高い耐遅延性を持つガウス消去法

遠 藤 敏 夫[†] 田 浦 健 次 朗[†]

密に通信を必要とする並列計算をグリッド環境において行なう上での障害は、広域ネットワークの高い通信遅延である。本稿は、そのような計算の一つとして密行列のガウス消去法を取り上げ、高遅延環境でも高性能な並列アルゴリズムを述べる。その主要な技術は batched pivoting と呼ばれるピボット選択手法である。本手法は、複数ステップのピボット選択処理をまとめて行なうことにより、同期コストを大幅に削減する。遅延をエミュレートした実験により、高遅延環境において本手法が partial pivoting よりもはるかに高速に動作することを示す。一方、本手法では partial pivoting よりも計算精度が低下する可能性があるが、比較的良好なピボットを選択することにより、その低下を抑えるよう設計されている。乱数行列を用いた数値実験を通して、本手法が partial pivoting に匹敵する計算精度を達成することを示す。

Highly Latency-tolerant Gaussian Elimination

Toshio Endo[†] and Kenjiro Taura[†]

Large latencies over WAN will remain to be an obstacle to running tightly coupled parallel applications on Grid environments. This paper takes one of such applications, Gaussian elimination of dence matrices and describes a parallel algorithm that is highly tolerant to latencies. The key technique is a pivoting strategy called batched pivoting, which largely reduces synchronization costs by batching pivot selections of several steps. Through experiments with large latencies emulated by software, we show our method works much faster than partial pivoting with large latencies. On the other hand, numerical accuracy of our method may be inferior to that of partial pivoting. However, our method is designed to suppress the degradation by selecting 'better' pivots. Through experiments with random matrices, the batched pivoting achieves comparable accuracy to that of partial pivoting.

1. はじめに

近年、大規模な科学技術計算やデータ解析のためにグリッド環境を利用する動きが広まっている。特に、サブタスク間の依存関係の少ない、疎結合な並列アプリケーションにおいてはすでに多くの成功例が見られる[1),3)]。一方、密に通信を必要とするような並列アプリケーションをもグリッド環境で動作させるために、グリッド向けプログラミングツールの研究が広がってきている[6),9),12)]。しかし、グリッド環境で高性能な密結合計算を行なうためには、広域ネットワーク帯幅の不足や高い通信遅延などが依然として障害となる。このうち帯幅についてはバックボーンネットワークの拡充に伴い、近い将来に解決されると考えられる。しかし、広域ネットワークの通信遅延は 10–100 ミリ秒のオーダより小さくはならず、遅延が数マイクロ秒であるスーパーコンピュータとの差が縮まることはない。したがって、グリッド環境で高性能な密結合計算を実現するためには、高い通信遅延に耐えることのできる並列アルゴリズムの研究が重要となると考えられる。

本稿では密行列を係数とした連立一次方程式を取り上げる。これはスーパーコンピュータのランキングである Top500[4)] においても用いられる問題である。その代表的な解法は High Performance Linpack(HPL)[10)] でも用いられている、partial pivoting を用いたガウス消去法である。しかしこの手法は頻繁なノード間同期を必要とするため通信遅延の影響を大きく受けてしまい、高遅延環境では良好な性能を得ることができない。本稿は高遅延に耐えることのできるピボット選択手法である *batched pivoting* を提案する。本手法は複数ステップのピボット選択処理をまとめて行なうことにより、同期コストを大幅に削減する。また本手法は良好な計算精度を達成するために、最良とは限らないが比較的良好なピボットが選択されるよう、設計されている。

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, University of Tokyo

以下、2 章で partial pivoting によるガウス消去法について説明し、3 章で我々の batched pivoting について述べる。そして本手法を 4 章で速度性能の面から、5 章で計算精度の面から評価する。6 章で本手法の適用範囲などを議論し、7 章でまとめを述べる。

