特集
マルチコアを活かすお手軽並列プログラミング

② GCC 上での並列プログラミングサポート

江本 健斗（東京大学）

GCC の並列プログラミングサポート

GCC（GNU Compiler Collection）[1] は，GNU プロジェクトにより開発されているフリーのコンパイラ群であり，C，C++，Objective-C，Fortran，Java，Ada のコンパイラと標準ライブラリを含む．近年，マルチコア CPU の普及により並列計算機環境が身近になってきたことを受け，容易な並列プログラミングのためのサポートが GCC に組み込まれてきている．

現在の GCC には，容易な並列プログラミングのためのサポートとして，OpenMP と libstdc++ parallel mode の 2 つが用意されている．前者は，並列プログラミングのための標準化されたインタフェースであり，簡単に言えば，並列化してほしい部分を簡単な注釈でコンパイラに伝えることで半自動で並列化してもらう枠組みである．一方，後者は，C++ の標準ライブラリである libstdc++ の一部の機能に並列実装を提供するものであり，標準ライブラリを使って書かれた既存のプログラムを修正なしに並列化できる．OpenMP は GCC のバージョン 4.2.0 から C，C++，Fortran で利用可能であり，libstdc++ parallel mode は GCC のバージョン 4.3.0 から実験的に利用可能となっている．

本稿は，対象言語を C++ に限定し，上に挙げた GCC の提供する容易な並列プログラミングのためのサポートを紹介する．以降では，1 つの逐次プログラムを例題にし，前半では OpenMP を用いて並列化する方法を，後半では libstdc++ parallel mode を用いて並列化する方法を示す．

❐ 並列化の例題とする逐次プログラム

本稿では，N-Queens 問題を解くプログラムを並列化の例題として用いる．図 -1（nqueens.cpp）および図 -2（querying.h）に具体的な C++ のコードを示す．プログラムの流れは以下のとおりである．まず，メイン関数内で盤面のサイズを n に代入し（44 行目），解の数を数える関数 nqueens を呼び出す（45 行目）．関数 nqueens では，まず，queen を少しだけ配置した盤面を生成して配列 xs に格納する（40 行目）．そして，その初期盤面の配列 xs と，与えられた盤面から始めて queen を置ききることのできる盤面の数を返す関数 nqueen_rest とを引数として，次に説明する解探索のメインループである関数 querying を呼び出す．その他の部分は並列化にあたって特に理解する必要はない．

図 -2 の関数 querying は，E 型の要素の配列（ベクタ）xs と関数 f とを入力とし（1 ～ 2 行目），2 つのループで以下の計算を行う．まず，1 つ目のループ（6 ～ 8 行目）は，xs のすべての要素に f を適用して，その結果を別の配列 cs に格納する．次に，2 つ目のループ（9 ～ 12 行目）は，1 つ目のループで計算された cs の和を求める．そして，この 2 つ目のループで計算された和がプログラムの出力になる．N-Queens 問題を解くプログラムでは，xs は初期盤面の配列であり，f は盤面から解の数を求める関数 nqueen_rest である．1 つ目のループにより，各初期盤面からの解の数が cs に求められ，2 つ目のループにより，各初期盤面からの解の和，すなわち N-Queens 問題の解が求められる．

配列 xs のサイズが大きい場合や関数 f の処理が重たい場合には，図 -2 の関数 querying の 2 つのループをそれぞれ並列化することにより，計算時間の短縮が期待できる．以降の並列化は関数 querying のみに注目し，N-Queens 問題を解くプログラムの他の部分（図 -1）は並列化の前後で共通となる．

