特集
マルチコアを活かすお手軽並列プログラミング

4 並列計算パターン（スケルトン）による並列プログラミング

岩崎 英哉（電気通信大学）
胡　振江（国立情報学研究所）

並列計算パターンとスケルトン

マルチコアCPUを搭載した計算機が，デスクトップPC，ノートPCを問わず一般的に用いられるようになった現在，マルチコア資源を有効に活用するようなプログラミングを行いシステムを開発することが求められている．計算ユニットであるコアを複数活用するわけであるから，必然的に複数の制御フローを持つプログラムが必要となる．これらの制御フローは，互いに協働し，必要に応じて計算結果などの情報を交換して計算を進める．したがって，メモリ空間が独立している複数のプロセスを用いるよりも，同じメモリ空間を共有するスレッドを用いる，すなわちマルチスレッドを利用するのが，マルチコアと相性が良い．

マルチスレッドを利用してマルチコア用のプログラミングを行う場合，以下のような2通りの方針が考えられる．

- POSIXスレッド（pthreads）などのネイティブスレッドを利用する．
- 並列性を隠蔽したライブラリを利用し，スレッドを陽に扱うコーディングを避ける．

前者の方針を採用した場合，きめ細かで低レベルのスレッド制御が可能なので，注意深く作成されたプログラムは，マルチコアの潜在能力を十分に引き出すことができる．その一方で，並列プログラミングの難しさの一般的な問題，すなわち，(1) どのように仕事をスレッドに分割しスケジューリングするのがよいかという設計上の問題，(2) スレッド間の同期，デッドロックなどの相互作用の問題，を抱えている．

これらに加え，マルチコアの性能を十分に引き出すためには，コア間で共有されているキャッシュメモリを考慮する，すなわち1つのコアがキャッシュに読み込んだデータを別のコアが使うといったように，キャッシュに関する最適化を行うことも必要となる．結局，前者の方針に基づき，マルチコアに最適化された高い性能を持つプログラムを作成するには，きわめて多大な労力が必要であり，デバッグ，保守も難しいといえよう．

後者は，マルチスレッドプログラミングの手間の軽減に重点を置いており，その根底には，「マルチコア向けの最適化を十分に行ったプログラムと比較して性能面で劣ることはやむを得ない．それより，プログラムを楽に記述でき，そこそこの性能が得られれば良しとする」という考えがある．

並列性を隠蔽したライブラリを用いたプログラミングとして有効であると期待されているのが，スケルトン並列プログラミング（Skeletal Parallel Programming）[1]である．スケルトン並列プログラミングでは，「並列スケルトン（Parallel Skeleton）」あるいは単に「スケルトン」と呼ばれる，並列計算に頻出する並列処理パターンを組み合わせることにより，プログラムを記述する．それぞれのスケルトンは，並列動作を内部に隠蔽しているので，プログラマは，逐次プログラムを作成するような感覚で並列プログラムを作ることができる．

本稿では，スケルトンのような並列処理パターンに基づくC++用の並列プログラミングシステム（ライブラリ）を2つ — Intel Threading Building Blocks[2), 3)]とSkeTo[4)]— を紹介する．

基本的な並列計算パターン

ライブラリの各論に入る前に，基本的な並列計算パターンのいくつかを具体的に紹介する．どのような並列計算パターンを用意するかは，ライブラリの設計にかかわる重要な問題であるが，本稿で紹介するものは，多くの並列スケルトンシステムに採り入れられており，それら

の組合せは高い表現力を持つことが経験的に実証されている．

並列スケルトンは，データ並列スケルトンとタスク並列スケルトンに分類できる．本特集の1つ目の解説における parallel-for ベースの方法はデータ並列スケルトンに，fork-join ベースの方法はタスク並列スケルトンに分類される．

データ並列スケルトンは，全体を構成する個々のデータに対して均一処理を並列に施す．ここでは，データの1次元の並び（$[x_1, x_2, \ldots, x_n]$ という形のリストで表記）を例にとり，代表的なデータ並列スケルトンを3つ，map，reduce，scan を紹介する．実際の C++ でのライブラリにおいては，リストは配列のようなデータ構造で実現される．

map スケルトンは，リストの各要素に同じ関数を適用したリストを生成する[☆1]．reduce スケルトンは，リストの各要素の間に結合的な二項演算子を挟んで畳み込んだ計算結果を返す．scan スケルトンは，reduce の途中結果をリストとして返す．

$$\mathrm{map}(f, [x_1, x_2, \ldots, x_n]) = [f(x_1), f(x_2), \ldots, f(x_n)]$$
$$\mathrm{reduce}(\oplus, [x_1, x_2, \ldots, x_n]) = x_1 \oplus x_2 \oplus \ldots \oplus x_n$$
$$\mathrm{scan}(\oplus, [x_1, x_2, \ldots, x_n]) = [x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \ldots \oplus x_n]$$

タスク並列スケルトンは，独立した計算を並列に行うことをパターン化したものである．たとえば，データのストリームに対して並列処理を施した結果をデータストリームとして返すような計算パターンは，タスク並列スケルトンである．そのようなスケルトンの例に pipe がある．pipe スケルトンは，関数で表現された2つの計算 f と g をパイプライン的に結合し，与えられたストリームに対して f と g を順番に適用した結果のストリームを返す．

$$\mathrm{pipe}(f, g, \langle x_1, \ldots, x_n \rangle) = \langle g(f(y_1)), \ldots, g(f(y_n)) \rangle$$

