

lu2_mpi 解説

牧野

2019 年 5 月 21 日

目次

第 1 章	概要と変更記録	5
1.1	概要	5
1.2	変更記録	5
1.2.1	2019/5/18	5
1.2.2	2019/5/20	5
第 2 章	lu2_mpi のアルゴリズム	7
2.1	HPL のアルゴリズム	7
2.1.1	非ブロック・非並列アルゴリズム	7
2.1.2	非ブロック・並列アルゴリズム	7
2.1.3	ブロック・非並列アルゴリズム	8
2.1.4	ブロック・並列アルゴリズム	11
第 3 章	lu2_mpi 解説	13
3.1	main()	13
3.2	lu_mpi_blocked_lookahead()	13
3.3	column_decomposition_recursive_mpi()	16
3.3.1	process_right_part_mpi_withacol()	17
3.3.2	column_decomposition_mpi_with_transpose()	17
3.3.3	column_decomposition_mpi_transposed()	17
3.4	MP_construct_scalevector()	17
3.5	MP_process_diagonal()	17
3.6	MP_solve_triangle_for_unit_mat()	18
3.7	MP_process_lmat()	18
3.8	MP_calculate_ld_phase1()	18
3.9	MP_calculate_ld_phase2()	18
3.10	process_right_part_mpi_using_dls_phase1()	18

3.11	<code>process_right_part_mpi_using_dls_phase2()</code>	18
3.12	<code>MP_process_row_comm_init()</code>	18
3.13	<code>process_right_part_mpi_using_dls_concurrent()</code>	19
3.14	<code>MP_update_multiple_using_diagonal()</code>	19
3.15	<code>MP_store_diagonal_inverse()</code>	19
3.16	<code>backward_sub_blocked_mpi()</code>	19

第1章 概要と変更記録

1.1 概要

この文書では、lu2_mpi について、実装の考え方とソースコードの構成の解説を試みる。まず、HPL の基本的なアルゴリズムを解説する。次に、並列化、lu2_mpi での従来の HPL とは違う点についてまとめ、それからソースコード自体の解説を行う。

1.2 変更記録

1.2.1 2019/5/18

作成開始した。

1.2.2 2019/5/20

アルゴリズムとトップレベルルーチンの説明を書いた。

第2章 lu2_mpi のアルゴリズム

2.1 HPL のアルゴリズム

以下、非ブロック・非並列、非ブロック・並列、ブロック・非並列、ブロック・並列、の順番で概説する。

2.1.1 非ブロック・非並列アルゴリズム

HPL で行うことは、基本的には単なる LU 分解、すなわちガウスの消去法である。方程式

$$Ax = b \quad (2.1)$$

に対して、ブロック化しない、部分ピボットを行うガウスの消去法のアルゴリズムは以下ようになる。行列サイズを N とし、行列要素を a_{ij} , ($0 \leq i < N, 0 \leq j < N$) とし、添字を 0 から始めることにする。

1. $i = 0$ から $N - 2$ について以下のステップ 2,3 を繰り返す
2. $a_{i,i}$ から $a_{i,N-1}$ までで絶対値が最大のものが $a_{i,p}$ として、方程式の第 i 行と第 p 行を入れ換える (部分ピボット)
3. $k = i + 1$ から $N - 1$ の各行から i 行に $a_{k,i}/a_{i,i}$ をかけたものを引く。この結果、 $a_{i+1,i}$ から $a_{N-1,i}$ は全てゼロになる。(行列アップデート)
4. この結果、 A の左下半分は 0 になるので、あとは下から順番に x を求めていくことができる (後退代入)

2.1.2 非ブロック・並列アルゴリズム

ブロック化の話をする前に並列化の話をしておく。サイズ N の問題を $P \times Q$ の 2 次元グリッドに並べたプロセッサ (ノード) で並列計算することを考える。単純に、行列を $P \times Q$ の小行列に切って (それぞれサイズが $N/P \times N/Q$)、各ノードに割り当てたとする。以下面倒なので $P = Q$ とする。