2. Partial pivoting によるガウス消去法

Partial pivoting によるガウス消去法は、密行列を係数とする連立一次方程式 $Ax = b$ を解く代表的手法である。A が $n \times n$ 行列のとき、浮動小数演算量は合計で $(2/3)n^3 + O(n^2)$ である。ここでは簡潔にそのアルゴリズムを述べ、その性能が通信遅延の影響を大きく受けることを述べる。

2.1 アルゴリズム

図 1 はアルゴリズムの概要を示す。簡単のために、並列化や最適化に関するコードは省略されている。最外ループの各ステップはピボット選択フェーズ、行交換フェーズ、更新フェーズから成る。第 k ステップにおけるピボット選択フェーズでは、第 k 列の中から絶対値が最大であるような要素がピボットとして選ばれる。行交換フェーズでピボットを含む行と、第 k 行が交換される。更新フェーズでは、第 k 列と (交換後の) 第 k 行の要素群を用いて、行列中の第 k 列より右であり第 k 行より下の部分が全て更新される。

絶対値が最大の要素をピボットとして選ぶ目的は、計算精度の悪化を防ぐことである。選択されたピボット a_{kk} を用い、更新フェーズで要素 a_{ij} に $-a_{ik}a_{kj}/a_{kk}$ という値が加算される。ここでもし a_{kk} がゼロに近いと、更新結果の絶対値が非常に大きくなり、大きな計算誤差が蓄積されてしまう。

2.2 高遅延環境における問題

並列ガウス消去の効率化のために、これまで様々な最適化技法が実装されてきた[10)]。特に、複数ステップにまたがった行交換フェーズや更新フェーズをまとめて処理する技法は重要であり、通信コストやキャッシュミスコストの大きな削減が可能である。また、ピボット選択以外の部分は、まとめ処理やパイプライン処理などの技法により、外乱や高い遅延に耐久可能である[5)]。しかし partial pivoting が導入されると、以下の理由により遅延の影響を大きく受けてしまう。

多くの並列実装において、係数行列は二次元ブロックサイクリック分割により分散配置されている。この場合、各列は複数ノードにまたがっているため、ピボット選択の度に通信処理が必要となる。しかも、各ステップのピボット選択は前のステップのピボット選択の結果に依存するため、異なるステップにおけるピボット選択をオーバラップさせることはできない。

以上のことから、partial pivoting における総同期コストは、l を通信遅延とするとき、少なくとも $O(nl)$ であることが分かる (なお、HPL においてはピボット候補の交換のために binary exchange アルゴリズムを用いるため、少なくとも $nl \log p$ のコストがかかる (p は計算ノード数))。このコストは遅延が非常に大きいときに、全体の計算を律速するかもしれない。同期コストがボトルネックとならないような遅延を「耐久可能」と呼ぶことにすると、その最大値は行列サイズ n と総 CPU 処理性能に依存する。図 2 はそのような耐久可能な遅延の最大値を、さまざまな n と Flops 値について示す☆。グラフによると、行列サイズが 10^6 のときに 100TFlops を達成するためには、遅延は 6.6 ミリ秒以下でなければならない。2005 年 6 月の Top500 ランキングによると IBM BlueGene/L が $n = 1,277,951$ において 136.8TFlops を達成しているが、通信遅延が 10 ∼ 100 ミリ秒のオーダであるグリッドではそれに匹敵する性能を得ることはできないことが分かる。

以上のような同期コストを削減するためには、更新ステップで行なわれているような複数ステップのまとめ処理を、ピボット選択フェーズでも行なえれば良い。単純には、複数列の全要素を単一ノードに集めて処理してしまうことにより、同期回数を削減することは可能である。しかしこの方法ではスケーラビリティが低下してしまう。次章では、スケーラビリティを保ちつつ、ピボット選択をまとめて処理する手法を述べる。

```
for (k = 0; k < n; k++) {
    /* ピボット選択 */
    第 k 行において |a_pk| が最大となる要素
    (ただし p ≥ k) をピボットとして選択
    /* 行交換 */
    第 p 行と第 k 行を交換
    /* 更新 */
    for (i = k + 1; i < n; i++) {
        a_ik = a_ik / a_kk
        for (j = k + 1; j < n; j++) {
            a_ij = a_ij − a_ik a_kj
        } } }
```