ここでは，n 個のデータからなるストリームを $\langle x_1, \ldots, x_n \rangle$ と表記している．また，fork-join 型の計算の典型例である分割統治法をスケルトンとして用意しているシステムもある．

Intel Threading Building Blocks

Intel Threading Building Blocks[☆2]（以下 TBB）は，インテル社が提供するマルチコア向け並列計算パターン用 C++ ライブラリであり，本稿執筆時点（2008年11月）での最新バージョンは 2.1 である．インテル社は TBB において「並列スケルトン」という言葉を使ってはいないが，並列計算パターンの組合せによる並列プログラミングをサポートしているという点において，並列スケルトンライブラリの一種と捉えることができる．TBB それ自体は単一の計算機（ノードと呼ぶ）からなる環境を対象としているため，PC クラスタのような複数の計算機からなる環境で利用するためには，MPI（Message Passing Interface）などのライブラリを併用する必要がある．以下では，単一ノードにおける TBB プログラミングを解説する．

TBB では，スレッド管理を抽象化することで，手軽な並列プログラミングを可能としている．具体的には，スレッドそのものを記述するかわりにスレッドで実行すべき仕事（タスク）を記述する．タスクをどのようにしてスレッドへ割り当てるか，すなわちタスクのスケジューリングは，すべてライブラリが提供するタスクスケジューラに一任し，プログラマはスレッドを直接操作することはしない．

タスクの記述に関して，以下の2つのレベルがユーザに提供されている．

(a) 並列反復計算パターンのテンプレート
(b) タスククラスとタスクスケジューラ

TBB は，これらをテンプレート関数，テンプレートクラスなど，それぞれに適した形で提供している．代表的なものを**表 -1** に示す．

(a) は，「基本的な並列計算パターン」で述べたデータ並列スケルトンと一部のタスク並列スケルトンに相当し，並列の繰り返し，並列の畳み込み，パイプラインなどを提供する．記述するアプリケーションにも依存するが，一般的にはこれらのテンプレートを利用してプログラムを記述することが多いであろう．これらを利用する限りにおいては，繰り返しのどの部分を1つのタスクとするかなど，タスクに関する詳細（生成，同期など）を記述する必要はいっさいなく，並列性の隠蔽の度合は非常に高い．

ここで，代表的なテンプレートである `parallel_for` を，1次元配列 `a` の各要素に対して関数 `f` を適用した結果を別の1次元配列 `b` に格納する例を用いて紹介する．ここで関数 `f` は別途適当に定義されているものとする．この例における処理を逐次的に書けば，次のようになる．ここでは配列の大きさを `N` とした．

```
for ( int i = 0; i < N; i++ )
  b[i] = f(a[i]);
```

☆1 ここでの map は，C++ の標準テンプレートライブラリ（STL）における連想コンテナの map とはまったく異なり，リストの各要素に同じ関数を適用するという，Lisp における map 関数に相当するものであり，STL の標準アルゴリズムにおける `for_each()` に対応する．

☆2 http://www.threadingbuildingblocks.org/

(a) 並列反復計算パターン	
parallel_for	反復範囲の分かっているループの並列化を行うテンプレート関数. map スケルトンの一般化.
parallel_reduce	反復範囲の分かっているループにおける畳み込み計算の並列化を行うテンプレート関数. reduce スケルトンの一般化.
parallel_scan	parallel_reduce の途中結果を保持するテンプレート関数. scan スケルトンの一般化.
parallel_do	反復範囲が不明なループの並列化を行うテンプレート関数.
pipeline	パイプライン処理の並列化を提供する抽象クラス.
parallel_sort	並列ソートを行うテンプレート関数.
(b) タスククラスとタスクスケジューラ	
task	タスクを表現するクラス. execute, spawn などタスクを制御するメソッドを提供.

表 -1　TBB で提供する代表的な並列計算パターン

parallel_for は，繰り返しの範囲を示すオブジェクト（標準テンプレートライブラリのイテレータに相当）を併用する必要がある．この例の場合，繰り返しの対象は 1 次元なので，blocked_range というクラスのインスタンスを使う．後の「例題：N-Queens 問題」で述べる 2 次元の繰り返しの場合には，blocked_range2d クラスを使う必要がある．

繰り返しを行うループは，関数オブジェクトを用い，関数呼び出しを行う () 演算子をオーバーロードして記述しなければならない．() 演算子では，与えられた繰り返し範囲について，for 文を用いて順次 f を適用する．なお，配列の各要素に適用する関数 f は普通の関数でもよいが，関数オブジェクトとして定義する方が効率的であることが多い．

```
struct MapF {
  int *src, *dst;
  MapF(int a[], int b[]) : src(a), dst(b) { }
  void operator()(const blocked_range<int>&
                                    r) const {
    for ( int i=r.begin(); i<r.end(); i++ )
      dst[i] = f(src[i]);
  }
};
```

このように関数オブジェクトを用意すれば，次のようにして parallel_for を呼び出せばよい．

```
int a[N] = { 1, 2, ... }     // 配列の値の設定
int b[N];
parallel_for(blocked_range<int>(0,N),
             MapF(a,b),auto_partitioner());
```

ここで parallel_for の第 3 引数は，繰り返し本体をタスクに分割する際のタスクの大きさ（粒度）を指定する．粒度が小さすぎるとタスクが増えすぎてスケジューリングのオーバヘッドが大きくなるし，逆に粒度が大きすぎるとコアを有効に活用できなくなる．auto_partitioner() は，粒度を「適切に」自動的に決定するよう，ライブラリに依頼することを表す．プログラマが明示的に粒度を指定することもできるが，適切な粒度を定めるには，いろいろな値を与えて性能を測定するなどの試行錯誤が必要である．