各ノードはそれぞれ自分のデータを更新する処理をすると、最初の N/P 行列の変形が終わったあとは、ノード $(N_{0,0}$ から $N_{P-1,P-1}$ までであるとして) $N_{0,k}$ と $N_{l,0}$ と (l も k も 0 から $P - 1$ まで) は何もすることがなくなってしまう。計算が進むにつれて何もしないノードが増えるので、並列化効率が落ちる。

(練習問題: 漸近的な効率低下を評価せよ)

この効率低下を防ぐには、いわゆるサイクリック分割をすればよい。つまり、1次元方向だけ書くと、 $a_{0,0}, a_{0,1}, \dots, a_{0,N/P-1}$ をノード 0 が持つだけでなく、 $a_{0,0}, a_{0,P}, \dots, a_{0,N-P}$ をノード 0 が、 $a_{0,1}, a_{0,P+1}, \dots, a_{0,N-P+1}$ をノード 1 が、という具合にする。こうすると、P 行処理が進むと全ノードで 1 サイズが減る、ということになり、ほぼロードバランスしたままで最後まで処理が進む。

(練習問題: サイクリック分割にしても通信量や通信回数には変化がないことを示せ)

サイクリック分割では、計算手順は以下のようになると考えられる。

1. ピボット列を担当する全ノードで、まずローカルの最大値、次に全体の最大とその値をもつ行を決める。
2. その最大値になる行 (実際のピボット) をもつプロセッサは列方向にそのデータを放送する。元々のピボットの位置のプロセッサは交換先 (実際のピボット) をもつノードにデータを送る
3. ピボット列のデータを行列方向に放送する
4. 全ノードで、残っている行の値を更新する演算を行う。

この手順を素直に実装すると、ピボットを探してその処理をしている時に他のノードは遊んでしまうが、次のピボットを探すのは、次の行について更新処理が終わった時点でできるので、全体のアップデートが終わらなくてもピボットを探し、次の更新のために必要なデータの放送等を開始することができる。それを始めたあとで、残っている現在の更新を行うことで、ある程度通信を隠蔽することができる。

2.1.3 ブロック・非並列アルゴリズム

ここまでみてきた非ブロックアルゴリズムでは、1行ずつ消去するので、計算量の一番大きな演算は、各行についてその (残っている) 要素と (対角要素を規格化したピボット行の積を引く、という演算であり、これは形式的には

$$A_{i,new} = A_{i,old} - l_i^T u_i \quad (2.2)$$

と書ける。ここで、 A_i は係数行列の $a_{i,i}$ を含む右下の部分、 l_i^T はピボットの後の A_{i-1} の 1 列目 (第一要素除く) u_i はピボットの後の A_{i-1} の 1 行目 (第一要素除く) である。この操作は、残っている行列のサイズを n として、メモリアクセスを $2n^2$ (線型項は無視した) して演算も $2n^2$ するので、倍精度なら $B/F = 8$ を要求する。近年の高性能プロセッサでは B/F は 0.1 前後まで低下しているので、これでは全く性能がでない。このため、「複数行をまとめて」処理することで必要な B/F を小さくする。まず、2行 (2列) 同時消去する方法を考えてみよう。 $i, i+1$ 行の処理は以下のように書くことができる。

1. $a_{i,i}$ から $a_{i,N-1}$ までで絶対値が最大のものが $a_{i,p}$ として、方程式の第 i 行と第 p 行を入れ換える (部分ピボット)
2. 第 i 行を対角要素が 1 になるように規格化しておく。
3. 第 $i+1$ 列について、 $a_{j,i+1} \leftarrow a_{j,i+1} - a_{j,i}a_{i,i+1}$ の変形を行う。
4. 第 $i+1$ 行について、 $a_{i+1,j} \leftarrow a_{i+1,j} - a_{i+1,i}a_{i,j}$ の変形を行う。

