ハイパフォーマンス
コンピューティング 78－9
（1999．10．7）

疎行列 LU 分解の SMP 上での並列化

須田礼仁

名古屋大学　工学研究科　計算理工学専攻

概要

OpenMP を用いて SMP 上で非対称疎行列 LU 分解の並列化を行った。アロケーションには Geist-Ng のアルゴリズムを改良したものを用い、メモリ消費が少なく書き込みの衝突が生じない中間積形式を用いて並列化を行った。性能解析として、並列化オーバーヘッドの内容をメモリ周りのオーバーヘッド、負荷の不均衡、同期その他に分類して分析を行った。その結果、スレッドへの割付けに対応した配列のソートと、スレッド私有配列を用いた並列性の利用と通信量の削減という、スレッドを意識したプログラミングによって性能が改善されることを明らかにした。

Parallel sparse LU decomposition on an SMP machine

Reiji Suda

Dept. of Computational Science and Engineering,
Graduate School of Engineering, Nagoya University

Abstract

We have parallelized an unsymmetric sparse LU decomposition program on an SMP machine using OpenMP/C. We propose an improvement on the Geist-Ng algorithm for task allocation. The middle product form is used for the decomposition, because it requires less storage and is free from write-write hazard. We analyze the parallelization overheads by breaking it down into three factors: memory access overheads, load imbalance, and others including syncronization. It is demonstrated that thread-conscious programming such as a sorting of array elements by accessing thread and use of thread-private arrays for higher utilization of parallelism and for reduction of communication volume.

1. はじめに

いよいよ実用領域に入ってきている並列計算であるが、並列計算機のアーキテクチャのスペクトルはかなり広い。本稿では最近（超）高性能ワークステーションとしてよく見掛けるようになった SMP (Symmetric MultiProcessor) による並列計算を取り上げ、半導体論理回路の過渡解析中の線形解法の並列化における性能特性を考察する。

回路解析に現れる連立一次方程式は、係数行列の性質が悪く一般的な反復法は適用できないので、枢軸選択付きの LU 分解法が用いられる。非対称不規則疎行列に対する LU 分解は、逐次性が高くて効率的な並列化が難しいとされているアルゴリズムである [2]。本稿では並列化における種々の選択肢について考察したうえで、一つの方法を選んで実装・評価を行う。

プログラミングには OpenMP/C を用いた。OpenMP [10] は 1997 年に FORTRAN の、1998 年に C/C++ のインタフェースが発表された共有メモリモデルに基づく並列処理プログラミングのための枠組みである。基本的にはベース言語に並列化指示（プラグマ）を埋め込むことによりスレッド化して並列プログラミングを行う。逐次コードに最小限の指示の追加で並列化ができる点がスレッドプログラミングと異なっているが、高い並列化効率を実現するためにはかなり工夫が必要である。実際今回の研究では、スレッド毎のローカリティ向上の工夫やスレッド私有配列といった、スレッドを十分に意識したプログラミングが最高の並列性能を実現するためには必要であることが明らかとなった。

2. 非対称不規則構造疎行列 LU 分解

LU 分解法は連立一次方程式を解く最も汎用的なアルゴリズムである。そのアルゴリズムの基礎は

$$a_{ij} = \bar{a}_{ij} - \sum_{k<\min\{i,j\}} a_{ik} a_{kk}^{-1} a_{kj}$$

という一つの式に要約される（厳密にはこれは LDU 分解である）。ここで $\bar{a}_{ij}$ は（分解前の）係数行列の要素、a_{ij} は分解後の行列の要素で、L, D, U を一つの行列に重ねて格納することを仮定した表記である。

今回取り上げる問題は回路の過渡解析中に現れる連立一次方程式である。これは性質としては非対称不規則構造の疎行列である。しかし、一度消去の順序を決定すると大抵そのままの順序で何度も計算することができるので、非零構造や並列性についての情報を利用することができる。今回はまず基本的な場合とし

て、枢軸選択順序を変更することはない場合について並列化を行った。この場合でも、コレスキー分解の場合よりもさらに複雑な非零構造を有しているためかなり並列化の難しい問題である。