図 -2 の関数 querying は，N-Queens 問題を解く以外にもいくつかの応用が考えられる．たとえば，ある文章を構成する文の集合を xs とし，関数 f を文中の特定

```
 1 #include<iostream>
 2 #include<vector>
 3 #include<algorithm>
 4 #include<numeric>
 5 #include"querying.h"
 6 int nqueen(int n,int i,int u,int l,int r) {
 7   if(i==n) return 1;
 8   const int nl = (l << 1) | 1;
 9   const int nr = ((r|(1<<n)) >> 1);
10   const int ni = i + 1;
11   int p = u & nl & nr, c = 0;
12   while(p!=0) {
13     const int lb = (-p)&p; p ^= lb;
14     c += nqueen(n,ni,u^lb,nl^lb,nr^lb);
15   }
16   return c;
17 }
18 struct state {
19     int n, i, u, l, r;
20     state(int n, int i, int u, int l, int r) :
           n(n), i(i), u(u), l(l), r(r) {}
21 };
22 int nqueen_rest(const state &st) { return
       nqueen(st.n, st.i, st.u, st.l, st.r); }
23 void gentasks(std::vector<state> &xs, int lim,
       int n, int i, int u, int l, int r) {
24   if(i==lim) {
25     xs.push_back(state(n, i, u, l, r));
26     return;
27   }
28   const int nl = (l << 1) | 1;
29   const int nr = ((r|(1<<n)) >> 1);
30   const int ni = i + 1;
31   int p = u & nl & nr, c = 0;
32   while(p!=0) {
33     const int lb = (-p)&p; p ^= lb;
34     gentasks(xs,lim,n,ni,u^lb,nl^lb,nr^lb);
35   }
36 }
37 int nqueens(int n) {
38   int b = (1<<n) - 1;
39   std::vector<state> xs;
40   gentasks(xs, 3, n, 0, b, b, b);
41   return querying(xs, nqueen_rest);
42 }
43 int main(int argc, char *argv[]) {
44   int n = argc > 1 ? atoi(argv[1]) : 16;
45   int c = nqueens(n);
46   std::cout << n << "␣" << c << std::endl;
47   return 0;
48 }
```

図-1　N-Queens 問題を解くプログラム（nqueens.cpp）．解の探索のメインループとして図-2 の querying.h に定義される関数 querying を用いる(41 行目)．

の単語の数を返す関数とすれば，文章中の単語の出現数が求められる．また，関数 f を特定の文との類似度を求める関数にすれば，xs で表現される文章と特定の文との関連度のようなものが求めるられる．一般に，xs をデータベースとしたクエリ処理は，2 つ目のループの加算を変更する可能性はあるけれど，おおむね図-2 のコードで処理可能であろう．

図-2 のメインループを並列化する際には注意が 1 つある．それは，関数 f が副作用を持ってはならないことである．副作用とは，計算機の状態に変化を与えて以降の計算に影響を及ぼすことであり，たとえば画面への出力やグローバル変数への代入などが副作用である．図-2 の 6 ～ 8 行目のループは，並列化されなければ配列 xs の先頭から順に関数 f を適用する．しかし，並列化された場合には関数 f の適用順序は一般には不定となる．そのため，関数 f に副作用がある場合，ループの計算結果も不定となる．たとえば，関数 f が要素を

```
 1 template<typename E, typename F>
 2 int querying(std::vector<E> &xs, F &f)
 3 {
 4   int n = xs.size();
 5   std::vector<int> cs(n);
 6   for(int i = 0; i < n; i++) {
 7     cs[i] = f(xs[i]);
 8   }
 9   int sum = 0;
10   for(int i = 0; i < n; i++) {
11     sum = sum + cs[i];
12   }
13   return sum;
14 }
```

図-2　解の探索をするメインループの逐次プログラムコード（querying.h）

画面に出力するコードを含んでいた場合，その出力は先頭用素から順番に並んでいるとは限らない．

OpenMP を用いた並列化

本章では，GCC の並列プログラミングサポートの 1 つである OpenMP を，図-2 の逐次プログラムの並列化を通して紹介する．

❒ OpenMP

OpenMP[2] は，並列プログラミングのための標準化されたインタフェースであり，コンパイラに並列化の指示を伝えるためのディレクティブと実行時環境を操作する関数とからなる．現在，OpenMP Architecture Review Board が OpenMP の仕様のバージョン 3.0 を定めており，C, C++, Fortran に関しての標準化がなされている．GCC は，独立していた GOMP プロジェクトの成果を統合し，バージョン 4.2.0 より OpenMP を正式にサポートしている．GCC の OpenMP は，バージョン 3.0 の仕様を実装しており，標準化のなされている C，C++，Fortran のすべてで利用できる．

