が得られていることを考えると, 16 コア時の性能劣化はこれが関係していると考えられる.

後者についてだが, この結果に至る過程で遭遇した現象として, 当初 X10 のスケーラビリティがほとんど得られなかった. それはどうやら乱数の状態を保持する配列 (16 ビット整数が 3 要素) が, ヒープ上同一キャッシュラインに割り当てられていた事 (false sharing) が理由のようである. 16 ビット整数 3 要素を, シミュレーションの状態全体を表すオブジェクト内に直接 (整数の要素 3 つとして) インラインに割り当てることで台数効果は大きく向上し, ここに書いた性能になっている. それでも効率が 100%近くにならない理由は, 特定できていない.

前節までで述べたとおり, Cilk と X10 においては完全にタスク並列構文のみで書かれている. Cilk では spawn, X10 では async 構文がそれぞれ 1024 回呼び出されている. X10 におけるタスク並列の実装がそれなりにしっかりしている事の表れである.

12.4 複数ノード・1 コア/ノードでの台数効果

図 3 は, 各ノードで用いるコア (プロセスやスレッド) を 1 にしてノード数だけを増やした際の台数効果である.

こちらは, 該当するすべての言語で 16 ノードまで良好な性能が得られている.

この結果を得る過程での唯一の「サプライズ」は, 当初 Chapel の台数効果が全く得られなかったことであった. 理由はプログラム中にアニメーション用の途中結果を出力する・しないを指定する