2.1. 枢軸選択

枢軸選択は、密行列の場合には数値的な安定性だけ、対称行列の場合には疎行列性（計算量・並列性）だけをそれぞれ考えて行えばよい。しかし、非対称な疎行列の LU 分解の場合には、枢軸選択が数値的な安定性と疎行列性の両方に影響をおよぼす。このため、枢軸選択は「ある数値的な安定性を確保したうえで、最も望ましい疎行列性を持つ」という基準で選択する必要がある。今回使用した枢軸選択は、枢軸要素の絶対値は同じ行・列の要素の最大絶対値の α 倍よりも大きいという制限つきの Markowitz-Tewarson 法である。ここで α は 1 以下の定数で、$\alpha = 1$ とすると完全枢軸選択になる。今回用いた例題では、α がいくつであってもそれほど疎行列性に違いは出ないようである。これは、数値的な安定性をある程度保証すると、消去順序がかなり制約されるということを意味しているようである。

数値的な安定性を確保するために、今回は枢軸要素が行・列両方に対して優位となるように制限をつけている（**完全枢軸選択**）が、どちらか一方だけを考慮する（**部分枢軸選択**）のが一般的である。完全枢軸選択は数値的な性質は非常に望ましいが、枢軸選択そのものの計算量がかなり大きいという欠点がある。今回は枢軸選択に要する時間は考慮していないが、今後は並列化を含めて十分な検討が必要である。

疎行列性については、今回は計算量の最小化だけを考えている。そもそも OpenMP は逐次プログラムの並列化を主なターゲットとした並列プログラミング手法であるため、並列化の対象としなかった枢軸選択部分は並列処理を考慮にいれていない。しかし、今後はさらに nested dissection [5] など並列処理向きの枢軸選択アルゴリズムも考慮にいれるべきであると考えられる。

2.2. 行列要素の扱い

不規則疎行列といっても、実際にはところどころに非零要素の塊ができる。これをうまく利用すると、行列要素へのポインタが少なくて済み、メモリ消費や間接アクセスの量を抑えることができるようになる。また、レジスタブロッキングやキャッシュブロッキングなどが出来るようになり、実行効率の向上が可能となる。対称行列の場合には消去木からスーパーノード [8] という形でこの非零要素の塊を検出することが可能である。しかし非対称行列の場合にはそれほど簡単でもないし、効果もやや弱いことが知られている [1]。

今回はスーパーノードについては考えずに、単純にそれぞれの要素を別々に扱っている。今回の研究の主要な目的が並列処理にあるので、この難問に取り組むことは避けた形であるが、今後はこの問題にも取り組まなければならない。

3. 疎行列 LU 分解の並列性

```
1  for i = k + 1, n
2      for j = k + 1, n
3          a_ij -= a_ik a_kk^{-1} a_kj
```

Prog. 1. k に関する掃き出し (外積形式)

疎行列 LU 分解のアルゴリズムには大きく分けて 2 種類の並列性があると考えられる。それは掃き出し間の並列性と、掃き出し内の並列性である。これはプログラミングの仕方によって若干異なってくるが、ここでは Prog. 1 のような計算を (k に関する) 一つの**掃き出し**と呼ぶことにする。

この一つの掃き出しの 2 重ループの計算の中身に依存性はないので、完全に並列に計算することができる。これが**掃き出し内の並列性**である。疎行列の場合には通常計算の最初の方の掃き出しは疎行列性のために極めて計算量が少なく、その並列性は極細粒度となる。しかし計算の終わりに近くなると **fill-in** のために計算量が多くなるので、並列化を考えることができる程度の並列性を持つようになってくる。

密行列とは異なり、疎行列の場合にはいくつかの異なる掃き出しを並列に実行することが可能となる場合がある。これが**掃き出し間の並列性**である。実際には掃き出し間にまったく依存がない場合と、修正を行う a_{ij} が衝突する場合とがある。後者の場合には、一時配列の使用やループ順序の変更によって並列化が可能である。掃き出し間の並列性は粒度が粗く並列化に利用するのに適している。但し、分解計算の最初の方では高い並列性があるが、計算の終盤に向かって急速に並列性が落ちてゆく。