5. $a_{i+1,i+1}$ から $a_{i+1,N-1}$ までで絶対値が最大のものが $a_{i+1,q}$ として、方程式の第 $i+1$ 行と第 q 行を入れ換える
6. 第 $i+1$ 行を対角要素が 1 になるように規格化しておく。
7. $i+2$ より右下の要素 $a_{k,l}$ について、2 行分の処理 $a_{k,l} \leftarrow a_{k,l} - (a_{k,i}a_{i,l} + a_{k,i+1}a_{i+1,l})$ を行う

最後の処理は、 n 行 2 列の行列と 2 行 n 列の行列の積になるので、非ブロックアルゴリズムに比べてメモリアクセスが半分になる。

この考え方を 3 行以上に一般化することは可能であり、それによって主要な操作である行列のアップデートで発生する操作をサイズの大きな行列行列積にしていけることができる。ここで、ブロック化するサイズを b としよう。

そうすると、行列アップデートのところでのメモリアクセスは非ブロック化アルゴリズムの $1/b$ になる。従って、メモリアクセスの量は全体で $O(N^3/b)$ になる。一方、ブロック化のために b 列分の処理をするところは普通にメモリアクセスするので、その部分では全体では $O(N^2b)$ のメモリアクセスが発生する。つまり、このブロック化では、 $b \sim \sqrt{N}$ のところにメモリアクセスが最小になる点があり、それ以上メモリアクセスを減らすことができないことがわかる。

この限界を超えてメモリアクセスを減らすことを可能にしたのが、Gustavson[1] である。考え方に別に難しいところはなく、単にこのブロッキングを再帰的に適用して可能な限り操作を行列行列積に置き換えていく、というものである。つまり、 b 列の変形を非ブロック化アルゴリズムでするわけではなく、 $b/2$ 列変形したら後ろ半分のアップデートは行列行列積で行う、さらに、 $b/2$ 列変形についても (これは 2 回ある)、 $b/4$ 列変形して後ろ半分をは行列行列積で行う、というのを繰り返していくわけである。

この方法を適用することで、図 2.1 の $D+L$ の部分の処理はブロック化できているが、それだけでは、 U の部分の処理がブロック化されない。

して、メモリアクセスを本当に減らすためには、 U の部分の処理もブロック化する必要がある。

上の、2 行だけの場合について、 U の処理を後回しにする手順を考えてみる。

1. $a_{i,i}$ から $a_{i,N-1}$ までで絶対値が最大のものが $a_{i,p}$ として、第 i 行と第 p 行の $i, i+1$ 列目を入れ換える。
2. 第 i 行の $i+1$ 列目は対角要素が 1 になるのに対応した規格化しておく。要素 $a_{i,i}$ はあとでまた使うので記憶しておく。
3. 第 $i+1$ 列について、 $a_{j,i+1} \leftarrow a_{j,i+1} - a_{j,i}a_{i,i+1}$ の変形を行う。
4. $a_{i+1,i+1}$ から $a_{i+1,N-1}$ までで絶対値が最大のものが $a_{i+1,q}$ として、 $a_{i+1,i+1}$ と $a_{q,i+1}$ を入れ換える。
5. U, R の部分について、 i 行と p 行、 $i+1$ 行と q 行の入れ換えを行う。
6. U の $i+1$ 行の部分について、残っていた
7. 第 $i+1$ 行について、 $a_{i+1,j} \leftarrow a_{i+1,j} - a_{i+1,i}a_{i,j}$ の変形を行う。