(a) のようなテンプレートを利用するだけでは処理をうまく記述できない場合，典型的には，fork-join 型の並列処理を記述したい場合などのために，TBB では，(b) のタスクスケジューラを陽に直接操作することが可能となっている．たとえば，再帰アルゴリズムを用いて，再帰が浅い間は再帰処理タスクを生成して並列処理を行い，ある程度の深さになれば逐次的に処理するようなプログラムを記述することができる．この場合，プログラムには，タスクの生成，同期，場合によっては再利用，破棄，スレッドに割り当てるべき次のタスクの指定などの低レベルの処理を記述する必要が生じるので，並列性の隠蔽の度合は (a) ほど高くはない．しかし，(b) の機能を利用することにより，(a) だけを利用した場合と比較して，より広い範囲の問題に対して TBB を応用することができる．もちろん先述したように，ここで陽に記述するのはタスク，すなわち並列に実行すべき仕事であり，タスクで指定された仕事を実際に実行するスレッドの振舞いを記述するわけではないことに注意されたい．

タスクを陽に記述するには，次のように task クラスの子クラスを定義し，execute という仮想関数をオーバーライドして仕事の本体を記述しなければならない．

```
struct ATask: public task {
  ATask(...) ...              // ATask の構成子
  task *execute() {
    ここにタスクで実行すべき仕事を記述
  }
};
```

タスクを生成するには new 演算子を用い，実行を開始するには，spawn などを用いる必要がある．以下は，ATask で定義されているタスクのインスタンスを 2 つ生成し，実行を開始する例である．

```
ATask& a = *new(allocate_child()) ATask(...);
                              // タスクの生成
ATask& b = *new(allocate_child()) ATask(...);
                              // タスクの生成
spawn(a);                     // aを実行開始
spawn_and_wait_for_all(b);
                // bを実行開始，子タスクの終了を待つ
```

はじめに生成したタスクaに対してspawnにより実行開始（正確にはタスクを実行待ち状態にする）を指示し，2番目に生成したタスクbに対してはspawn_and_wait_for_allにより実行開始を指示すると同時に，自身は子タスク全員の終了を待つ．

ここでは詳しくは述べないが，TBBは，並列に作動するタスク間の相互作用を記述するための基本機構と並列データ構造(メモリ割り当て，排他制御，共有データ)も提供している．たとえば，C++の標準テンプレートライブラリのコンテナ（データの集合が抽象化されたもの，たとえばキューなど）は，並列に利用すると内部に矛盾を生じる可能性があるので，TBBで使うのは適当ではない．かわってTBBでは並列利用に耐え得る(「スレッドセーフ」という)コンテナ(並列キューなど)を提供している．

ここで，TBBの内部構成に関して簡単に触れておく．TBBでは，生成されたタスクを親子関係に基づく木構造で管理している．タスクスケジューラは，効率的と思われる方法で，この木構造中にあるタスクの実行をスケジューリングする．並列実行の主体である各スレッドには，自身に割り当てられた実行可能タスクを保持する「レディプール」と呼ばれるデータ構造が付随している．スレッドは，前に実行したタスクにより次のタスクが指定されていればそれを実行するし，そうでなければ，自身のレディプール中のタスクをある基準で選んで実行する．自身のレディプールにタスクがなければ，他スレッドのレディプールからタスクを「盗んで」実行する．この機構を「タスクスチール」と呼ぶ．

SkeTo

SkeTo[☆3]（Skeletons in Tokyo）は，東京大学，電気通信大学，国立情報学研究所の共同研究プロジェクトで開発中の，PCクラスタのような分散環境を対象とした並列スケルトンライブラリである．SkeToの現在公開されているバージョン1.0は，クラスタの各ノードがシングルコアであることを想定しているが，現在，各ノードがマルチコアCPUで構成されているような分散環境向けの新しいバージョンを開発中である．ここでは，現在開発中のマルチコア対応のSkeToを解説する．

SkeToは，次のような特徴を持つ．

- データ構造の再帰的な定義とその上の演算に関する構成的アルゴリズム論[5]と呼ばれる理論に基づいたデータ並列スケルトンを提供している．
- ノードへのデータの分散と収集，各ノードにおける並列計算などは，データの構成子や並列スケルトンの中に完全に隠蔽されている．したがって，ユーザは完全に逐次的な感覚でプログラムを記述することができる．
- リスト(分散1次元配列)，行列(分散2次元配列)，木(分散2分木)といった多様なデータ型を提供している．

TBBとの大きな違いは，SkeToは分散環境を前提としており，各ノードへのデータの分散，ノードをまたがった並列計算もSkeToの守備範囲という点である．

SkeToで提供する代表的なスケルトンを図-1に示す．図-1は，関数型言語風の記法を用いているが，実際にはC++のテンプレートを用いたライブラリとしてユーザに提供されている．

以下，図-1 (a) に示したリスト用スケルトンを例として，SkeToにおけるスケルトンの利用法を紹介する．ここでの例は，以下の逐次プログラムに対応するものである．

```
double as[100];
for ( int i = 0; i < 100; i++ ) as[i] = f(i)
                // 配列の初期値を関数fを用いて設定
double ave = compute_average(as);
                                  // 平均の計算
for ( int i = 0; i < 100; i++ )
   as[i] = as[i] - ave;
for ( int i = 0; i < 100; i++ )
   as[i] = as[i] * as[i];
```