図 1 Partial pivoting によるガウス消去法アルゴリズム

☆ これは単に nl と $(2/3)n^3/f$ を比べる楽観的な見積りである (f は総 Flops 値)。交換アルゴリズムの影響等を考慮に入れると、より耐久は困難となる

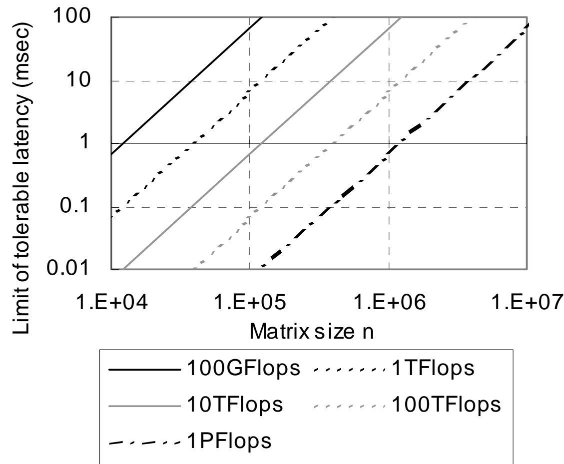

図 2 Partial pivoting における、耐久可能な遅延の限界の見積り。遅延がこの限界よりも大きい時、同期コストが全体の計算においてボトルネックとなる

3. 提案手法

3.1 アルゴリズム

高い耐遅延性を持つガウス消去法を実現するために、*batched pivoting* という手法を提案する。本手法は、複数ステップのピボット選択処理をまとめて処理することにより、頻繁な同期を避ける。まとめるステップ数 (以下、バッチステップ数と呼ぶ) を d とすると、同期コストは $O(nl)$ から $O(nl/d)$ へと削減される。このため、耐久可能な遅延の範囲は d 倍に拡大される。なお、この手法において選ばれるピボットは、partial pivoting ほどは良いとは限らないが、比較的良いピボットを選ぶことに焦点を置いている。

以下では、Batched pivoting が第 k ステップから第 $(k+d-1)$ ステップまでのピボット選択をどう行なうか述べる。ここで参照される領域は第 k 列から第 $k+d-1$ 列までの d 本の列である。図 3 は、行列が二次元ブロックサイクリック分割されている時の様子であり、d 本の列は灰色で示される。以降、これらの列を分け合うノード群を分担ノードと呼ぶ (本図においては $P1$ と $P2$ である)。Batched pivoting では、これらの分担ノードが局所的に d ステップ分の「ピボット候補」を選択し、その後それらを集計して最終的なピボットを決定する。

ここで、まとめたピボット選択が可能になるために必要な、データ分割に関する条件を述べる。注目する d 本の列の領域は、行方向にのみ分割されていなければならない。図 3 のように、データが d よりも大きいサイズのブロック単位でノード間に割り当てられていれば、この条件は満たされる。このとき、各分担ノードは図の右部分のように、概念的に $c \times d$ のサイズの部分行列を持つ。ここで c は図中の灰色の領域のうち各ノードが持つ行の数であり、ノードによって異なりうる。

以下に、第 k ステップから第 $(k+d-1)$ ステップまでのピボット選択手法の詳細を示す。

(1) 各分担ノードは以下のようにピボット候補のリストを決定する。ノードは自分の $c \times d$ 部分行列に対して局所的に partial pivoting によるガウス消去法を行なう。この計算は投機的なものであり、元行列を書きかえないように一時的な作業領域をもちいる。その過程において見つかった d 個のピボット要素とその位置を、ピボット候補リストとして記録する。

(2) 各分担ノードは自分のピボット候補リストの「得点」を計算する。得点は $\min_{k \le k' < k+d} |p_{k'}|$ ($p_{k'}$ は第 k' ステップにおけるピボット要素) として計算される。得点が大きいとより良いピボット候補リストと見なされる。