OpenMP を用いた並列化では，プログラマは逐次プログラムの並列化してほしい部分を簡単な注釈でコンパイラに伝え，コンパイラが与えられた注釈に従って半自動でプログラムを並列化する．この際の注釈に使用されるのがディレクティブである．ディレクティブには，大きくわけて，並列処理の開始を指示するものと，並列処理すべき仕事の生成を指示するものと，並列処理の同期を指示するものとがある．たとえば，ディレクティブ parallel は並列計算の開始を指示し，それ以降の仕事生成ディレクティブに従って生成された仕事が並列処理されるようになる．仕事生成のディレクティブの例と

```
1 template<typename E, typename F>
2 int querying(std::vector<E> &xs, F &f)
3 {
4   int n = xs.size();
5   std::vector<int> cs(n);
6 #pragma omp parallel for
7   for(int i = 0; i < n; i++) {
8     cs[i] = f(xs[i]);
9   }
10  int sum = 0;
11 #pragma omp parallel for reduction(+:sum)
12  for(int i = 0; i < n; i++) {
13    sum = sum + cs[i];
14  }
15  return sum;
16 }
```

図 -3　ディレクティブを追加した並列プログラムコード（querying.h）

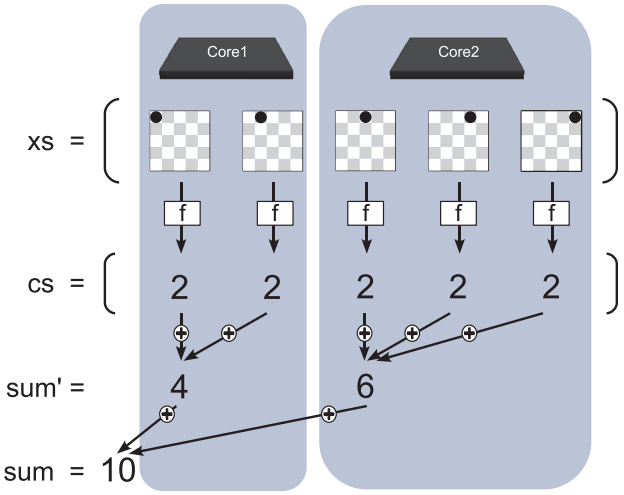

図 -4　OpenMP プログラムの並列計算イメージ

しては，ループ反復の分割による仕事の生成を指示するディレクティブ for があり，このディレクティブの直後にあるループの並列化がなされる．本稿では，容易に利用できるループの並列化に関する基本的なディレクティブのみを以下で紹介する．

❒ ディレクティブの追加による並列化

図 -2 の逐次プログラムを OpenMP ディレクティブを書き足すことで並列化し，基本的なディレクティブの使い方を説明する．**図 -3** のプログラムがディレクティブ（6 行目と 11 行目）を書き足した後の並列プログラムである．コンパイラへの指示であるディレクティブは，C++ ではプラグマ（#pragma）で記述され，OpenMP に関するプラグマを表す omp の後ろにディレクティブ名とオプションの条項が続く．

最初のループに付加したディレクティブ parallel for（6 行目）は，直後にある for ループの並列化を指示する．このディレクティブは，よく使われるディレクティブである parallel と for を組み合わせたショートカットである．ただし，並列化されるループは for(i= *s*; i *rel e*; i+= *d*) の形を持ち以下の条件を満たす必要がある．まず，ループの反復に関する式 *s*，*d*，*e* はすべてループ不変でなければならない．そして，条件式中の *rel* は，<，>，<=，>= の不等号のいずれかでなければならない．さらに，ループのインデックス i の型は，整数，ポインタ，ランダムイテレータのいずれかでなければならず，また，ループの本体で i を更新してはならない．これらの条件が課されるのは，ループの反復を分割して並列に処理するために，反復の回数をループ開始前に知る必要があるからである．さらに，反復の並列処理によってループ本体の実行順序が不定になるため，ループ本体には副作用があってはならない．