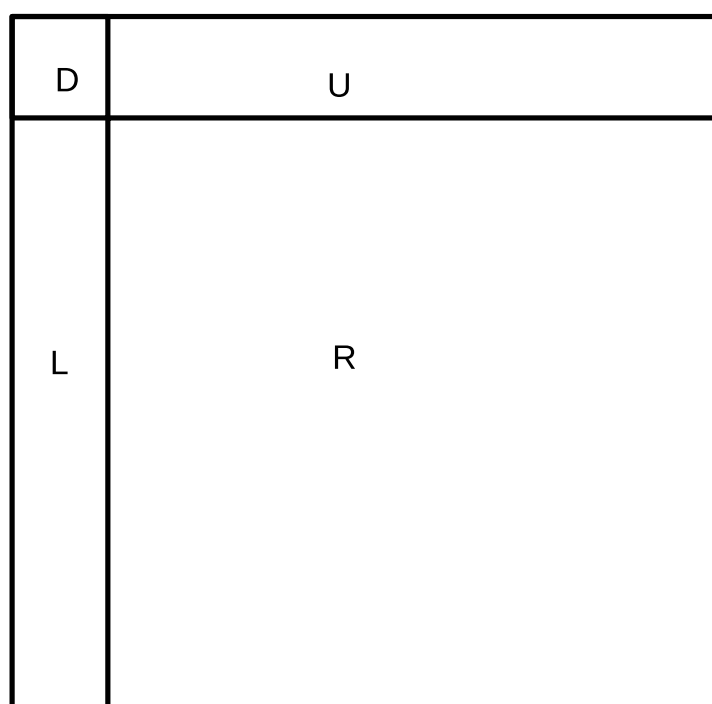


図 2.1: ブロック化アルゴリズムでの行列 (既に処理が終わった部分は書いてない。現在処理中のブロックとその右のみを書く)

8. $i+2$ より右下の要素 $a_{k,l}$ について、2 行分の処理 $a_{k,l} \leftarrow a_{k,l} - (a_{k,i}a_{i,l} + a_{k,i+1}a_{i+1,l})$ を行う

ここでやっていることをブロックサイズ b に拡張して、形式的に書いておこう。ややこしいので、まずピボットを無視して考える。 D を $D = D_L D_U$ と LU 分解したとする。これは、 $D_U = D_L^{-1} D$ ということなので、 U のところも $U \leftarrow D_L^{-1} U$ と変形すればよい。形式的には L のところも $L \leftarrow L D_U^{-1}$ と変形できる。しかし、 L のほうはピボットを選ぶ前にその行の変形が完了している必要があるので、 D だけを処理するわけにはいかない。一方、 U のほうは、その変形は全部ピボット処理の後にやってもかまわない。

なお、 $U \leftarrow D_L^{-1} U$ は「多数のベクトルについて 3 角行列を解く」という操作であり、3 角行列を解く操作自体を再帰的ブロック化する等で行列積に置き換えられる。BLAS では、これのための DTRSM というルーチンが用意されている。しかし、lu2_mpi では、「 D_L^{-1} を実際に計算し、しかもこれを U に掛けるのではなく L に掛ける」という文献にはないアルゴリズムを使っている。

2.1.4 ブロック・並列アルゴリズム

これは基本的にはブロックサイズ b でサイクリック分割する、というだけである。通信と計算のオーバーラップ、メモリアクセスの最小化等についてはコードを解説しながらみていく。

基本的にはいわゆる 1 段のルックアヘッドをしており、最初のパネル分解をしたあと、 L, U を放送するが、次のパネルを担当するノードはそのパネルだけ処理して L の通信を始め、それから残りの行列のアップデートをする。 U については、 U をブロックサイズ b で分割し、行列アップデートとその次の部分の通信を並行して処理することも可能な実装になっている。

ただし、 L の非同期通信のところはちゃんと考えてないところがあるかも。

第3章 lu2_mpi 解読

書いた人も記憶がなくてなぜ動いているのかわからないので、コードを眺めていく。main から階層的にみていこう。

3.1 main()

main 関数で最初に数値をいじっているところは、lu2_mpi.c の L4103 あたり (以下 2019/5/19 前後のソースでの行番号)

```
MP_randomsetmat(controls.nncol, controls.nnrow, a,&parms,
                &controls,1,b0);
```

である。これは、HPL で解くべきランダム行列の初期設定をする。a, b0 が係数行列と値ベクトルになっていると思われる。

```
lu_mpi_blocked_lookahead(controls.nnrow, controls.nncol, a,
                          acol,acol2,dinv,arow,arow2,
                          b,&controls, &parms);
```