SkeToでは，上の逐次プログラムの最初のfor文における配列値の設定については，リストの構成子を用いることで，2番目と3番目のfor文については，配列の各要素に関数を適用するmapスケルトンを用いることで並列化を行う．

SkeToでは，リストはdist_listテンプレートクラス（実体は要素が分散配置された1次元配列）で表現されている．リストを作成するには要素の型をパラメータとして与え，構成子に対して全要素数と(必要ならば)要素値初期化を行う関数オブジェクトを与える．下の例は，double型の100個の要素からなるリストを生成し，そのリストへのポインタを変数asに格納している．リストの要素値はGenクラスのインスタンスである関数オブジェクトで生成される．その内容は，リスト全体に

☆3 http://www.ipl.t.u-tokyo.ac.jp/sketo/

$$
\begin{aligned}
\mathsf{map}\,(f,[x_1,x_2,\ldots,x_n]) &= [f(x_1),f(x_2),\ldots,f(x_n)]\\
\mathsf{reduce}\,(\oplus,[x_1,x_2,\ldots,x_n]) &= x_1\oplus x_2\oplus\cdots\oplus x_n\\
\mathsf{scan}\,(\oplus,[x_1,x_2,\ldots,x_n]) &= [x_1,x_1\oplus x_2,x_1\oplus x_2\oplus\cdots\oplus x_n]
\end{aligned}
$$

(a) リスト用スケルトン

$$
\mathsf{map}\,(f,\begin{pmatrix} x_{11} & x_{12} & \ldots & x_{1n}\\ x_{21} & x_{22} & \ldots & x_{2n}\\ \vdots & \vdots & \ddots & \vdots\\ x_{m1} & x_{m2} & \ldots & x_{mn}\end{pmatrix}) = \begin{pmatrix} f(x_{11}) & f(x_{12}) & \ldots & f(x_{1n})\\ f(x_{21}) & f(x_{22}) & \ldots & f(x_{2n})\\ \vdots & \vdots & \ddots & \vdots\\ f(x_{m1}) & f(x_{m2}) & \ldots & f(x_{mn})\end{pmatrix}
$$

$$
\mathsf{reduce}\,(\oplus,\otimes,\begin{pmatrix} x_{11} & x_{12} & \ldots & x_{1n}\\ x_{21} & x_{22} & \ldots & x_{2n}\\ \vdots & \vdots & \ddots & \vdots\\ x_{m1} & x_{m2} & \ldots & x_{mn}\end{pmatrix}) = \begin{array}{l}(x_{11}\otimes x_{12}\otimes\cdots\otimes x_{1n})\oplus\\ (x_{21}\otimes x_{22}\otimes\cdots\otimes x_{2n})\oplus\\ \cdots\\ (x_{m1}\otimes x_{m2}\otimes\cdots\otimes x_{mn})\end{array}
$$

$$
\mathsf{scan}\,(\oplus,\otimes,\begin{pmatrix} x_{11} & x_{12} & \ldots & x_{1n}\\ x_{21} & x_{22} & \ldots & x_{2n}\\ \vdots & \vdots & \ddots & \vdots\\ x_{m1} & x_{m2} & \ldots & x_{mn}\end{pmatrix}) = \begin{pmatrix} y_{11} & y_{12} & \ldots & y_{1n}\\ y_{21} & y_{22} & \ldots & y_{2n}\\ \vdots & \vdots & \ddots & \vdots\\ y_{m1} & y_{m2} & \ldots & y_{mn}\end{pmatrix}\ \mathsf{where}\ y_{ij} = \begin{array}{l}(x_{11}\otimes\cdots\otimes x_{1j})\oplus\\ (x_{21}\otimes\cdots\otimes x_{2j})\oplus\\ \cdots\\ (x_{i1}\otimes\cdots\otimes x_{ij})\end{array}
$$

(b) 行列用スケルトン

$$
\mathsf{map}\,(f,\ \begin{array}{c}x_1\\ /\backslash\\ x_2\ \ x_3\\ /\backslash\\ x_4\ \ x_5\end{array}\ ) = \begin{array}{c}f(x_1)\\ /\backslash\\ f(x_2)\ \ f(x_3)\\ /\backslash\\ f(x_4)\ \ f(x_5)\end{array}
$$

$$
\mathsf{reduce}\,(\oplus,\otimes,\ \begin{array}{c}x_1\\ /\backslash\\ x_2\ \ x_3\\ /\backslash\\ x_4\ \ x_5\end{array}\ ) = x_1\oplus((x_2\oplus(x_4\otimes x_5))\otimes x_3)
$$

$$
\mathsf{uAcc}\,(\oplus,\otimes,\ \begin{array}{c}x_1\\ /\backslash\\ x_2\ \ x_3\\ /\backslash\\ x_4\ \ x_5\end{array}\ ) = \begin{array}{c}y_1\\ /\backslash\\ y_2\ \ y_3\\ /\backslash\\ y_4\ \ y_5\end{array}\ \mathsf{where}\ \begin{cases} y_1 = x_1\oplus((x_2\oplus(x_4\otimes x_5))\otimes x_3)\\ y_2 = x_2\oplus(x_4\otimes x_5)\\ y_3 = x_3\\ y_4 = x_4\\ y_5 = x_5\end{cases}
$$

$$
\mathsf{dAcc}\,(\odot,g_l,g_r,c,\ \begin{array}{c}x_1\\ /\backslash\\ x_2\ \ x_3\\ /\backslash\\ x_4\ \ x_5\end{array}\ ) = \begin{array}{c}z_1\\ /\backslash\\ z_2\ \ z_3\\ /\backslash\\ z_4\ \ z_5\end{array}\ \mathsf{where}\ \begin{cases} z_1 = c\\ z_2 = c\odot g_l(x_1)\\ z_3 = c\odot g_r(x_1)\\ z_4 = c\odot g_l(x_1)\odot g_l(x_2)\\ z_5 = c\odot g_l(x_1)\odot g_r(x_2)\end{cases}
$$