ループ本体に，ループ外で定義される変数 x を適当な式 *a* で更新する x = x *op a* や x *op*= *a* のような計算（簡単な副作用）がある場合は，上記の parallel for の後に reduction(*op*:x) を追加して並列化を指示する．このような変数の更新式は和や積を求めるループに現れる．たとえば，2 つ目のループの本体（13 行目）では sum = sum + cs[i] によって和を求めているので，11 行目ディレクティブには reduction(+:sum) が追加されている．ディレクティブ reduction(*op*:x) に使用できる演算子 *op* は，+，-，*，&，|，^，&&，|| であり，演算子のオーバロードはしてはならない．また，変数 x の型は int や double のような基本型でなければならず，その更新に使われる式 *a* には変数 x 自身を含んではいけない．

注釈を追加した並列プログラムによる N-Queens 問題の計算のイメージを**図 -4** に示した．入力 xs として 5×5 の盤面の 1 行目に queen を置いた状態を考え，2 つのコアが使える計算機での計算を考える．各々の盤面に対して解の盤面数を求める最初のループ計算は，盤面を 2 つと 3 つに分けて別々のコアで同時に計算することで，自然に並列実行される．さらに，2 つ目のループでの和の計算は，各々のコアで小計（sum'）を求めて後にそれらを合計することで，自然に並列に計算される．実際にはコア間の仕事量を均一にするためにもう少し複雑な仕事の割り振りが行われるかもしれないが，最も単純な並列計算は以上のとおりである．

❒ OpenMP プログラムのコンパイルと実行

OpenMP のディレクティブを追加されたプログラムは，コンパイルオプション -fopenmp をつけることで

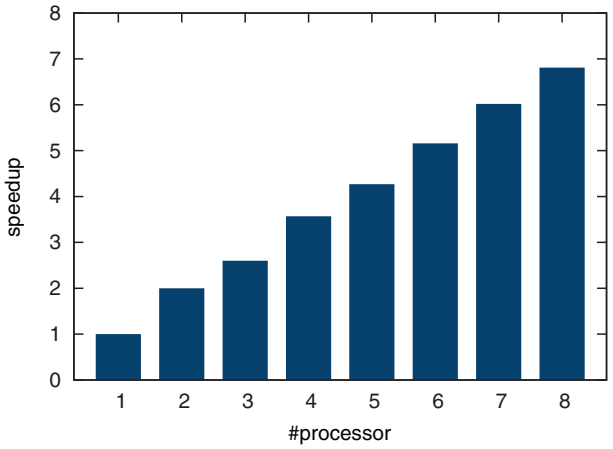

図 -5　OpenMP プログラムの速度向上

並列実行可能なバイナリにコンパイルされる．たとえば，ソースを nqueens.cpp として，次のコマンドで並列プログラムのバイナリ nqueens が得られる．

```
g++ -o nqueens nqueens.cpp -O3 -fopenmp
```

得られたバイナリは，特にオプションなどをつけることなく単純に実行することで，使用可能なプロセッサ・コアをすべて利用した並列計算を行う．これは，デフォルトで，使用可能なプロセッサ・コア数と同じ数のスレッドを生成して並列計算を行うからである．生成するスレッド数(すなわち使用するプロセッサ・コア数)を制限したい場合には，環境変数 OMP_NUM_THREADS に生成するスレッド数を指定する．たとえば，スレッド数を 4 にして実行するには，Linux などの bash を使用している場合

```
OMP_NUM_THREADS=4 ./nqueens
```

のように環境変数を指定して実行する．生成するスレッド数をプロセッサ・コア数よりも多くしても計算時間の短縮にはならないことに注意する．

OpenMP で並列化された図 -3 のプログラムで N-Queens 問題を解き，使用するプロセッサ・コア数を変化させたときの実行速度の変化を測定した．**図 -5** に測定結果を示す．横軸が計算に使用したプロセッサ・コアの数（1 ～ 8 台）であり，縦軸はもとの逐次プログラムに対する速度向上（すなわち，逐次プログラムの計算時間を並列プログラムの実行時間で割ったもの)である．理想は p 台のプロセッサ・コアを使用したときに p 倍の速度向上を得ることである．図 -5 より，8 台のプロセッサ・コアで 7 倍程度となる良好な速度向上が得られていることが分かる．理想からの多少のずれは，プロセッサ・コア間で計算の重さが多少不均一になることと，並列化のためのオーバヘッドがあることによる．