が計算本体である。

3.2 lu_mpi_blocked_lookahead()

この関数の現在の制御構造は大体こんな感じになっている。

```
column_decomposition_recursive_mpi();
MPI_Bcast(pv...);
MP_construct_scalevector();
MPI_Bcast(scale...);
MP_process_diagonal();
int nrows=MP_process_lmat();
MP_calculate_ld();
for(i=0;i<n;i+=nb){
    generate_src_and_dst_lists();
```

```

    if ((i+nb < n) && havec){
        process_right_part_mpi_using_dls_phase1();
        process_right_part_mpi_using_dls_phase2();
        column_decomposition_recursive_mpi(ii, nb, nnrow, nncol, a, acp2,
                                           b, pvp2, controls, parms);

        MP_construct_scalevector();
        cfirst=c2;
    }else{
        cfirst=c1;
    }
    if (i+nb < n){
        MP_process_diagonal_phase1();
    }
    nrow=MP_process_row_comm_init();
    process_right_part_mpi_using_dls_concurrent();
    check_and_continue_rcomm_transfer(&rcomm,1);
    MP_update_multiple_using_diagonal();
    MP_store_diagonal_inverse();
    if (i+nb < n){
        MP_calculate_ld_phase1();
        MP_calculate_ld_phase2();
    }
}
backward_sub_blocked_mpi(nnrow, nncol, a,nb, dinv, b, controls, parms);

```

もうちょっと綺麗に書ける気がするが、現在のところブロック化した上でルックアヘッドもする関係で、最初のブロックの縦方向の処理 (パネル分解) をブロックについてのループに入る前に行う。これが

```
column_decomposition_recursive_mpi();
```

である。それから、ピボット (行交換) の情報を、行方向に放送する (列方向では、パネル分解を担当した列で既にコピーされている。さらに、行交換に相当する処理でピボットベクトルと一緒に使うスケールベクトルを構成し (MP_construct_scalevector) さらに行方向に放送する。

次の、MP_process_diagonal(); では D_U の逆行列を求めている (はずである)。このため、 D をもつプロセスで、それをローカルの配列にコピーしてから MP_solve_triangle_for_unit_mat() を呼んでいる。ここでは求まった逆行列を行方向に放送している。その次の MP_process_lmat では、 L を行方向に放送している。最初のところは非同期にしてもしょうがないので、ここは放送である。次の MP_calculate_ld は MP_calculate_ld_phase1 MP_calculate_ld_phase2 を順番に呼ぶ。MP_calculate_ld_phase1 は、 D_U^{-1} を列方向に放送する。MP_calculate_ld_phase2 は、 LD_U^{-1} を計算する。

このあと、本体のループに入る。本体のループでは、ルックアヘッドを行うので、自分のところに次のパネル分解すべきブロックがある列のノードでは、

```
process_right_part_mpi_using_dls_phase1();
```

```
process_right_part_mpi_using_dls_phase2();
```

で、パネル分解すべきブロックだけのアップデートをしたあと、

```
column_decomposition_recursive_mpi();
MP_construct_scalevector();
```

でパネル分解をすませておく。さらに

```
MP_process_diagonal_phase1();
```

で D_u^{-1} を求める。これは実際には対角ブロックをもつノードでだけ行う。次に

```
nrows=MP_process_row_comm_init();
```

で、行内の通信を始める。

```
process_right_part_mpi_using_dls_concurrent();
```

がアップデートの本体である。内容はあとで。

```
check_and_continue_rcomm_transfer(&rcomm,1);
```

は、行方向の通信の終了待ちである (多分)

```
MP_update_multiple_using_diagonal()
```

は、 U をアップデートしている。これは、行列のアップデートにはアップデートする前の U を使うが、後退代入の時には U の最終のものがいるのでここで求めているような気がする。なんか違うかも。

```
MP_store_diagonal_inverse(nnrow, nb, dinv, parms, controls, i, aip);
```