(c) 二分木用スケルトン

図 -1　SkeTo で提供する代表的なスケルトン

おける添字を引数にもらい，格納されるべき値を関数 f を用いて計算する．

```
struct Gen: public
  skeleton::unary_function< int, double > {
  double operator()(int i) const {
    return f(i);
  }
};
dist_list< double > *as
   = new dist_list< double >(Gen(), 100);
```

このように生成したリストの各要素に対し，map スケルトンを用いて関数を適用するには，適用する関数を関数オブジェクトとして用意する必要がある．

```
struct Sqr: public
  skeleton::unary_function< double,double > {

  double operator()(double x) const
    { return x * x; }
};
struct Sub: public
  skeleton::unary_function< double,double > {
```

```
  double val;
  Sub(double v) : val(v) { }
  double operator()(double x) const
    { return x - val; }
};

double ave = compute_average(as);
                        // 平均の計算
list_skeletons::map_ow(Sub(ave), as);
                        // ave を減じる関数を map
list_skeletons::map_ow(Sqr(), as);
                        // 2 乗する関数を map
```

Sqr は引数を単純に 2 乗する関数オブジェクトのためのクラスである．これは double 型の引数を 1 つとり，double 型の値を返す 1 引数関数なので，SkeTo システムで用意されている 1 引数関数を表現する unary_function クラスを継承するように定義する．また Sub は，引数からある値を減じた値を返す関数オブジェクトのためのクラスである．減じる値をクラス内のメンバ変数 val に格納し，1 引数関数においては引数と val の差を返している．このように関数オブジェクトを用意した上で，リスト用の map スケルトンを実現する SkeTo の関数の 1 つである list_skeletons::map_ow を呼ぶ．list_skeletons::map_ow は，引数で与えられた配列の各値を関数適用の結果の値で破壊的に上書きする関数である（map スケルトンを実現する関数には，ほかにも関数適用の結果を格納する配列を引数に受け取る list_skeletons::map などもある）．

このように，SkeTo のプログラムには，字面上では並列実行に関する記述が陽には現れない．実際，上のプログラムの意味を，「1 つのプロセッサ上に配置されている配列 as に対し，list_skeletons::map_ow を用い，配列の各要素に関数を順番に（逐次的に）適用する」と考えても何ら支障はない．実際の動作は，dist_list の構成子により配列の要素はノードに分散配置され，さらに，list_skeletons::map_ow により，配列の要素への関数適用が，各ノード上の CPU コアにより並列に行われる．

ここで，マルチコア対応の SkeTo の実現方法に関して簡単に触れておく．SkeTo は，処理すべき全体の仕事を，次の 2 段階を経てコアに分散させる．

1. (計算機ノードへの分散) 仕事の全体を管理するマスタは，全体の仕事を適切な大きさに分割し，ノードに分散させる．データの送受信には MPI（Message Passing Interface）を用いている．
2. (ノードでのコアへの分散) 各ノードにおいては，マスタより課せられた仕事をさらに細かい仕事に分割し，コアに分散させる．ノードにおいて 1 つの仕事が終われば，次の仕事をもらうべくマスタに問い合わせる．

このように，仕事全体を 2 段階に分割することにより，処理速度の遅いノードによる処理性能への影響を最小限にとどめるように負荷をうまく分散させると同時に，コアの有効活用をはかっている．

先に述べたように，マルチコア対応の SkeTo は現在開発中であるが，準備が整いしだい公開する予定である．

例題：N-Queens 問題

ここでは，TBB と SkeTo の共通の例題として，N-Queens 問題を取り上げる．はじめに逐次版のプログラムを図 -2 に示す．nqueen は，一辺 boardsize のチェス盤の第 col 列未満（列の範囲は 0 以上 boardsize 未満）の各列にクイーンが置かれた状態における解の個数を求める関数である．ここで，すでに配置されたクイーンの場所は配列 b で与えられる．nqueen は，第 col 列に新しいクイーンを置き nqueen を再帰的に呼ぶ，という単純なものである．

N-Queens 問題の並列解法として，本特集の 1 つ目の解説にあるように，0 列目と 1 列目にクイーンを配置（全部で N^2 通りの組合せ）し，そのそれぞれの組合せに対してその先を逐次的な計算によって解の個数を求める，という作業を並列に行う．最後にそれらを全部合計して解の総数を求める．