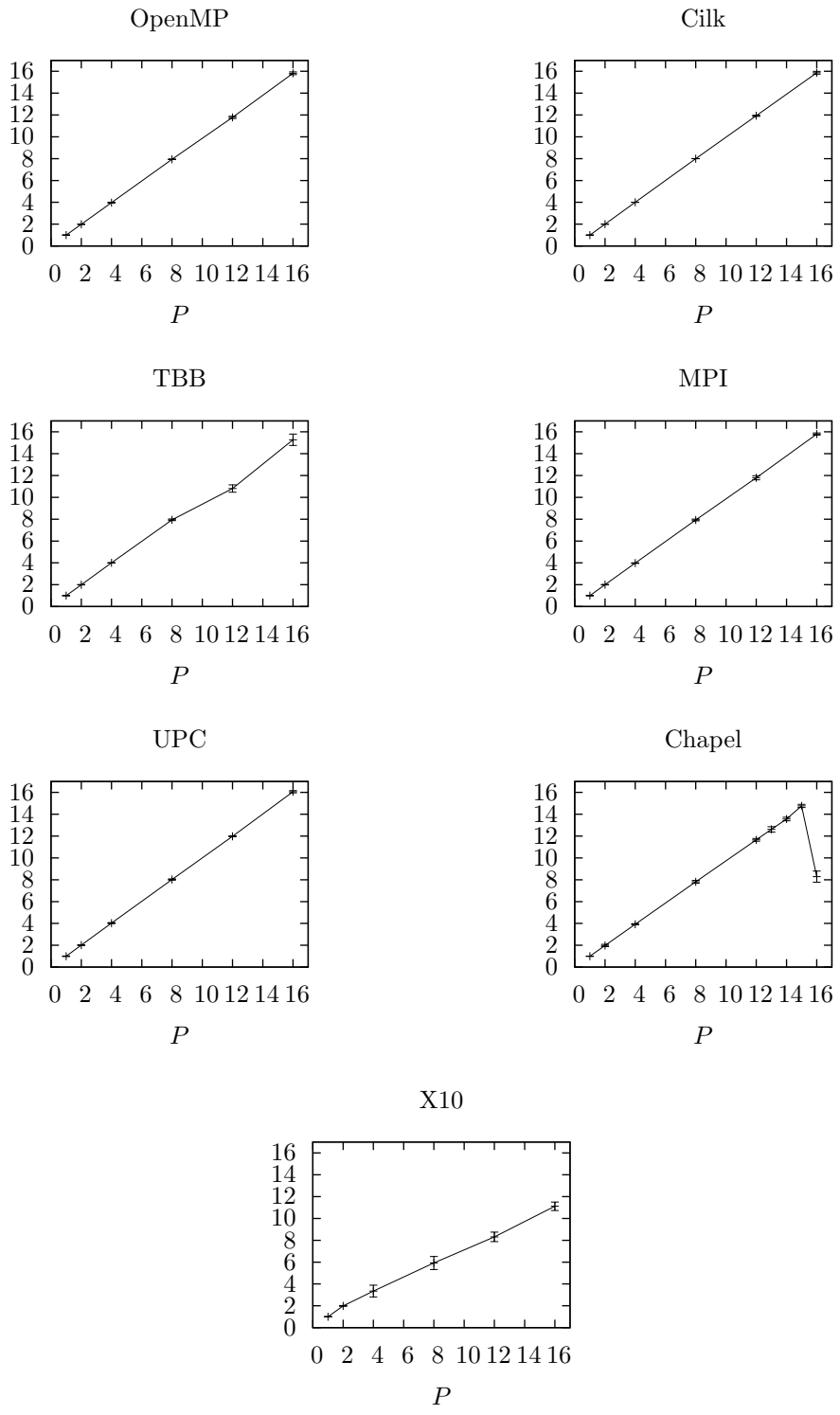

図 2: 1 ノードでの台数効果. P は並列度 (プロセス数やワーカスレッド数). 数値は, 同一プログラムに対する台数効果

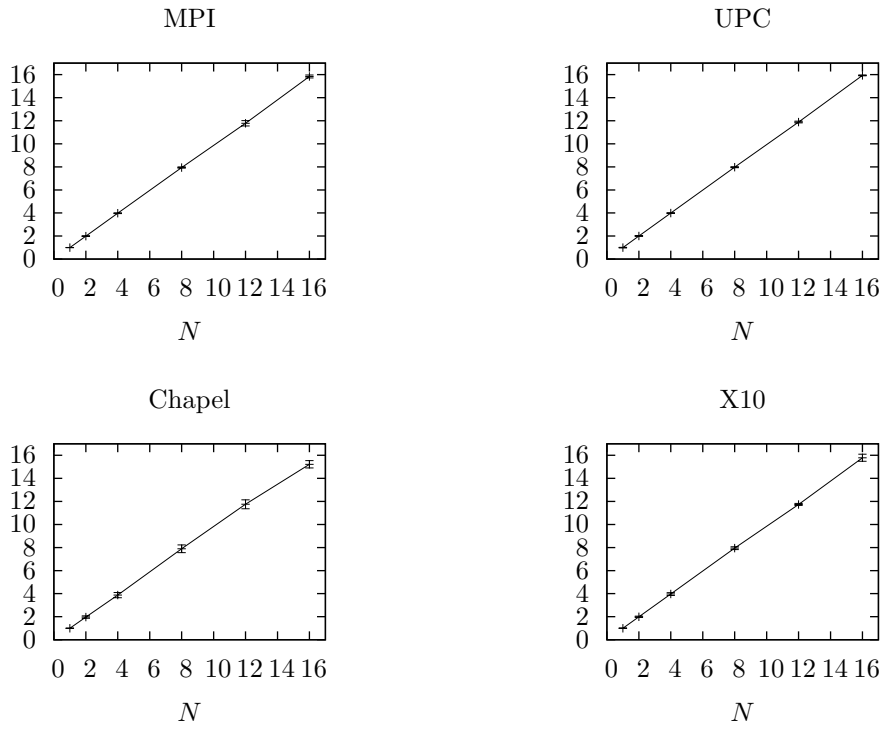

図 3: 複数ノード (1 コア/ノード) での台数効果

Read Only なフラグ変数が, 常に 0 番ノードへの通信を引き起こしていたせいであった. ある意味で Chapel の分散透明性が仇となったとも言えるが, 直し方も簡単で, その変数の宣言を

```
config var v = ...;
```

から,

```
config const v = ...;
```

と変更するだけである.