掃き出し内の並列性は細粒度なので、できれば掃き出し間の並列性だけで並列化したいところである。しかし、掃き出し内の並列性を全く利用しない場合には、問題規模を大きくしても速度向上率や並列化効率は理想値に近づかず、すなわち粒度に関するスケーラビリティを有していない。プロセッサ数や問題規模が小さい場合にはそれほど問題にならない場合もあるが、掃き出し内の並列性も利用して並列化を行うことが望ましい。

さらに連立一次方程式を解くためには分解だけではなく、**代入計算**も必要である。代入計算は計算量は分解に比べてかなり少ないが、密行列の場合と異なり疎行列の場合には、分解計算との計算量の比が問題規模を大きくしても一定にとどまる傾向にある。従っ

て、十分なスケーラビリティを得るためには代入部分も何らかの並列化を行わなければならない。

以上のように、疎行列 LU 分解の並列化の際には 3 種類の並列化を考える必要がある。以下、それぞれについてさらに詳しく考察する。

3.1. 掃き出し間の並列化

掃き出し間の並列化の際には、修正先である a_{ij} の衝突をどのように解決するかが問題である。

第 1 に、a_{ij} が衝突した場合には並列化しないことが考えられる。しかし、掃き出し間の並列性は疎行列 LU 分解では最も粒度が粗く効率的に並列化できる部分であるから、持っている並列性はできるだけ引き出すのが望ましい。従ってこの選択は良くない。

第 2 に、書き込みが衝突するような a_{ij} については**スレッド私有**の一時領域に書き込んでおき、最後に結果をまとめるということが考えられる。この方法を用いれば粗粒度の並列性を最も有効に利用することができる。しかし、スレッド私有の領域のためにメモリを余計に必要としてしまう。また、スレッド数に依存して総和の順序が変更されてしまうため、計算結果が微妙に異なってしまうという問題もある。

```
1  for k = 1, i − 1
2      for j = i, n
3          a_ij −= a_ik a_kk^{-1} a_kj
4      for j = i + 1, n
5          a_ji −= a_jk a_kk^{-1} a_ki
```

Prog. 2. i に関する掃き出し (中間積形式)

第 3 に、書き込みが衝突しないように掃出し計算を定義するということが考えられる。これは例えば Prog. 2 のような掃き出しを考えれば可能である。この場合には i が異なれば修正される要素が異なるので、書き込みの衝突は決して起らない。この方法では第 2 の方法のような余計なメモリは必要ないし、計算結果はスレッド数に依存しない。また、このような中間積形式の場合には、不規則形状疎行列の非零要素へのポインタに要するメモリ量が少なくて済むという特長がある。しかし、掃き出し間並列で並列処理される部分の計算量はやや少なく、粗粒度の並列性を十分に活かしているとは言えない。また、他のスレッドが書き込んだデータを読み込む「**通信量**」が第 2 の方法よりも多い傾向にある。

3.2. 掃き出し内の並列化

掃出し間の並列性が不十分となる分解計算の最後の部分では、掃き出し内の並列処理を行わなければならない。もちろん、個別の掃き出しを並列化するのでは粒度が細かすぎるので、**ブロック化**が必要である。掃き出し内の並列化にはさまざまな選択肢がある。

第 1 に、ブロック化を **1 次元**で行うか、**2 次元**で行うかが問題である [7]。1 次元分割の場合には大抵の掃き出し形式で可能であるが、2 次元分割を行うためには Prog. 2 のような中間積型は向いていない。1 次元分割には、行方向・列方向の他に、Prog. 2 の修正領域のような**逆 L 字型**の分割も考えられる。しかし、1 次元分割では並列度やロードバランスが問題になり易く、「通信量」も多くなる傾向にある。

第 2 に、行列の私有化を行うかどうかが問題である。行列を共有メモリに取っておくと、書き込みの衝突や false sharing が問題となりうる。しかし、行列の私有化を行うと、余計なメモリを必要としたり、スレッド数によってメモリアロケーションを変えなければならないなどややプログラムが面倒になる。OpenMP が、逐次プログラムと同じソースで並列化できる利点を活かすためには、行列の私有はできるだけ避けたい。