本稿の OpenMP に関する紹介は以上である．OpenMP を使用することで，注釈の追加という簡単な修正のみで逐次プログラムを並列化でき，並列化による十分な速度向上を簡単に得ることができた．

libstdc++ Parallel Mode を用いた並列プログラミング

本章では，GCC のもう 1 つの並列プログラミングサポートである libstdc＋＋ parallel mode を，図 -2 の逐次プログラムの並列化を通して紹介する．

❒ libstdc++ と Parallel Mode

libstdc＋＋ は GCC に含まれる C＋＋ の標準ライブラリ[3]である．標準ライブラリのうち，テンプレートを用いて抽象化された汎用的なコンテナ(データ構造)とコンテナを操作する汎用的なアルゴリズム（コンテナ上の汎用サブルーチン．以降，本章ではアルゴリズムという言葉をこの意味のみに用いる)の集まりは，STL（Standard Template Library）と呼ばれている．STL の提供するコンテナには，例題のプログラムで配列の代わりとして使用しているベクタ(std::vector)のほか，リスト(std::list)，集合（std::set）などがある．STL の提供するアルゴリズムには，与えられたデータ構造のすべての要素に関数を適用するもの，和や積のように二項演算子での合計を求めるもの，ソートを行うものなどがある．

紹介する libstdc＋＋ parallel mode（以下，parallel mode と略す）は，STL で提供される各種アルゴリズムに並列実装を与えるものである．そのため，STL を使って書かれた逐次プログラムが，コードの変更なしに再コンパイルのみで並列プログラムになる．この parallel mode は，もとは MCSTL（Multi-Core STL）[4] として GCC の外部で開発されていたものであるが，現在は GCC に統合されつつあり GCC バージョン 4.3.0 から実験的に利用可能となっている．

❒ Parallel Mode を用いた並列化

Parallel mode を用いた並列化は簡単である．STL のアルゴリズムを用いてプログラムを書き直し，特定のオプションをつけてコンパイルするだけである．以下，図 -2 の逐次プログラムを STL のアルゴリズムを使って書き直す．

STL で書き直したプログラムを**図 -6** に示す．STL を使い慣れているプログラマには自明な書き換えであろう．まず，逐次プログラムの 1 つ目のループ(図 -2 の 6 ～ 8 行目）は，ヘッダファイル algorithm に定義されるアルゴリズム std::transform に書き換えられる(図 -6 の 5 行目)．このアルゴリズムは，与えられた配

```
1 template<typename E, typename F>
2 int querying(std::vector<E> &xs, F &f)
3 {
4     std::vector<int> cs(xs.size());
5     std::transform( xs.begin(), xs.end(),
          cs.begin(), f);
6     int sum = std::accumulate( cs.begin(),
          cs.end(), 0, std::plus<int>());
7     return sum;
8 }
```