なお Chapel には, 通信の回数を報告する API [2]があり, これを用いて原因を特定することが出来た.

12.5 複数ノード・複数コア/ノードでの台数効果

図 4 に, 複数ノード, 複数コア (1 ノードにつき, 8, 12, 16 スレッド) での台数効果を示した. 総じて, 1 ノード複数コアでの台数効果と, 複数ノード・1 コア/ノードでの実行結果とから予測される値と, よく一致しているが, Chapel においては, 12 コア/ノードの際の台数効果の伸びが悪い (16 ノード × 12 コア =192 並列で, 約 150 倍). 現在はまだ理由を特定できていないが, 次回以降に究明できればと考えている.

[2]配布物中の, chapel-1.3.0/chapel/doc/technotes/README.comm-diagnostics

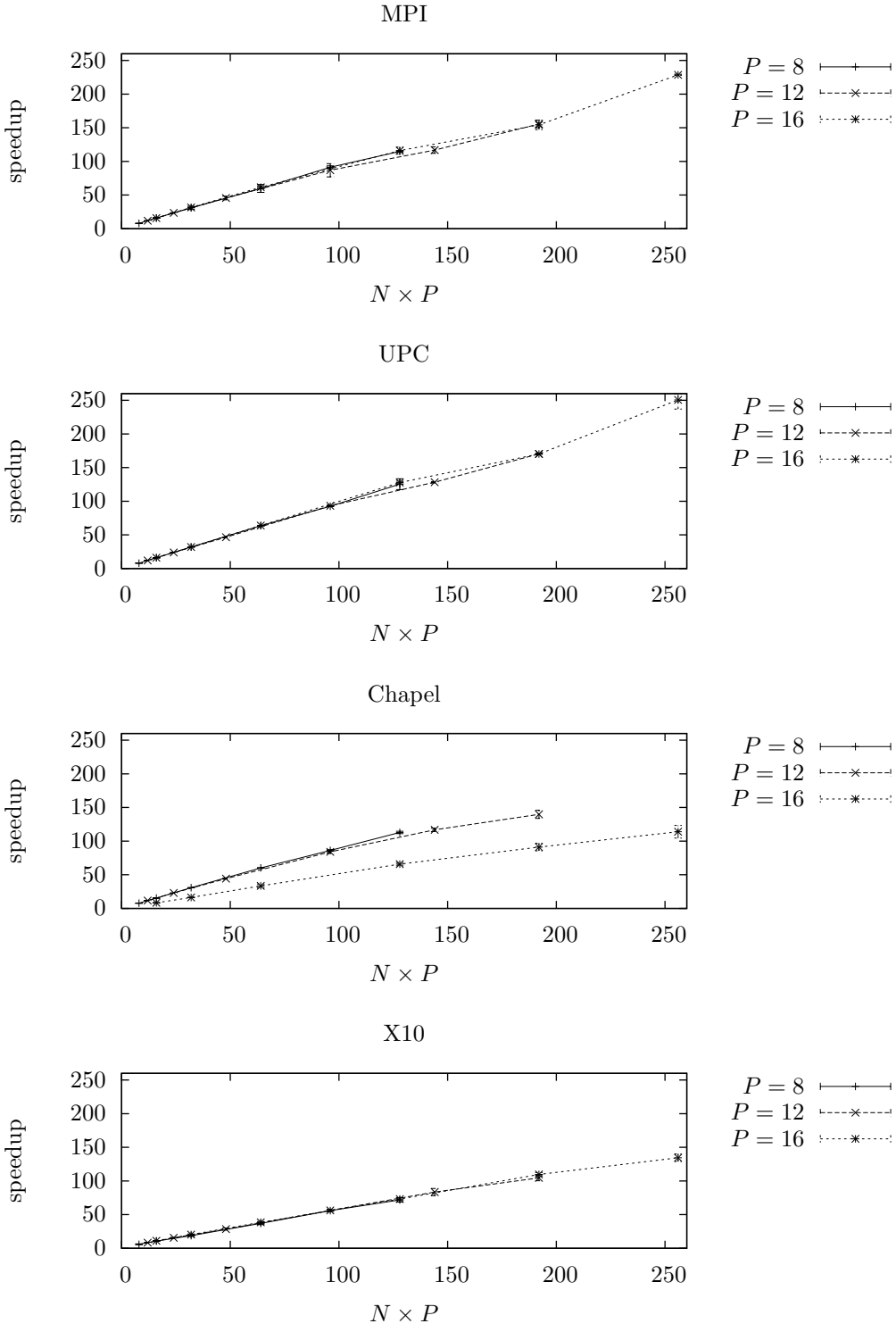

図 4: 複数ノード (8,12,16 コア/ノード) での台数効果

表 4: 各処理系のバージョン

言語	処理系・バージョン	構築に用いたコンパイラ	使用した MPI 実装
C	gcc 4.5.2		
OpenMP	gcc 4.5.2		
Cilk	MIT Cilk 5.4.6	gcc 4.5.2	
TBB	tbb30_20100915oss_lin.tgz	gcc 4.5.2	
MPI	HA8000 上の mpich-mx-gcc		
UPC	Berkeley UPC 2.12.1	gcc 4.5.2	mpich-mx-gcc
Chapel	Chapel 1.3.0	gcc 4.1.2	mpich-mx-gcc
X10	X10 2.1.2	gcc 4.5.2	mpich2-mx-gcc

参考文献

[1] The Chapel parallel programming language. http://chapel.cray.com/.

[2] The Cilk project. http://supertech.csail.mit.edu/cilk/.

[3] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 2:264–280, July 1991.

[4] The message passing interface (MPI) standard. http://www.mcs.anl.gov/research/projects/mpi/.

[5] *OpenMP Application Program Interface Version 3.0.* http://www.openmp.org/mp-documents/spec30.pdf.