第 3 に、ブロックサイズをどのように決定するかが問題となる。さらに、同期をどのように行うか (バリアで良いか、1 対 1 の同期を行うか) も問題である。

3.3. 後退代入の並列化

前進代入は分解と一緒に行うことができるので、実際に問題になるのは後退代入の部分だけである。

ここでは第 1 に、分解部分の並列化とデータ分割を合わせるかどうかが重要な問題となる。対称行列の場合とは異なり、非対称行列の場合には分解と代入では依存性が若干異なっており、分解計算でも代入計算でも依存がないようにデータ分割をすると掃き出し間の並列性が少し減ってしまう。また、分解計算と代入計算では計算量が異なるので、最適なロードバランスを得るためには個別にアロケーションを行う必要がある。しかし、分解と代入で異なるデータ分割で並列化をすると再分散に相当する「通信量」が必要になるという問題がある。実際現代のワークステーションでは代入計算の所要時間はほとんどメモリアクセスなので、できるだけ「通信量」は抑えたい。また、行列のスレッド私有化を行っている場合には、代入で再分散を行うのはかなり面倒である。

代入計算は 1 次元、2 次元の分散が可能であるが、2 次元にしてもそれほど性能的に魅力はないと思われる。1 次元の分割の場合、列で分割することも行で分割することも可能である。

4. 並列実装と評価

本節では、今回実際に行った並列化について詳述する。代入計算の並列化については分解計算の並列化にあわせて並列化を行うことにした。これは必ずしも

最適とは限らないが、代入計算の所要時間は短いのでとりあえずこれで済ました形である。また、掃き出し内並列性の利用には今回は至らなかった。これについては今後研究を進めて行きたい。

今回は分解計算に中間積形式を利用した。これはスレッド私有の配列が不要で、OpenMP ですなおに並列化ができるほか、使用するメモリ量が少ないことに注目したものである。また、スレッドの生成などの時間を軽減するために回路解析全体を parallel 指示行の中にして、**orphan directive** によって並列処理を行うようにプログラムした。これによって下記の例では 0.1 秒程度実行が速くなる。但し、実際に並列処理を行っているのは線形解法の部分だけである。

プログラムは C 言語、OpenMP コンパイラは RWCP で開発した Omni compiler [9] で、バックエンドは SUN WorkShop C Compilers である。最適化オプションは `-xO2 -native` である。このオプションでは `-fast` などよりも 1.3 倍ほど実行が遅いが、`-xO3` 以上の最適化オプションを設定すると共有データへの書き込みが正しく行われないため、このようにせざるを得なかった。ハードウェアは SUN Enterprise 450 の 4 CPU で、メインメモリは 1 GB である。

4.1. 掃き出し間並列性解析

前節で説明したように、最初のうちは掃き出し間に並列性がある。コレスキー分解の場合には掃き出し間依存性のグラフは木になるが、非対称行列の場合には**依存性グラフ**は DAG になる。今回は共通の子孫を持つノードの間に依存性を追加することにより、グラフを木に変更した。こうすることにより依存性の解析は簡単になるが、若干の並列性を無駄にしている可能性がある。しかしここで無駄になった並列性は、活かしたとしても粒度が十分ではない可能性があると予想した。

木の構造となった掃き出し依存性グラフを、以下のように各スレッドに割付ける。これはコレスキー分解のタスクアロケーションのための Geist と Ng [4] によるアルゴリズムを少し改良したものである。Geist-Ng のアルゴリズムの基本はグリーディなビンパッキングで、部分木を計算量の大きいものから順に、割付けられている計算量が最も少ないスレッドに割付ける。割り付けの結果のロードバランスが満足できるものでない場合には、最も大きな部分木の根の要素を取り除く。もしこの最も大きな部分木の根が複数の子を持っていれば、この部分木が分割されることになる。この Geist-Ng のアルゴリズムはロードバランスで停止判定を行っているが、今回はアロケーションの対象からはずされた（掃き出し間並列性を用いて並列化されない）部分の計算量を考慮にいれて、推定並列計算時間が最小になるようにアルゴリズムの修正を行った。ロードバランスのみを考えたアルゴリズムでは目標とするロードバランスの定義の仕方によりアロケーションの結果が極度に異なるという問題があったが、改良によりロードバランスの条件設定によらずに安定にアロケーションができるようになった。