図 -6　STL を用いて書き直したプログラムコード（querying.h）

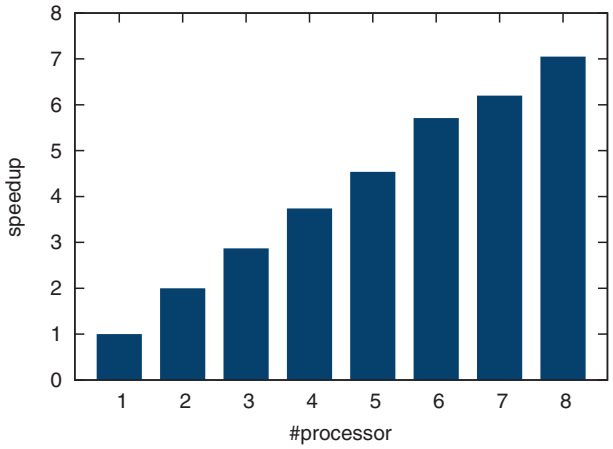

図 -7　Parallel Mode プログラムの速度向上

列のすべての要素（イテレータで与えられる反復範囲にある要素）に与えられた関数を適用して，別の配列（イテレータで示される範囲）に結果を代入する．

逐次プログラムの 2 つ目のループ（図 -2 の 9 ～ 12 行目）は，ヘッダファイル numeric に定義されているアルゴリズム std::accumulate に書き換えられる（図 -6 の 6 行目）．このアルゴリズムは，与えられた配列のすべての要素（イテレータで与えられる反復範囲にある要素）の合計を与えられた演算子を使用して求める．図 -6 のプログラムでは，和を求めるため，通常の加算演算子 plus<int>() が引数に渡されている．

ここで，parallel mode による並列化のために上記のアルゴリズムを使用する際の注意点を述べておく．まず，アルゴリズム std::transform の並列実装では，OpenMP で並列化されたループと同様に，関数の要素への適用の順番が一般に不定となる．したがって，std::transform によって要素へ適用される関数には副作用が含まれてはならない．そして，アルゴリズム std::accumulate の並列実装では，合計を求める際の演算子の適用順序が一般に不定となる．そのため，使用する演算子は結合則を満たさなければならない．すなわち，演算子を $\oplus$ として，$a\oplus(b\oplus c) = (a\oplus b)\oplus c$ が任意の a，b，c に対して成立する必要がある．たとえば，和を求めるために使う通常の加算演算子 +（プログラム中は plus<int>()）は結合則を満たすため，並列化される std::accumulate に使用しても問題ない．他の結合的な演算子の例としては，文字列の連結，集合の和と積，行列の和と積などがある．逆に，結合的でない，すなわち並列化される std::accumulate に使用できない演算子の例としては，除算や減算がある．

❐ STL プログラムのコンパイルと実行

STL を用いたプログラムから parallel mode の並列実装を有効にしたバイナリを得るためには，オプション -D_GLIBCXX_PARALLEL をつけてプログラムをコンパイルする．また，parallel mode の並列実装は OpenMP と各 CPU の持つアトミック操作とを使用しているため，OpenMP を有効にするオプションと CPU の種別を特定するオプションも同時に指定する．たとえば，ソースを nqueens.cpp として，次のコマンドで並列実装が有効になったバイナリ nqueens が生成される．

```
g++ -o nqueens nqueens.cpp -O3 -fopenmp \
      -march=native -D_GLIBCXX_PARALLEL
```

コンパイルされたバイナリは，OpenMP で並列化したプログラムと同様に，普通に実行するだけで利用できるプロセッサ・コアすべてを使用した並列計算を行う．また，使用するスレッド数も環境変数 OMP_NUM_THREADS で制限可能である．

Parallel mode で並列化された図 -6 のプログラムで N-Queens 問題を解き，使用するプロセッサ・コア数を変化させたときの実行速度の変化を測定した．**図 -7** に測定結果を示す．図の見方は図 -5 と同様である．図より，STL でプログラムを書くだけで，OpenMP と同様に良好な速度向上が得られることが分かる．

最後に，2008 年 11 月現在で利用できる GCC（4.3.2 および svn 上のソース）のバグについて述べておく．Parallel mode では，STL のアルゴリズムのすべての使用に対して並列実装が用いられるわけではなく，アルゴリズムの各々の使用に対してその引数の性質に基づく実装の選択が行われる．この選択は，並列実装を用いるか逐次実装を用いるかの選択である．しかし，現在の GCC の実装選択は，いくつかのアルゴリズムに対して，並列実装を選択すべき場合でも逐次実装を選択してしまうというバグを持つ．このため，バグの影響するアルゴリズムに対してデフォルトでは逐次実装しか使用されないという状況が生じている．このバグの報告はすでにな

