社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2007－ARC－172 (3)
2007－HPC－109 (3)
2007／3／1

疎行列ソルバにおける非同期反復解法の性能評価

藤 井 昭 宏†

通信遅延が多く含まれる環境や動的に負荷状態が変化する環境において非同期反復解法が注目されている．本研究では，不規則疎行列線形ソルバを非同期反復解法に基づいて作成し，性能評価を行う．定常反復解法を非同期にすることにより，プロセッサ間で解の最新の値が交換されることが保障できなくなり，収束までに要する反復回数は増加する．一方で，更新された解データの受信ための同期待ち時間がないためCPUの利用効率は高くなる．また，非同期にすることにより，負荷分散を全体の同期をせずに行えるメリットもある．本稿では同期反復解法を非同期化し，動的負荷分散を実現し，これらのオーバーヘッドと性能向上に関して，予備的な実験と評価を行う．

Evaluation of Asyncronous Iterative Method for Sparse Matrix Solver

Akihiro Fujii†

Asynchronous iterative solver is known as one of the best solvers on the computing environment with large communication latency or dynamic change of load. In this paper, we have implemented the asynchronous iterative solver for sparse matrix linear problems, and evaluated it. Asynchronism may cause some processors receive the values of the solution vector behind time. On the other hand, the time of waiting for incoming messages is reduced, and computation is done more efficiently. In addition, asynchronism helps the dynamic load balancing without synchronization among all processor elements. In this paper, we have evaluated these features of asynchronous iterative solvers by implementing the solver for the sparse matrix linear problems.

1. はじめに

通信遅延が多く含まれるような環境や動的に負荷状態が変化するような計算環境において非同期反復解法[1),4),5)]が注目されている．非同期反復解法は，各プロセッサ間での依存関係を切断することにより同期待ち時間を削減し，CPUの利用効率を上げ，性能の向上を目指す手法である．各PEの負荷情報を非同期に交換することにより，非同期に反復的な負荷分散も実現できることが報告されている．

ただ，依存関係が複数のPEに及ぶような疎行列演算に対応した事例は少なく，かつすべてMPI上で実装されているものは存在しない．そこで本研究では，MPI上に非同期反復解法を実現し，疎行列ソルバとして負荷分散も含めた性能評価を行った．

2節に非同期反復解法の概略を示し，3節にMPI上でのアルゴリズムと実装について示した．4節，5節において数値実験を行い，評価を行った．

† 工学院大学/科学技術振興機構 CREST
Kogakuin University/CREST, JST

2. 非同期反復解法に基づく疎行列ソルバ

本節では，対象とする問題，反復解法の並列化手法とそのデータ構造を説明し，本研究で扱う非同期反復解法の概略について説明する．

問題は行列Aとベクトルbが与えられたときに，以下の式を満たすベクトルxを求めることである．

$$Ax = b$$

行列Aのサイズを$N \times N$とし，二つのベクトルx, bはN個の要素を持つこととする．

次に，並列化の方法，各PEが保持している行列データや通信テーブルについて説明する．本研究では，疎行列ソルバの並列化は行列をブロック行分割することにより実現する．ここでPE iが担当する行数をN_iとする．このとき$N = \sum_i N_i$となる．また，このN_i行ブロック部分だけの行列ベクトル積を計算するときにアクセスされるベクトルの要素数をNP_iと定義する．このNP_iはN_i行ブロック部分の非ゼロ要素を含む列数に対応する．各PEiが担当するブロック行の方程式を

$$A_i x_i = b_i \quad (1)$$

とすると，行列A_iのサイズは$N_i \times NP_i$となり，ベク

トル x_i は NP_i 個，ベクトル b は N_i 個の要素を持つこととなる．ベクトル x_i の NP_i 個の要素のうち N_i 個の要素は PE i で値を更新できるが，残りの $NP_i - N_i$ 個の要素の値は他の PE で更新された値を通信により取得しなければならない．このため通信テーブルを利用するが，そのテーブルには，ベクトル x_i のどの要素の値をどの PE に送出すればいいか，またどの要素の値をどの PE から受け取ればいいか，が記録されている．