4 スレッドで実行する場合、並列性が 4 の掃き出し間並列がなくなっても、並列性が 2 の掃き出し間並列があればそれを 2 スレッドずつペアにして並列に処理することが考えられる。このような並列処理は **subtree-to-subcube mapping** [6] と呼ばれている。今回はこの subtree-to-subcube mapping (以下この方法を **tree mapping** と呼ぶ) のほかに、スレッド数だけの並列性がなくなってしまったら掃き出し間並列性を利用しない (以下 **flat mapping** と呼ぶ) 方法も実装した。Tree mapping の方が並列性を有効に利用できるように思われるかもしれないが、小さい問題では並列度が 2 の掃き出し並列というのは計算量が少なく、粒度的に不利となってしまいがちである。

4.2. 性能評価

次に性能について報告する。解析に用いた回路は ECL による carry-look ahead 付きの 32 ビットの加算器である。加算器は基本的な論理回路としては比較的複雑な構造をしていて、うまく分割するのが難しく、並列化効率を出すのが難しい例題である。連立一次方程式の未知数は 3522 である。300 ステップの過渡解析中、連立一次方程式は 1028 回呼ばれる。実験はそれぞれの条件で 3 回連続で行い、合計実行時間が中間の結果を用いた。所要時間は gettimeofday を用いた wall-clock time で計った。実行ごとに所要時間は若干異なるが、所要時間のばらつきは 0.01〜0.05 秒程度であった。

逐次実行では線形解法（中間積形式）に 24.53 秒かかっている。なお、シミュレーション全体では 49.0 秒、そのうち回路要素評価には 16.6 秒を要する。従って、線形解法と回路要素評価に 9 割近くの時間がかかっていることになる。

Flat mapping による並列化— Flat mapping を用いて 4 スレッドで並列処理させたところ、11.32 秒となった。理想的に並列化された場合には 6.13 秒で終了しなければならないから、5.19 秒の**並列化オーバーヘッド**がかかっていることになる。このオーバーヘッドの要因を調べるために、計算時間の内訳を調べた。それを図 1 に示す。

図 1 中の四角の中の数字はそれぞれ数値計算部分の所要時間であるが、その合計は 28.82 秒となり、逐次の時の 24.53 秒よりも 4.3 秒ほど余計にかかっている。この原因としてキャッシュミスヒットの増加や false sharing、あるいはメモリアクセスの際のバスの衝突などが考えられる。これら**メモリ周りのオーバーヘッド**として 1.07 秒かかっていることになる。

計算時間のクリティカルパス長を求めると 11.01

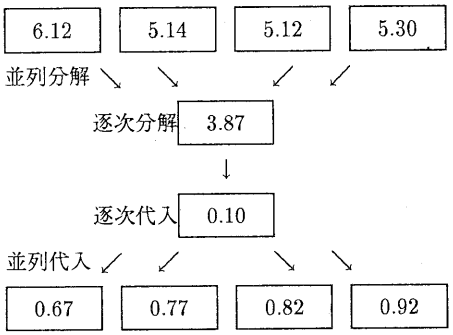

図 1. 並列 LU 分解の数値計算所用時間

秒となる。28.82 秒のロードが 4 つのスレッドに均等に割り当てられていれば 7.21 秒になるはずであるから、3.81 秒のオーバーヘッドは**負荷の不均衡**によることになる。この負荷の不均衡の内、掃き出し内並列を利用していないことによるもの（逐次分解・逐次代入部分）が 2.98 秒で、掃き出し間並列の負荷の不均衡（並列分解・並列代入部分）が 0.83 秒となる。

最後に、このクリティカルパス長と実際の実行時間との間には 0.31 秒の違いがある。これは OpenMP の関数呼び出しやバリヤ同期などにかかっている可能性があるが、実際の主要因は今のところ明らかではない。

以上をまとめると、5.19 秒のオーバーヘッドの内訳はメモリ周りで 1.07 秒、負荷の不均衡で 3.81 秒、同期その他で 0.31 秒ということがわかった。

行列要素のソート— このうち、メモリ周りのオーバーヘッドはキャッシュに関するものとバス・メモリの衝突によるものとが考えられる。ここで、行列要素を格納する配列を書き込みが行われる順番にソートをしてみた。すると数値計算の合計所要時間は 25.71 秒と 3.11 秒短くなり、全体の所要時間も 10.36 秒になった。並列化オーバーヘッドは 4.23 秒で、そのうちメモリ周りのオーバーヘッドは 0.30 秒と、大幅に改善された。その他のオーバーヘッドは負荷の不均衡は 3.60 秒、同期などが 0.33 秒である。