(3) 全分担ノードのピボット候補リストとその得点を通信によって集める。その中で最も大きい得点をもつ候補リストを決定版として選び、全分担ノードへ送る。

(4) 各分担ノードは受けとった決定版のピボットを用い、残りの計算を続ける。

選択されるピボット Batched pivoting においては、まとめて選択される d 個のピボットは必ず同一ノードによって選ばれる。このような性質のため、partial pivoting とは異なるピボットが選ばれる可能性が高い。しかし、各ノードが分担範囲の限りで最良のピボットを求め、さらに候補リストの中から最良のものを選ぶため、高い確率で良好なピボットを選択できると期待できる。

演算量 Batched pivoting では局所ガウス消去計算のため、浮動小数演算が増加する。しかしその増加量は $O(dn^2)$ であり、$d \ll n$ ならば元々の演算量 $(2/3)n^3 + O(n^2)$ に比べて小さいと言える。

3.2 他手法との比較

Partial pivoting よりも厳格でないピボット選択手法は広く研究されている。Threshold pivoting[8] においては、batched pivoting と同様に、ピボットとして列中の絶対値最大要素 (v とする) が選ばれるとは限らない。代わりに、$a_{pk} \ge \tau v$ ($0 \le \tau \le 1$ は前もって定められたパラメータ) を満たす要素のうち任意のものを選ぶ自由がある。そのため行交換のための通信量

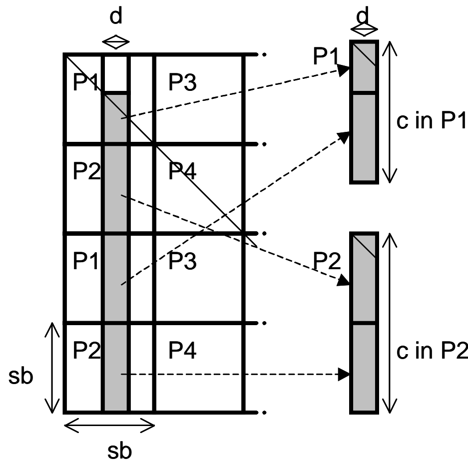

図 3 Batched pivoting の様子を示す。行列は二次元ブロックサイクリック分割されている。d 本の列がノード $P1$ と $P2$ によって分担されているとき、各ノードは自分の持つ $c \times d$ 部分行列に対して局所ガウス消去を行なう。

が少なくなるようにピボットを選ぶことができる。しかし、各行の絶対値最大要素を得るためにステップ毎の同期が依然として必要であり、遅延の影響を軽減することはできない。

これまで述べた手法は全て各ステップで一つのピボットを選択するものだったが、pairwise pivoting[11] は異なるアプローチを取る。この手法では隣り合った二行を行列の下側から順に取り出し、必要があれば交換し、二行のうちの一行を消去／更新する、という処理を繰り返す。この手法では異なるステップのピボット選択をオーバラップできるため、耐遅延性が高いと考えられる。しかし、第 5 章で示すようにこの手法は平均的な計算精度が悪い。

以上の手法と異なり、batched pivoting は耐遅延性と平均的な計算精度に注目し、これらを両立させることに焦点を置いている。

4. 耐遅延性評価

Batched pivoting によるガウス消去法の耐遅延性の評価を、クラスタを用いた並列実験により行なった。評価に利用したプログラムは、HPL のコードを変更して batched pivoting を実装したものである。数値計算カーネルとして後藤による BLAS ライブラリ[7] を、通信のために mpich ライブラリ 1.2.6[2] を利用した。本章の実験では行列サイズ n を 32,768, ブロックサイズ s_b を 256 とした。なお、全ての試行は HPL の残差チェックを通過した。