通常の同期型反復解法では，式 (1) を，各 PE で独立に解いた後に通信を行い同期を行う．即ち，全体で足並みをそろえて，反復回数を増やしていく．一方，非同期反復解法では，各 PE は独立に式 (1) を解いた後，通信を行うが，その時点で受信できない通信データを同期待ちをせず，次の反復に入る．非同期にすることによる利点は，同期待ち時間が削減されることだけでなく，隣接 PE と負荷情報を交換することにより動的負荷分散が全体の同期なしに実現できる点も挙げられる．

Algorithm1 に，動的負荷分散を実現する非同期反復解法における PEi が行う処理の概略が書かれてる．反復を TRY_LB_TIMES 回行うごとに，隣接との負荷情報を比較して疎行列のブロック行を隣接 PE との間で再分散を行うかどうか判断している．アルゴリズム中の *ld* には PE i の負荷情報が入っているとし，「Exchange(*ld*, *ld_array*)」は隣接 PE と負荷情報を交換し，*ld_array* に届いている負荷情報を記録する関数である．この関数も他の関数と同様に，同期待ちはせず，受信済みの負荷情報のみを処理する．「x_i=Solve(A_i,x_i,b_i)」は，ガウスザイデル法などの定常反復解法を通信なしに 1 反復行い更新された解が x_i に書き戻されることを表す．「AsyncSENDRECV(x_i)」は通信テーブル基づいて解 x_i を送受信する関数である．この関数は呼ばれた時点で受信されたデータのみを解ベクトル x_i に反映する．

Algorithm 1 非同期反復解法

```
count = 0
TRY_LB_TIMES = 20
repeat
  Exchange(ld, ld_array)
  if ブロック行が転送されてきた then
    ブロック行受け入れ処理
  end if
  if count = 0 then
    if 隣接 PE より負荷が高い then
      その隣接ＰＥへ行列の一部を移動
    end if
    count = TRY_LB_TIMES
  else
    count = count-1
  end if
  x_i = Solve(A_i,x_i,b_i)
  AsyncSENDRECV(x_i)
  if 収束条件を満たす then
    終了処理
  end if
until 収束条件を満たす
```

2.1 動的負荷分散

本研究では，動的負荷分散を Jacques ら[1] が提案している手法に基づいて実装した．

その手法では，一定反復回数ごとに，連番の PE 間でのみ負荷情報の交換を行い，連番の PE と比較して閾値以上に自分の PE の負荷が高い状態になっていれば，負荷の移動を行う手法である．疎行列ソルバに対する場合，PE i は PE $i-1$ と PE $i+1$ との間で，負荷情報の交換を行い，必要に応じて，ブロック行を転送することになる．このように連番の PE 間のみでブロック行の移動をすることで，行列全体を考えたときの行番号のオーダリングは変更されないというメリットがある．

各 PE の負荷の指標として，1 反復あたりにかかる計算時間や，各 PE の残差ノルムなどが考えられるが，本研究では各 PE のブロック行の行数を負荷の指標として利用した．

2.2 収束検知と終了処理

非同期反復解法では，全体で同期をとることは皆無であるため，収束状態の条件を新しく定義する必要がある．本研究では，すべて PE の残差ノルムが閾値以下になったときに収束したと定義する．Jacques ら[2] は集中管理方式に基づかない，収束検知を提案しているが，本研究では実装の簡略化のため収束の検知は集中管理方式で実現した．以下に手順を示す．

(1) 各 PE で，残差の無限ノルムを算出し，それが閾値以下になったら，PE 0 に収束を伝える信号を送る．但し，反復計算は継続する．

(2) PE 0 では，全 PE から収束したというメッセージが到達すると，全 PE に確認のメッセージを送出する．

(3) 各 PE が収束確認のメッセージを受け取ったら，収束状態かどうか確認し再度，収束状態を PE 0 に送る．

(4) すべての PE から収束確認が取れたら，全体に収束に到達したことを意味するメッセージを送る．一部の PE が収束状態にない場合は，終了処理を中断し (1) に戻る．