ここでは，TBB において上の解法を次の 2 通りの方法で記述する．

データ並列版　2 重の for ループで N^2 通りの計算を行いその結果を集計する作業を，parallel_reduce を用いて並列化する．

タスク並列版　タスクスケジューラの機能を利用し，それぞれの逐次計算を行うタスクを陽に発生して計算を行う．

それぞれの方法を記述したプログラムを，図 -3 と図 -4 に掲げる．与えられた盤面から逐次的に解の数を求める関数 nqueen は，図 -2 と同じなのでここでは省略した．

parallel_reduce を用いる場合（図 -3），Nqueens というクラスにおいて () 演算子をオーバーロードして計算を行う．その計算は 2 重の for ループとなるので，2 次元の範囲を表す blocked_range2d を用いる．さらに Nqueens クラスでは，タスクへの分割で用いる構成子と，タスクによる各部分範囲の計算結果を集計するための join というメソッドも適切に定義する必要があ

```
#include <iostream>
#define MAXN  20
int board[MAXN];                              // クイーンの位置の配列

int nqueen(int boardsize, int b[], int col)
{                                             // col 列目までは配置済
  int nsolutions, k;

  if ( boardsize == col ) return 1;           // col が盤面の大きさならば解
  nsolutions = 0;
  for ( int q = 0; q < boardsize; q++ ) {     // col 列目のクイーンを配置
    for ( k = 0; k < col; k++ )               // 配置済の各クイーンとの当たり判定
      if ( b[k] == q || b[k] - q == col - k || q - b[k] == col - k )
        break;                                // 当たっているので置けない
    if ( k == col ) {                         // 当たっていないので大丈夫
      b[col] = q;                             // 実際に col 列にクイーンを置き
      nsolutions += nqueen(boardsize, b, col + 1);  // 再帰
    }
  }
  return nsolutions;
}

int nqueen_seq(int boardsize)                 // 逐次版 N クイーンの関数
{
  return nqueen(boardsize, board, 0);
}

int main(void)
{
  int boardsize = 16;                         // 盤面の大きさ (16) の設定
  std::cout << nqueen_seq(boardsize) << std::endl;
  return 0;
}
```

図 -2　N-Queens 問題のプログラム（逐次版）

る．このように，TBB の parallel_reduce は，map スケルトンと reduce スケルトンを融合した一般的なものとなっている．

一方タスク並列版（図 -4）では，N-Queens 問題を計算するタスクを NqueensTask というクラスで定義している．そこて定義されている execute は，すでにクイーンを 2 個配置していれば（メンバ変数 col は次にクイーンを配置する列番号なので，これが 2 以上であればクイーンが 2 個あることが分かる），そこから先は逐次版の関数 queen を呼んで解の個数を求める．そうでなければ，クイーンの新たな配置（N 通り）それぞれに対して子タスクを生成し実行を開始する．ただし，タスクスケジューラにタスク情報を正しく管理させるために，set_ref_count で参照数を設定しなければならない．実行開始にあたっては，最後の配置以外は spawn を用いるが，最後だけは子タスクの計算結果（結果の格納場所は子タスク生成時に引数で与えている）を集計する必要があるので，spawn_and_wait_for_all を用いる．

最後に SkeTo 版のプログラムを図 -5 に示す．チェス盤の 0 列目と 1 列目にあらかじめクイーンを配置するので，図 -1 (b) に示した行列用のスケルトンを用いる．行列（実体は分散された 2 次元配列）は 2 つ用いる．1 つは入力として 2 列分のクイーンの配置が与えられた board で，その (i, j) 成分はペア $\langle i, j \rangle$（標準的に用意されている pair を利用）を要素として持つ．要素の初期化は，ペアを生成する関数オブジェクト GenPair を構成子の引数に与えることにより行っている．もう 1 つの行列は，それぞれの盤面における解の総数を保持する solutions である．こちらには計算結果が格納されるので，生成時に値を生成する必要はない．board を行列として用意した後は，行列用 map スケルトンを実現する C++ 関数 matrix_skeletons::map を用いて，解の数を求める関数オブジェクト Nqueens を各要素に適用し，最後にその結果を，行列用 reduce スケルトンを実現する C++ 関数 matrix_skeletons::reduce を用いて集計する．matrix_skeletons::reduce には，列方向の畳み込みと行方向の畳み込みを行う 2 つの関数オブジェクト（図 -1 (b) での $\oplus$ と $\otimes$ に相当）を引数に与える必要がある点に注意されたい．ここでは，両方とも単に和を求める Add クラスの関数オブジェクトを与えている．

SkeTo のプログラムでは，実行本体を SketoMain という関数に記述する．main は SkeTo で提供するライブラリの中で定義されており，SkeTo のライブラリをリンクすると自動的に付加される．この main は，MPI 等の必要な初期設定を行った後に，ユーザが定義した SketoMain を呼ぶようになっている．

これらのプログラムを単一計算機ノードで実行した結果を，表 -2 に示す．実験環境は，CPU は QuadCore Xeon E5405 2.5GHz × 2（合計 8 コア），メモリは 4GB,

```
#include <tbb/task_scheduler_init.h>
#include <tbb/blocked_range2d.h>
#include <tbb/parallel_reduce.h>
#include <iostream>
using namespace tbb;
// ここに図-2の MAXN と nqueen の定義を置く（省略）

struct Nqueens {                        // parallel_reduceのための関数オブジェクトのクラス
  const int boardsize;                  // 盤面の大きさ
  int nsolutions;                       // 解の数

  void operator()(const blocked_range2d<int>& r) {
    for ( int q0 = r.rows().begin(); q0 < r.rows().end(); q0++ )
      for ( int q1 = r.cols().begin(); q1 < r.cols().end(); q1++ )
        if ( q0 != q1 && q0 != q1 + 1 && q0 != q1 - 1 ) {
          int board[MAXN];
          board[0] = q0; board[1] = q1; // 0, 1列目にクイーン配置
          nsolutions += nqueen(boardsize, board, 2); // 逐次版利用
        }
  }
  void join(const Nqueens& y) {         // 解の集計の方法の定義
    nsolutions += y.nsolutions;         // 自分の解数とyの解数を足せばよい
  }
  Nqueens(Nqueens& x, split) : boardsize(x.boardsize), nsolutions(0) { }
  Nqueens(int n) : boardsize(n), nsolutions(0) { }
};

int nqueen_tbb_parallel_reduce(int boardsize, int ncore)
{                                       // paralle_reduceを利用する関数，ncoreはコア数
  task_scheduler_init init(ncore);      // 必ず必要．コア数を教える
  Nqueens nq(boardsize);                // Nqueensのインスタンス作成
  parallel_reduce(blocked_range2d<int>(0, boardsize, 0, boardsize),
                  nq, auto_partitioner());
  return nq.nsolutions;                 // nq.solutionsに答がある
}

int main(void)
{
  int boardsize = 16, ncore = 4;        // 盤面の大きさ(16)とコア数(4)の設定
  std::cout << nqueen_tbb_parallel_reduce(boardsize, ncore) << std::endl;
  return 0;
}
```