評価を行なった環境は 190 ノード Linux クラスタである。各ノードは Xeon プロセッサを 2 台持つ。実験では 2.4GHz のノードを最大 64 台、2.8GHz のノードを最大 96 台用いた。ノードあたり 1 プロセスのみ立ち上げ、またプロセッサ速度に関わらずデータを均等に割り当てた。ノードはツリー状に構成された Gigabit Ethernet で結合されている。ノード間の通信遅延は 55 ～ 75 マイクロ秒である (MPI で 1 word を通信して計測)。高遅延環境における評価を行なうために、ソフトウェアで遅延をエミュレートして実験を行なった。

まず、図 4 は遅延エミュレートを行なわないときの並列性能を示す。Batched pivoting におけるバッチステップ数 d を 4, 16, 64 とし、それぞれ Batched(4), Batched(16), Batched(64) で表す。Partial は元々の HPL を表す。公平な比較のために、batched pivoting の GFlops 値の計数には局所ガウス消去の計算量は含まれない。グラフから Batched は Partial と同様なスケーラビリティを達成していることが分かる。Batched は Partial よりも速度がやや低くなっているが、これは局所ガウス消去のコストのためと考えられる。最も差が大きい Batched(64) の低下は 7.5 ～ 15%である。

高遅延をエミュレートした場合の実験結果を図 5 に示す。追加された遅延は $2, 5, 10$ ミリ秒であり、各試行について全ノード間で均一とした。明らかに Partial は高遅延に弱く、遅延を追加した場合には全ての場合において Batched より低速となる。遅延を+10ms とすると、+0ms のときより 6.0 倍速度が低下する。一方 Batched ははるかに高い耐遅延性を持ち、d が大きいほど耐遅延性が高いことが分かる。Batched(64) の場合、+10ms における速度低下は 1.22 倍に抑えられており、このとき Partial よりも 4.8 倍高速である。

5. 計算精度評価

計算精度の評価を以下のような数値実験により行なった。実験には逐次プログラムを用い、batched, partial, threshold, pairwise の各ピボット選択手法の比較を行なった。実験に用いた行列サイズは 64 ～ 2048 であり、その各要素は [-1, 1] の範囲の乱数により生成されている。各サイズについて、異なる乱数系列を用いて 40 回試行を行なった。方程式 $Ax = b$ を解いた時の精度を、相対残差 $||A\tilde{x} - b||_\infty/(||A||_\infty||\tilde{x}||_\infty n\epsilon)$ により評価を行なった。ここで $\tilde{x}$ は計算された解、ϵ は機械イプシロン ($= 2^{-53}$) である。HPL の残差チェックの通過条件の一つは、この相対残差が $O(1)$ 以下であることである☆。

☆ HPL のログ表示には $||A\tilde{x} - b||_\infty/(||A||_\infty||\tilde{x}||_\infty\epsilon)$ と表示

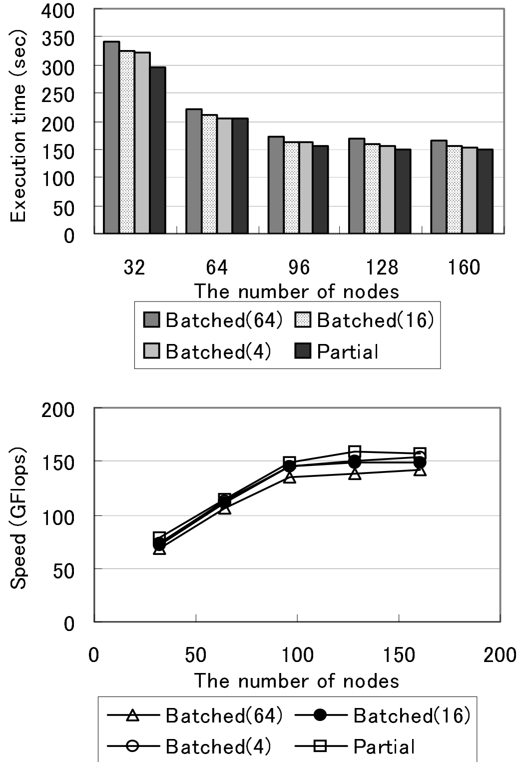

図 4　クラスタにおける並列性能 (n = 32768, s_b=256)。上図は実行時間を、下図は速度 (GFlops) を示す。プロセスグリッドサイズは $4 \times 8, 8 \times 8, 8 \times 12, 8 \times 16, 8 \times 20$