最初に示した 1 スレッドでの所要時間 24.53 秒は行列ソートを行った場合で、これを行わないと 26.36 秒かかる。配列のソートによって逐次の場合でも 1.8 秒ほど速くなるわけであるが、並列では 3.1 秒も速くなる。このような違いが出る原因としては、false sharing などが軽減されただけではなく、プロセッサあたりのワーキングセットが小さくなってキャッシュデータの再利用率が上がった可能性がある。

バス・メモリアクセスの衝突— さらに、バスの衝突の影響を調べるために、同期を余計に入れて複数のスレッドの計算が同時に実行されないようにしてみたところ、数値計算の合計所要時間は 25.43 秒とさらに減った。これと先の実行結果との差である 0.28 秒がバスや主記憶アクセスの衝突の影響と思われる。**衝突のオーバーヘッド**は 0.07 秒であるが、これは並列処理をしている 6.32 秒のおよそ 11 % に相当する値である。残った 0.22 秒のオーバーヘッドは、他のスレッドが書き込んだデータを読み込むための「**通信**」や、**false sharing** (行列要素はソートしたが、ベクトル要素はソートしていない) によるものと思われる。

Tree mapping— 行列のソート付きで割り付けを tree mapping にしてみたところ、所要時間はさらに短縮されて 9.85 秒となった。オーバーヘッド 3.72 秒の内訳はメモリ周りで 0.28 秒、負荷の不均衡で 3.16 秒、同期などで 0.28 秒である。負荷の不均衡は改善されたものの、3.60 秒の内わずか 0.44 秒であった。これは、4 未満で 2 以上という中途半端な並列性が非常に少ないためであると思われる。この部分の並列性は掃き出しの順序によって決まってしまうものであるので、掃き出し順序を並列処理に向いたものに改良することも必要であろうと思われる。

スレッド私有配列の利用— 以上、最終的には 9.85 秒ということで、4 スレッドで 2.49 倍の速度向上が得られた。理想的なスピードアップに至らない原因であるオーバーヘッドの分析は上述の通りであるが、その大半は負荷の不均衡によるものである。負荷の不均衡のうち 1.19 秒は割付けアルゴリズムが想定した負荷分散と実際の実行時間とのずれであるため避けがたいが、残る 1.97 秒は並列性を利用していないことによるものである。まだ利用していない並列性のうち、一部の修正計算についてはスレッド私有の一時的な行列に結果を格納することにより並列処理が可能となる。

これを実際に行ってみたところ、所要時間は 8.15 秒とかなり短縮された。オーバーヘッドは 2.02 秒でそのうちメモリ周りで 0.16 秒、負荷の不均衡で 1.52 秒、同期などで 0.34 秒であった。負荷の不均衡がおよそ半分に軽減されただけではなく、メモリ周りのオーバーヘッドも半分近くにまで減っているのは注目に値する。他人が書き込んだデータを読み込む「通信」の量をカウントしてみると、以前は 8208 ワード (double のデータ一つを 1 ワードとしている) であったものが 2304 ワードに減少している。負荷の不均衡に比べれば小さいとは言え、単純計算で 1000 ワードでおよそ 20 μs と、同期などに比べると意外と大きなオーバーヘッドである。しかも、実際にはスレッド私有のデータの初期化や集積のために全体としてのメモリアクセス量は増えているはずであるので、この「通信」のオーバーヘッドは実際はさらに大きいはずである。

まとめ— 以上、中間積表現を用いた疎行列 LU 分解を、掃き出し間並列性のみを用いて並列化を行った。表 1 はそのまとめである（略語の説明は省略する）。その結果、データ割り付けにしたがってソートすることによりメモリ周りのオーバーヘッドが軽減でき

ること、他のスレッドが書き込んだデータを読み出す「通信」のオーバーヘッドもかなりあることがわかった。また、共有メモリプログラミングでは、書き込みの衝突はスレッド私有の配列を用いずにループ変換で解決するのがふさわしいように思われるが、それでは並列性や局所性を無駄にしかねないということも明らかとなった。