図-3　N-Queens問題のプログラム（TBB，データ並列版）

OSはLinux Fedra 7である．Cコンパイラは GCC 4.3.0で-O2オプションを与えた．また，SkeToで利用しているMPIはMPICH2-1.07であるが，今回の実験は単一ノードで行っているので，MPIは実行速度にはほとんど影響を与えない．

この結果を見ると，TBBの両版，SkeToともに実行性能に大きな差はなく，ほどよく台数効果が出ていることが分かるであろう．N-Queens問題は並列化と相性が良いため並列効果が大きかったが，効果の度合は当然のことながら問題の性質やプログラムの書き方に依存する．したがって，TBBやSkeToを利用するにあたっては，プログラムの書きやすさと並列効果とのトレードオフを見極める必要がある．

今後の展望

前章の例題から分かるように，TBBもSkeToも，C++の次のような機能に大幅に依存している．

- テンプレート
- 関数オブジェクトと演算子のオーバーロード

ユーザは上のC++の機能を駆使してプログラムを作成する必要があり，C言語の延長としてC++を用いているような平均的なC++プログラマにとっては，敷居が若干高いという点は否めないであろう．

C++の次の標準化(C++200X)において導入されるであろうラムダ式が利用できるようになれば，関数オブジェクトの記述に関してユーザの負担が大幅に軽減することが期待される．たとえば，「double型の引数を2乗して返す関数オブジェクト」は，ラムダ式を用いれば

```
[&](double x) { return x * x; }
```

と記述することができる．これを利用して，たとえばSkeToの場合，関数オブジェクトのためのクラスを陽に記述せずに

```
list_skeletons::map_ow
  ([&](double x) { return x * x; }, as);
```

と書くことができれば，プログラムの記述のしやすさは

```
#include <tbb/task_scheduler_init.h>
#include <tbb/task.h>
#include <iostream>
using namespace tbb;
// ここに図 -2 の MAXN と nqueen の定義を置く （省略）

struct NqueensTask: public task {
  const int boardsize, col;                          // 盤面の大きさと有効な列数
  int qpos0, qpos1;                                  // 0 列目と 1 列目のクイーンの位置
  int *psolutions;                                   // 結果を格納する場所へのポインタ

  NqueensTask(int bsize, int c, int q0, int q1, int *p) :
    boardsize(bsize), col(c), psolutions(p) {
      if ( c == 1 ) {                                // 列数が 1 の場合
        qpos0 = q1; qpos1 = -1;
      } else {                                       // 列数が 2 の場合
        qpos0 = q0; qpos1 = q1;
      }
  }

  task *execute() {                                  // タスクでの仕事本体
    if ( col >= 2 ) {                                // 2 列分のデータは配置済
      if ( qpos0 != qpos1 && qpos0 != qpos1 + 1 && qpos0 != qpos1 - 1 ) {
        int board[MAXN];                             // 0, 1 列目のクイーンの位置を確認後
        board[0] = qpos0; board[1] = qpos1;
        *psolutions = nqueen(boardsize, board, 2); // 逐次実行
      } else                                         // 0, 1 列目のクイーンが当たっている
        *psolutions = 0;
    } else {                                         // col==0 あるいは col==1
      int solutions[MAXN];
      set_ref_count(boardsize + 1);                  // 参照数を設定
      for ( int q = 0; q < boardsize - 1; q++ ) {    // 次の列の各位置に関して
        NqueensTask& a = *new(allocate_child())      // タスクを生成
            NqueensTask(boardsize, col + 1, qpos0, q, &solutions[q]);
        spawn(a);                                    // 実行開始
      }
      NqueensTask& a = *new(allocate_child())        // 最後の位置だけは
          NqueensTask(boardsize, col + 1, qpos0, boardsize - 1, &solutions[boardsize - 1]);
      spawn_and_wait_for_all(a);                     // 実行を開始し子タスクの終了を待つ
      int r = 0;
      for ( int q = 0; q < boardsize; q++ ) r += solutions[q]; // 結果集計
      *psolutions = r;
    }
    return NULL;     // NULL は次に実行すべきタスクを指定しないという意味
  }
};

int nqueen_tbb_task(int boardsize, int ncore)
{
  task_scheduler_init init(ncore);                   // コア数
  int nsolutions;
  NqueensTask& a = *new(task::allocate_root())       // おおもとのタスク
                   NqueensTask(boardsize, 0, -1, -1, &nsolutions);
  task::spawn_root_and_wait(a);                      // 実行開始
  return nsolutions;
}

// main は nqueen_tbb_parallel_reduce を nqueen_tbb_task に変える以外は
// 図 -2 と同じ （省略）
```