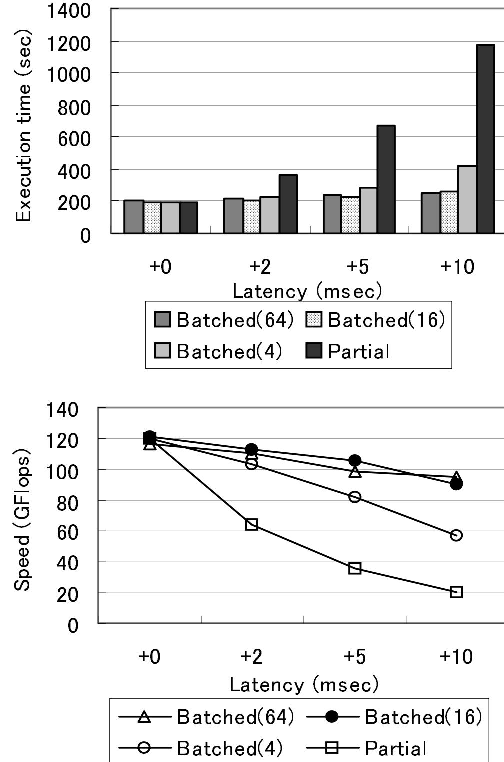

図 5　高遅延をエミュレートしたときの並列性能 (n = 32768, s_b=256,　64 ノード)。上図は実行時間を、下図は速度 (GFlops) を示す。プロセスグリッドサイズは 8×8

Batched pivoting をそのまま逐次処理で行なうと partial pivoting と同様の挙動になってしまう。並列処理の状況を再現するために以下のように変更して実験を行なった。行列をサイズ 16 のブロックに分割して各ブロックを仮想的にノードと見なす。つまり、各ブロックについてピボット候補リストを求める。

図 6 に結果を示す。グラフの横軸は行列サイズ n を、縦軸は相対残差を示す。ここでは 40 回の試行にわたる相対残差の平均値が示されている。Batched の括弧内の数値はバッチステップ数 d を、Threshold の括弧内の数値は 3.2 節で示したパラメータ τ を表す。Partial, Batched, Threshold, Pairwise, No(ピボット選択を行なわない場合) の中で、予想通り Partial が最も良い結果を示している。Batched と Threshold においては Partial よりも平均残差が大きくなっているものの、Batched(4) の平均残差は Partial の $1.09 \sim 1.55$ 倍に抑えられている。一方、d が大きいとき残差はより大きくなっており、耐遅延性と精度の間にトレードオフがあることが分かる。今回の実験においては Partial, Batched, Threshold において、相対残差 $\leq O(1)$ という条件を満たしている。

一方、Pairwise は全く異なる挙動を示している。$n = 64$ における残差は他手法と近いオーダーだが、n が増えるにつれ大きく悪化する。今回の実験より大きな n では、HPL の残差チェックを通らないことが予想される。Partial と Threshold は同期コストの影響を大きく受けることを考慮すると、batched pivoting のみが高い耐遅延性と良好な計算精度を両立していると言える。

6. 議　　論

Pairwise より精度の良い理由　前章で batched, pairwise, threhold が pairwise pivoting よりもはるかに良い精度を達成することを示した。Trefethen ら[13] によると、平均的数値安定性を達成するための条件の一つは、各ステップの更新フェーズにおいて、一つのピボット行が用いられることであ