	TIME	OVHD	MEM	LOAD	SYNC
flat	11.32	5.19	1.07	3.81	0.31
sort	10.36	4.23	0.30	3.60	0.33
tree	9.85	3.72	0.28	3.16	0.28
priv	8.15	2.02	0.16	1.52	0.34

表 1. 並列化オーバーヘッドのまとめ

本文では詳述しなかったが、バスや主記憶の衝突のオーバーヘッドは flat mapping では 0.1 秒程度あったが、tree mapping ではほとんど見られなかった。

今後は掃き出し内の並列性の利用も考えなければならないが、実はこれは容易ではない。現在逐次処理されている部分の行列は、flat な割り付けでは 59 次の行列になる。これが密であれば共有メモリなら簡単に並列化できるが、実際にはかなり帯行列に近いため並列性が低い。この部分の並列処理の検討は今後の課題である。

5. まとめ

本稿では不規則疎行列 LU 分解の SMP 上での並列処理についての研究について報告した。キャッシュアクセスや「通信」量などのローカリティについての工夫をすることにより性能が向上すること、およびスレッド私有配列を用いることにより効率的に並列性が引き出せることを示した。

OpenMP などの共有メモリモデルによる並列化は MPI のようなメッセージパッシングによる並列化よりも容易であろうか。これは並列化する対象のアルゴリズムに依存するように思われる。そもそもアルゴリズムの並列性が複雑であることもあって、今回取り上げたこの問題に関しては並列化プログラミングはメッセージパッシングに比べて易しいとは思われなかった。もちろん、あまり重要でない部分を適当に処理してしまうのは共有メモリプログラミングの方が簡単である。しかし、send-recv 型では他人の作った情報が自分の世界に入り込んでくるタイミングを明示的に制御できるのに対し、共有メモリや put-get を用いた場合にはそれを同期という方法で間接的にしか制御できない。このため、共有メモリでのバグはタイミング依存で再現性がないことが多く、メッセージパッシングの方がデバッグがしやすいと思うこともあった。

今後は掃き出し内の並列処理と枢軸選択を含めた並列化を考えて行かなければならない。さらにさまざまな選択肢を実装して評価し、SMP という環境における非対称疎行列 LU 分解の並列化の指針を得たいと考えている。

謝辞

OpenMP のコンパイラを使わせて頂きました、佐藤三久さんをはじめとする新情報処理開発機構の皆様に深く感謝致します。本研究の一部は日本学術振興会未来開拓学術研究推進事業および豊田理化学研究所研究嘱託による。

参考文献

[1] J. W. Demmel, S. C. Eisentat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, "LAPACK Working Note 103: A supernodal approach to sparse partial pivoting", Tech. Rep. 95-304, Univ. Tennessee, Knoxville, Computer Science Department, 1995.

[2] I. S. Duff, "Sparse numerical linear algebra: direct methods and preconditioning", TR-PA-96-22, CERFACS, 1996.

[3] B. Dumitrescu, M. Doreille, J.-L. Roch and D. Trystram, "Two-dimensional block partitionings for the parallel sparse Cholesky factorization: the fan-in method", INRIA rapport de recherche No. 3156, 1997.

[4] G. A. Geist and E. Ng, "Task scheduling for parallel sparse Cholesky factorization", *Int. J. Parallel Prog.*, No. 18, pp. 291–314, 1989.

[5] A. George, "Nested dissection of a regular finite element mesh", *SIAM J. Numer. Anal.*, Vol. 10, No. 2, pp. 345–363, 1973.

[6] A. George, J. W. H. Liu, and E. Ng, "Communication results for parallel sparse Cholesky factorization on a hypercube", *Parallel Computing*, Vol. 10, 1989, pp. 287–298.

[7] A. Gupta and V. Kumar, "A scalable parallel algorithm for sparse Cholesky factorization", *Proc. Supercomputing '94*, pp. 793–802, 1994.

[8] E. Ng, "Supernodal symbolic Cholesky factorization on a local-memory multiprocessor", *Parallel Computing*, Vol. 19, pp. 153–162, 1993.

[9] RWCP の Omni コンパイラホームページ
`http://pdplab.trc.rwcp.or.jp/pdperf/Omni/`

[10] OpenMP ホームページ
`http://www.openmp.org/`