	OpenMP	libstdc++ parallel mode
ループの並列化のために必要なコード修正	ループ直前にディレクティブをプラグマとして追記する.	STL のアルゴリズムでループを置換する. すでにSTL のアルゴリズムを使用していれば修正の必要なし.
配列の各要素に関数を適用するループの並列化	`#pragma omp parallel for` をループの直前に追記する. ループ本体には副作用があってはならない.	`std::transform` でループを置き換える. ループ本体には副作用があってはならない.
合計を求めるループの並列化	`#pragma omp parallel for reduction(+:r)` をループの直前に追記する (+での合計を r に求める場合). 合計を求める際に使用できる演算子は+, -, *, &, |, ^, &&, || のみであり, 演算子オーバロードは不可. 合計値の型は基本型のみ.	`std::accumulate` でループを置き換える. 結合的であれば任意の演算子を使用可能. 合計値の型として任意の型を使用可能.
必要なコンパイルオプション	`-fopenmp`	`-fopenmp -march=native -D_GLIBCXX_PARALLEL`

表 -1　OpenMP と Parallel Mode による並列化のまとめ

```
1 template<typename E, typename F>
2 int querying(std::vector<E> &xs, F &f)
3 {
4     std::vector<int> cs(xs.size());
5     std::transform( xs.begin(), xs.end(),
          cs.begin(), f, __gnu_parallel::
          parallel_unbalanced);
6     int sum = std::accumulate( cs.begin(),
          cs.end(), 0, std::plus<int>(),
          __gnu_parallel::parallel_unbalanced);
7     return sum;
8 }
```

図 -8　Parallel Mode のバグ回避のために明示的に並列実装を使うよう書き直したプログラムコード (querying.h).

されており，libstdc++ のメーリングリストにはパッチが投稿されているため，このバグは今後のリリースで修正されるはずである．

上記のバグは本稿で紹介する `std::transform` と `std::accumulate` にも影響し，これらのアルゴリズムは逐次実装での計算となる．そのため，これらを並列実装で計算させるためには，パッチ☆1 を当てた GCC でコンパイルするか，これらのアルゴリズムの使用に対して明示的に並列実装を使うように引数を 1 つ追加する必要がある．たとえば，図 -6 のプログラムに対してバグ回避のための修正を施したプログラムは，**図 -8** のようになる．図 -6 のプログラムがうまく並列化されない場合には，修正を施したほうを試してもらいたい．

☆1 The libstdc++ mailing list archives に保存された September17, 2008 投稿のメール参照.
http://gcc.gnu.org/ml/libstdc++/2008-09/msg00116.html

これからのプログラミング

本稿では，GCC の容易な並列プログラミングのためのサポートとして，逐次プログラムを簡単な注釈の付加だけで並列化できる OpenMP と，STL のアルゴリズムで書かれたプログラムを修正なしに並列化できる libstdc++ parallel mode とを紹介した．**表 -1** に本稿で紹介した並列化のまとめを示す．特に，後者では，アルゴリズムレベルで構造化されているプログラムには並列化にあたってのコード修正が不要であるという大きな利点が見出せる．これからのプログラミングでは，このような恩恵を少しでも多く受けられるよう，汎用的なアルゴリズムの組合せでプログラムを書くのがよいだろう．

参考文献

1) GNU Project : GCC, the GNU Compiler Collection, http://gcc.gnu.org/
2) OpenMP Architecture Review Board : OpenMP Application Program Interface, Version 3.0, http://openmp.org/wp/openmp-specifications/
3) 柏原正三 : C++ 標準ライブラリの使い方完全ガイド, 技術評論社 (2005).
4) Singler, J., Sanders, P. and Putze, F. : MCSTL : The Multi-core Standard Template Library, Euro-Par 2007, Parallel Processing, LNCS 4641, Springer, pp.682–694 (2007).

(平成 20 年 9 月 30 日受付)

江本　健斗　emoto@ipl.t.u-tokyo.ac.jp

2004 年東京大学工学部計数工学科卒業．2006 年同大学院情報理工学系研究科修士課程修了．同年博士課程へ進学．現在，同特任研究員として並列プログラミングに関するプログラミング言語面からの研究に従事．