されるが、ソースコード `HPL_pdtest.c` によるとこれは誤りであり、実際には n で除算された値が表示される

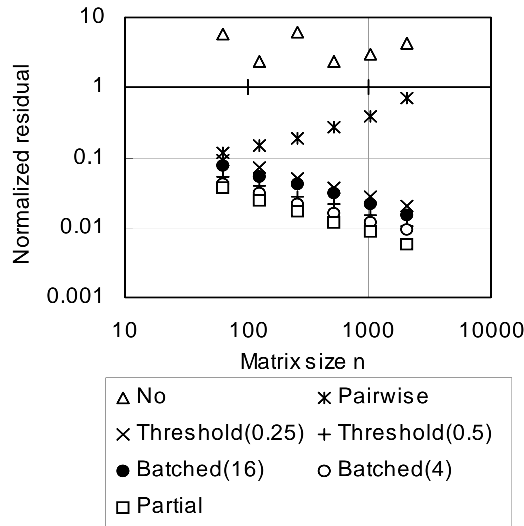

図 6 各ピボット選択手法による相対残差。各サイズあたり 40 個の乱数行列に対する結果の平均を示す

る。Pairwise pivoting はその条件を満たさず、他の手法は満たしている。

Batched が失敗する場合 Batched pivoting は乱数行列に対して良好な精度を達成する。しかし行列の性質によっては、partial pivoting が成功し batched pivoting が失敗する場合がある。列上にならんだ全ての $c \times d$ 部分行列のランクが d 以下であると失敗する。そのような場合はなんらかの回復手法が必要である。最も単純には、問題が起こった d 列についてのみ partial pivoting を行なうことが考えられる。

7. おわりに

高遅延に耐えることのできるピボット選択手法である、batched pivoting を提案した。複数ステップにまたがるピボット選択処理をまとめて行なうことにより、同期コストを大きく削減することができる。遅延が 10 ミリ秒の場合には、batched pivoting は partial pivoting よりも 4.8 倍高速となる ($d = 64, n = 32,768$)。その代償として、Partial pivoting に比べて浮動小数演算量の増加と計算精度の低下が起こるが、実験によりそれらが小さいことを示した。今回比較した手法の中で、batched pivoting は耐遅延性と良好な計算精度を両立する唯一の手法である。

今後は、本手法が失敗する場合における効率的な回復手法の設計と実装を行なう予定である。また、計算精度の理論的解析も行なっていきたい。今回の実験ではクラスタ上で遅延エミュレートを行なったが、実際のグリッド環境における評価も行ないたい。

参考文献

1) Folding@home. `http://folding.stanford.edu`.
2) MPICH - a portable MPI implementation. `http://www-unix.mcs.anl.gov/mpi/mpich/`.
3) SETI@home. `http://setiathome.ssl.berkeley.edu`.
4) TOP500 supercomputer sites. `http://www.top500.org/`.
5) Toshio Endo, Kenji Kaneda, Kenjiro Taura, and Akinori Yonezawa. High performance LU factorization for non-dedicated clusters. In *Proc. of CCGrid*, 2004.
6) I. Foster, J.Geisler, W. Gropp, N. Karonis, E.Lusk, G.Thiruvathukal, and S.Tuecke. Wide-area implementation of the message passing interface. *Parallel Computing*, 24(12):1735–1749, 1998.
7) Kazushige Goto. High-performance BLAS. `http://www.cs.utexas.edu/users/flame/goto/`.
8) Joel Malard. Threshold pivoting for dense LU factorization on distributed memory multiprocessors. In *Proc. of Supercomputing*, pages 600–607, 1991.
9) Motohiko Matsuda, Tomohiro Kudoh, and Yutaka Ishikawa. Evaluation of MPI implementations on grid-connected clusters using and emulated wan environment. In *Proc. of CCGrid*, 2003.
10) A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. `http://www.netlib.org/benchmark/hpl/`.
11) D. C. Sorensen. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Transactions on Computers*, c-34(3):274–278, 1985.
12) Kenjiro Taura, Kenji Kaneda, Toshio Endo, and Akinori Yonezawa. Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources. In *Proc. of PPoPP*, pages 216–229, 2003.
13) L. Trefethen and R. Schreiber. Average case stability of Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 11(3):335–360, 1990.