[6] Intel Threading Building Blocks 3.0 for open source. http://www.threadingbuildingblocks.org/.

[7] Unified Parallel C. http://upc.gwu.edu/.

[8] X10. http://x10.codehaus.org/.

[9] 田浦健次朗. 連載講座: 高生産並列言語を使いこなす (1). Technical Report 1, 東京大学情報基盤センター, 1 月 2011.

A 各処理系の仕様バージョン

表 4 に, 用いた処理系のバージョンを示す. C, OpenMP, Cilk, TBB は 1 ノード内の処理系で, 構築も実行方法も単純である. MPI は HA8000 にもともとインストールされているもので, 特段の構築作業はしていない. 以下の節で, 個々の処理系の構築の際の設定や, 実行に用いたパラメータを示す.

B OpenMP の実行方法

ワーカスレッド数の指定には環境変数 `OMP_NUM_THREADS=`を設定する. P ワーカスレッドでの実行には以下のようにする.

`OMP_NUM_THREADS=`P *program*

C　Cilk の実行方法

ワーカスレッド数の指定には, プログラム起動時に-nproc オプションを指定する. P ワーカスレッドでの実行には以下のようにする.

program -nproc P

D　TBB の実行方法

ワーカスレッド数の指定には, API の task_scheduler_init を用いる. このクラスのオブジェクトを作るときに指定した値が, そのオブジェクトが生存している間有効になる. プログラム実行中 P ワーカスレッドで実行したければ, 並列実行が始まる前に,

new task_scheduler_init(P);

とする. `<tbb/task_scheduler_init.h>`を include する.

E　MPI の実行方法

HA8000 では色々な MPI とコンパイラの組み合わせが用意されており, /opt/itc/mpi/mpiswitch.sh を source することで, PATH が適切に設定される.

```
sh /opt/itc/mpi/mpiswitch.sh
```

とすれば利用可能な MPI が列挙されるが, 今回は mpich-mx-gcc を用いた. そのためにはバッチファイル内に,

```
. /opt/itc/mpi/mpiswitch.sh mpich-mx-gcc
```

と書けば, 適切な mpirun が PATH に入る.