は、 D_U^{-1} のストア

```
MP_calculate_ld_phase1();
MP_calculate_ld_phase2();
```

は、ループに入る前と同じで、 D_U^{-1} の列方向の放送と LD_U^{-1} の計算である。

```
backward_sub_blocked_mpi()
```

は、名前の通り後退代入であり、ここまでで処理が終わる。以下では、重要そうな関数についてもうちょっとみていこう。

3.3 column_decomposition_recursive_mpi()

ここでは、名前の通り再帰的ブロッキングアルゴリズムでパネル分解を行う。コードでは

```
if (nb <= 8){
    column_decomposition_mpi_with_transpose();
}else{
    b が8より大きい時の処理
}
```

となっていることにまず注意してほしい。これは、現在のコードでは $b = 8$ になるところで行列を転置して、処理ルーチンを切替えるからである。通常使われている HPL では、行列は column-major (列優先) で格納される。列優先では、 a_{00} の次のアドレスには a_{10} がくる。これはピボットサーチでは連続アクセスになるので有利だが、行交換では不連続アクセスが多数発生する。それでも、ブロック化していれば行交換のほうが不連続アクセスによる性能低下が小さいために通常列優先を使う。しかし、メモリバンド幅が極端に小さいマシンの場合、行交換が不連続アクセスであることのオーバーヘッドは致命的になるので、lu2_mpi では、再帰的パネル分解の途中で転置する、というアルゴリズム [2] を採用した。

以下、 b が8より大きい時の処理をみていく。

```
column_decomposition_recursive_mpi(ifirst, nb/2, nnrow, nncol,
                                   a, acol, b, pv, controls, parms);
```

では、ブロックを2分割して左側を処理する。さらに

```
process_right_part_mpi_withacol(ifirst, nb/2, nb/2, nnrow, nncol, a,
                                acol, b, pv, controls, parms, 1);
```

で、その結果を使った右半分のアップデートを行い、

```
column_decomposition_recursive_mpi(ifirst+nb/2, nb/2, nnrow, nncol,
                                   a, acol, b, pv+nb/2, controls, parms);
```

で右側半分をさらに再帰的に処理していく。

```
MP_swap_rows_blocked(nnrow, nncol, a, parms, controls, i, pv[ii],
                    c1, c2);
MP_scale_row_blocked(nnrow, nncol, a, parms, controls, i, c1, c2, 1);
```

では、右側の処理で必要になった左側の行交換とスケールリングを行う。2.1.3 節の議論から、ブロック化してできる L 行列は全体として正しく行列交換されている必要がある。このため、再帰処理では、右半分の処理結果を左半分に反映させる必要がある。

3.3.1 process_right_part_mpi_withacol()

処理の構造は以下のようになる。

```
MP_process_diagonal();
for (ii=0;ii<nb;ii++){
    MP_swap_rows_blocked();
    MP_scale_row_blocked();
}
MP_update_multiple_using_diagonal();
MP_update_multiple_blocked_global_withacol();
```

MP_process_diagonal() では、左半分について D_U の逆行列を求めている。それから、MP_swap_rows_blocked(); MP_scale_row_blocked(); では右半分の行交換とスケーリングを行う。MP_update_multiple_using_diagonal() では U のアップデートを行い、MP_update_multiple_blocked_global_withacol() では本当に dgemm 呼んでアップデートを行っている。

3.3.2 column_decomposition_mpi_with_transpose()

これは、行列を転置して

```
column_decomposition_mpi_transposed(ifirst,nb,nnrow,nncol,
                                     awork, b, pv,controls,parms);
```

を呼んで、また転置するだけである。

3.3.3 column_decomposition_mpi_transposed()

これは、転置した行列 (細いパネル) のパネル分解をするルーチンだが、単純に 1 行ずつ変形している。ここでも再帰することも考えられるが、既にキャッシュにのっていて、列方向には SIMD オペレーションができるので、あまり大きな性能向上はなかったためであろう。