図 -4　N-Queens 問題のプログラム（TBB，タスク並列版）

改善され，平均的なユーザに対する敷居も低くなるであろう．

これらの記述性の改善，さらには，提供する並列計算パターンのさらなる改良などを通し，ライブラリの機能や性能が進化し，システム開発者のみならず一般のユーザにも並列計算パターンによる並列プログラミングが広く浸透し利用されるようになることを期待したい．

参考文献

1）胡　振江，岩崎英哉：スケルトン並列プログラミング，情報処理，Vol.46, No.10, pp.1158-1162 (Oct. 2005).
2）Reinders, J：Intel Threading Building Blocks, O'REILLY (2008).（邦訳）菅原清文監訳：インテルスレッディング・ビルディング・ブロック，オライリー・ジャパン (2008).
3）Robison, R.：General Parallel Algorithms in Threading Building Blocks, Proc. 21st International Workshop on Languages and Compilers for Parallel Computing (LCPC 2008) (2008).
4）Matsuzaki, K., Emoto, K., Iwasaki, H. and Hu, Z.：A Library of Constructive Skeletons for Sequential Style of Parallel Programming, Proc. 1st International Conference on Scalable Information Systems (InfoScale 2006) (2006).
5）岩崎英哉：構成的アルゴリズム論，コンピュータソフトウェア，Vol.15, No.6, pp.57-70 (1998).

（平成 20 年 10 月 10 日受付）

```
#include <mpi.h>
#include <stdlib.h>
#include "matrix_skeletons.h"
#include "dist_matrix.h"
#include "primitive_functions.h"
using namespace primitive_functions;
// ここに図-2の MAXN と nqueen の定義を置く（省略）

struct Nqueens: public skeleton::unary_function< std::pair<int,int>, int > {
  int boardsize;
  void operator()(const std::pair<int,int> p, int *nsolutions) const {
    int q0 = p.first;                                  // ペアの第1成分は0列目のクイーンの位置
    int q1 = p.second;                                 // ペアの第2成分は1列目のクイーンの位置
    if ( q0 != q1 && q0 != q1 + 1 && q0 != q1 - 1 ) {
      int board[MAXN];                                 // 0，1列目のクイーンの位置を確認後
      board[0] = q0; board[1] = q1;
      *nsolutions = nqueen(boardsize, board, 2);       // 逐次実行
    } else                                             // 0，1列目のクイーンが当たっている
      *nsolutions = 0;
  }
  Nqueens(int bsize) : boardsize(bsize) { }
};

struct Add: public skeleton::binary_function< int,int,int > {
  void operator()(int a, int b, int *c) const { *c = a + b; } // 加算
};

struct GenPair: public skeleton::binary_function< int,int,std::pair<int,int> > {
  void operator()(int i, int j, std::pair<int,int> *p) {
    p->first = i; p->second = j;                       // ペアにiとjを設定
  }
};

int nqueen_sketo(int boardsize, int ncore)
{
  int nsolutions;
  dist_matrix< std::pair<int,int> > board(GenPair(), boardsize, boardsize);
  dist_matrix<int> solutions(boardsize, boardsize); // 解数を保持
  matrix_skeletons::map(Nqueens(boardsize), &board, &solutions);
  matrix_skeletons::reduce(Add(), Add(), &solutions, &nsolutions);
  return nsolutions;
}

int SketoMain(int argc, char **argv)
{
  int boardsize = 16, ncore = 4;                       // 盤面の大きさ(16)とコア数(4)の設定
  skeleton::set_cores(ncore);
  skeleton::cout << nqueen_sketo(boardsize, ncore) << std::endl;
  return 0;
}
```

図-5　N-Queens 問題のプログラム（SkeTo 版）

	コア数			
	1	2	4	8
逐次プログラム	436.9	−	−	−
TBB データ並列版	−	216.3	120.2	62.14
TBB タスク並列版	−	219.8	110.5	55.48
SkeTo	−	220.8	112.5	58.34

表-2　16 クイーンの実行速度の比較（単位：秒）

岩崎　英哉（正会員） iwasaki@cs.uec.ac.jp

1960 年生．1983 年東京大学工学部計数工学科卒業．1988 年同大学院工学系研究科情報工学専攻博士課程修了．同年同大計数工学科助手．1993 年同大教育用計算機センター助教授．その後，東京農工大学工学部電子情報工学科助教授，東京大学大学院工学系研究科情報工学専攻助教授，電気通信大学情報工学科助教授を経て，2004 年より電気通信大学情報工学科教授．工学博士．プログラミング言語，システムソフトウェア等の研究に従事．日本ソフトウェア科学会，ACM 各会員．

胡　　振江（正会員） hu@nii.ac.jp

1966 年生．1988 年中国上海交通大学計算機科学系を卒業．1996 年東京大学大学院工学系研究科情報工学専攻博士課程修了．同年日本学術振興会特別研究員を経て，1997 年東京大学大学院工学系研究科情報工学専攻助手，同年 10 月同専攻講師，2000 年同専攻准教授，2008 年より国立情報学研究所教授．博士（工学）．プログラミング言語，関数プログラミング，ソフトウエア工学，並列プログラミングなどに興味を持つ．日本ソフトウェア科学会，ACM 各会員．