UPC, Chapel, X10 を正しく起動するために, HA8000 上の MPICH のプロセス配置について, 説明する. HA8000 環境では, MPI のプロセス配置は基本的には, バッチスケジューラに要求したノード数 N と, ppn (process-per-node) の値で決まる.

- 最大で N × ppn までのプロセスを作成できる (mpirun の-np の引数に指定できる)
- 通常はもともと, N × ppn の値を-np に指定するが, 一回のバッチジョブ中に複数回, 異なる並列度で実行する時など, N × ppn 未満の値を指定したいこともある. その場合 mpirun は, なるべく少ないノードにプロセスが収まるように, プロセスを配置する. つまり, 1 個目から ppn 個までのプロセスが 1 つ目のノードへ, (ppn + 1) 個目から 2 ×ppn までのプロセスが 2 つ目のノードへ, ... という具合にプロセスが配置される.
- 従って, ppn 個未満のプロセスを複数のノードに配置することは, mpirun ではできない.
- mpirun の多くは内部で mpiexec (/opt/torque/bin/mpiexec) を呼んでおり, これを直接を呼ぶと, より細かくプロセス配置が指定できる.

  `mpiexec -n` N `-npernode` P *program*

  で, N 個の各ノードに P プロセスを配置できる (合計 NP プロセス). MPI の実装により, mpiexec にオプションを渡している. mpirun が mpiexec をどのように呼んでいるかは, `/opt/itc/mpi/*/bin/mpirun` を見るとわかる.

F　UPC の構築と設定

F.1　処理系の構築

HA8000 にインストールされている gcc (/usr/bin/gcc) のバージョンは, 4.1.2 で, それを用いると構築時に, 4.0, 4.1, 4.2 はバグがあるので用いるなという旨のエラーが出る.

```
checking for known buggy compilers... CC is gcc 4.x, for x < 3
configure: error: Use of gcc/g++ 4.0, 4.1 or 4.2 for compiling this
software is strongly discouraged. ...
```

そこで構築には gcc 4.5.2 をビルドして用いている. UPC は SMP, UDP, MPI, MX (Myrinet) など, 用いる通信レイヤを選択できる. 処理系の構築時, PATH に用いたい MPI (mpicc) がはいっている状態で, configure をすれば, その MPI が使われるようになる.

F.2　コンパイル

最適化オプションと, ノード内で Pthreads を用いて UPC スレッドを立ち上げるオプション`-pthreads`を用いる.

`upcc -O -pthreads` *program*.c

F.3　起動

UPC プログラムを起動するのは upcrun というランチャである.

`upcrun -n` $N \times P$ `-pthreads` P

で, UPC スレッドが $(N \times P)$ 個立ち上がる. `-pthreads` P は, 一ノードにスレッドを P 個, Pthreads を用いて起動するオプションである.

これを MPI をランチャとして起動するには, `mpiexec` コマンドに対して, 各ノードには 1 プロセスだけを立ちあげるよう指示する必要がある. `upcrun` がどのように MPI を立ち上げるかは, 環境変数 `MPIRUN_CMD` を設定することで制御できる. ここに, `-npernode 1` を指定すればよい. 結論として以下のように起動することで, 1 ノードにつき P 並列, それを N ノード, という実行ができる.

`MPIRUN_CMD="mpiexec -n %N -npernode 1 %C" upcrun -n` $N \times P$ `-pthreads` P

G　Chapel の構築と設定

G.1　処理系の構築

Chapel も UPC 同様, 通信レイヤとして UDP や MPI などから選択できる. デフォルトは UDP で, しかも処理系構築時に選択しなくてはならない. 以下の二つの環境変数を設定した状態で処理系をビルドすると, MPI を通信レイヤとして使うようになる.

```
CHPL_COMM=gasnet
CHPL_COMM_SUBSTRATE=mpi
```

なお, タスクの実装に Pthreads を使うのがデフォルトだが, 1 ノードでのみ実行する処理系であれば, nanox, Qthreads など, より軽量なスレッドを選択できる. しかし今回は複数ノードでの実行が必須なので, Pthreads を用いている.