3.4 MP_construct_scalevector()

規格化のため、カレントブロックの対角要素の逆数を作る。

3.5 MP_process_diagonal()

対角ブロックをもつプロセスでは、MP_solve_triangle_for_unit_mat を使って D_U^{-1} を求める。引数の値によっては行方向に放送する。これカレント行でないところでも放送しているけど無駄な気がする。

3.6 MP_solve_triangle_for_unit_mat()

3 角行列を再帰的に解く。

3.7 MP_process_lmat()

L を行方向に放送する。ルックアヘッドありのルーチンでは、最初だけ使われる。その後は `MP_process_row_comm_init()` による非同期な通信が使われる。

3.8 MP_calculate_ld_phase1()

L (U でもだが) と掛けることのできる D_U^{-1} を列方向に放送する。

3.9 MP_calculate_ld_phase2()

L と D_U^{-1} を掛ける。

3.10 process_right_part_mpi_using_dls_phase1()

アップデート部分についての行交換とスケーリングを行う。 `vcommscheme=1` だと古い実装になる。

`global_swap_using_src_and_dest()` が実装の本体で、通信回数の最適化をしている模様。

3.11 process_right_part_mpi_using_dls_phase2()

`MP_update_using_lu()` が本体。DGEMM 呼んでアップデートを行う。これは基本的に DGEMM を 1 ブロック分 (縦には長い) で呼び出すようになっている。

3.12 MP_process_row_comm_init()

現在パネルをもつノードは L の通信を始める。但し、ルックアヘッドをしているので、ここで送られる L はこれからのアップデートに使う L ではなく、次のブロックで使う L である。

通信は `register_to_rcomm` と `start_rcomm_transfer()` で始められる。これらは、送る/受け取るべきメッセージの状態を、INITIALIZED, RECEIVING, RECEIVED, SENDING, SENT の 5 状態で管理する。`check_and_continue_rcomm_transfer()` は、どこまで終わったかチェックし、`blockmode` が 0 でなければ全て終了まで待つ。

現在の lu2_mpi の実装では、行方向の通信は 1 次元リングモードであり、カレントブロックをもつ列から順番に送っている。これは、MPI_Broadcast を信用していないからである。あと、非同期で動くので計算とオーバーラップさせやすい。レイテンシはともかくバンド幅はどうせ同じなので、特に MPI_Broadcast に直す必要はないかもしれない。

3.13 process_right_part_mpi_using_dls_concurrent()

CONCURRENT_UCOMM を定義すると、列方向の通信も非同期でやろうとするが、これは少なくとも GRAPE_DR のホストでは動作が不安定になった。以下列方向の通信は同期を仮定する。DO_RCOMM_LAST も定義しない。

これは、 U の幅で ninc (現在の設定は b に等しい) 毎に、行列アップデートを行う。従って、 L は縦に長い、 U は正方形である。この単位 (大きくすることも可能) で、行交換・スケーリングと実際のアップデートを、現在の設定 (CONCURRENT_UCOMM 未定義) では順番に行う。ハードウェアがまともなら非同期で行える。

GRAPE_DR の場合、 L をデバイスメモリにおいて、 U を送って LU を返す、という実装になっており、 L は使い回して、1 度しか送らないようになっていた。

3.14 MP_update_multiple_using_diagonal()

U をアップデートしている。

3.15 MP_store_diagonal_inverse()

D_U^{-1} のストア。

3.16 backward_sub_blocked_mpi()

単なる後退代入だが、色々頑張って並列化・ブロック化はしてある。

関連図書

- [1] Fred G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41:7367–756, 1997.
- [2] Junichiro Makino, Hiroshi Daisaka, Toshiyuki Fukushige, Yutaka Sugawara, Mary Inaba, and Kei Hiraki. The performance of grape-dr for dense matrix operations. *Procedia Computer Science*, 4:888 – 897, 2011. Proceedings of the International Conference on Computational Science, {ICCS} 2011.