G.2 コンパイル

コンパイル時には, `CHPL_HOME`, `CHPL_COMM`, `CHPL_COMM_SUBSTRATE` という 3 つの環境変数を指定する. 後ろ二つはビルド時と同じものを指定する. `CHPL_HOME` は, Chapel インストール時にできる chapel-1.3.0/chapel ディレクトリを指すようにする. `--fast` オプションが, 性能評価をするにあたっては必須の最適化オプションである.

`CHPL_HOME=`*chapel-home* `CHPL_COMM=gasnet CHPL_COMM_SUBSTRATE=mpi chpl --fast` *program*`.chpl`

G.3 起動

この設定で構築した処理系で Chapel プログラムをコンパイルすると, MPI プログラムとして起動できる実行可能ファイルが生成される. その実行可能ファイルは locale の数を`-nl` というオプションで指定でき, locale 内で forall を実行する際のスレッド数を, `--dataParTasksPerLocale` というオプションで指定できる. UPC 同様, MPI に対しては 1 ノードに 1 プロセスだけを立ちあげるよう指示する. まとめると以下のように起動する.

`mpiexec -n` N `-npernode 1` *program* `-nl` N `--dataParTasksPerLocale` P

G.4 参考ドキュメント

Chapel のソースツリー中のテキストファイルに様々な情報が書かれている.

- 構築に関して: chapel-1.3.0/chapel/doc/README.chplenv
- ランチャ, 通信基盤の選択: chapel-1.3.0/chapel/doc/README.multilocale
- 並列度調整のためのオプション: chapel-1.3.0/chapel/doc/README.executing

H X10 の構築と設定

H.1 処理系の構築

X10 も MPI や sockets などいくつかの通信レイヤを選択可能だが, sockets は SSH による起動を前提としているため, HA8000 では MPI を用いる. それには用いたい MPI をパスに入れて, ビルド (ant dist) に`-DX10RT_MPI=true` を与える. また, MPI は MPI2 が必要である.[3] そこで X10 が用いる MPI は, mpich2-mx-gcc としている.

[3] http://x10.codehaus.org/X10RT+Implementations

H.2 コンパイル

MPI を用いるプログラムであることを明示するため, `-x10rt mpi` とういオプションを与える.

```
x10c++ -O -NO_CHECKS -x10rt mpi program.x10
```

Chapel 同様, `-O -NO_CHECKS` は性能評価をするのであれば必須のオプションである.

H.3 起動

いくつかのチュートリアルによると, `x10c++`を使って生成したバイナリは, `runx10` というコマンドを用いて起動するという記述があるが, そのようなプログラムは, 配布物に含まれていない. X10Launcher というランチャが同梱されているが, これは sockets を用いた場合のランチャのようである. `-x10rt mpi` を指定して作った実行可能ファイルは, そのまま MPI プログラムとして起動すれば良いようである.

X10 プログラムのノード内並列度 (スレッド数) は, 環境変数 `X10_NTHREADS` で指定する. これだけだと, 状況によりまだ動的にスレッドを生成する. それを抑止するために, `X10_STATIC_THREADS=1` とする. Chapel や UPC と同様, mpiexec には 1 ノード 1 プロセスの MPI プログラムとして起動する.

```
X10_NTHREADS=P X10_STATIC_THREADS=1 mpiexec -n N -npernode 1 program
```

H.4 参考情報

- http://x10.codehaus.org/X10RT+Implementations
- http://x10.codehaus.org/For+Users
- http://dist.codehaus.org/x10/documentation/languagespec/x10-env.pdf
- http://x10.codehaus.org/Performance+Tuning+an+X10+Application