(5) 収束に到達したメッセージを受け取ると，各 PE は終了処理を実行する．

終了処理では，通信状態や，行列の状態などを確認し，プログラムが終了できる状態になるまで，反復を続ける．各 PE が終了できる状態なったら，収束の検

知と同様の処理を行い，全体で同時に終了する．

3. MPIによる非同期反復解法の詳細

この節では，本研究で実装した非同期反復解法のアルゴリズムについて記述する．非同期通信やPE間での部分行列の移動についてはサブセクションで取りあげる．

まずロックの必要性とその範囲について考える．行ブロックが移動すると，送信側と受信側のPEとそのPEの通信テーブルでつながっているPE（隣接PE）の保持しているデータに影響がでてくる．そのため，送信側と受信側のPEは行ブロックを移動する前に，隣接PEで別の行ブロック移動処理が実行されていないことを保障する必要がある．そこで行ブロックを移動するまえに，送信側と受信側のPEで隣接PEをロックする．隣接PEのすべてが送信側PE，もしくは受信側PEにロックされたときのみ，行列移動処理に入り，ロックが成功しなかった場合は行列移動処理を中断する．この仕組みに基づいたアルゴリズムをAlgorithm 2,3,4に示す．変数の内容は以下のようになる．

lb_done 行列転送処理中かどうかを表すフラグ

sender_flag 送信側PEであるか，受信側PEであるかを表すフラグ

locked_by_neighbor ロックされたときに，ロックをしたPE番号を記憶する整数変数．ロックされていないときは-1.

mat_recv_decision ブロック行の受信処理を複数段階に分けるための変数．0〜2の値を取りうる．

mat_send_decision ブロック行の送信処理を複数段階に分けるための変数．0〜2の値を取りうる．

隣接PEとしてロックリクエストを受け取る関数が「Lock_neighbors_reception(locked_by_neighbor)」である．ロックされていなければ，そのPE番号をlocked_by_neighbor変数に記録する．その後，locked_by_neighbor変数のデータをロックリクエストを出したPEへ送信する．これにより，ロックリクエストを送信したPEがロックが成功したかどうか，どのPEにロックされていたのかを知ることができる．次に「Comm_modify_reception()」関数について説明する．この関数は隣接PE上でブロック行が移動したときに，データの変更を受け付ける関数である．データの変更は転送処理をしている送信側PEもしくは受信側PEから送信される．

受信側の処理は，「lock_request_reception()」によって他のPEから行列転送のリクエストを受け付ける．locked_by_neighborを参照し，他のPEにロックされていない場合は，locked_by_neighborに自分のPE番号を記録し，lb_done=trueにして送信側PEに受信動作に入れることをを返信する．受信処理では「lock_neighbors()」により隣接PEにロックリクエストを送信する．隣接PEから返信があり，ロックが成功したかどうか分かるようになるとmat_recv_decisionが1以上となる．mat_recv_decisionが0の間は反復を継続し，何度もlock_neighbors()が呼び出されることになる．送信側PEと受信側PEで隣接PEすべてロックできた場合，ブロック行の転送処理「Matrix_transition()」に入る．その後，「Unlocking_neighboring_PEs()」により，ロックした隣接PEをアンロックする．アンロックリクエストはすべてのアンロックが完了するまで，この関数は終了しない．行列の移動に関連する処理を終了するため，lb_doneフラグをfalseにして，locked_by_neighborを-1に，countをTRY_LB_TIMESに戻す．

送信側の処理は，mat_send_decisionという変数によっていくつかの段階に分けて実行される．初期値は0となっている．「Decision()」を呼び出し，負荷情報と自分がロックされていないことを確認して負荷分散をするかどうか判断し，負荷分散を実行する場合は，lb_doneとsender_flagをtrueにセットする．またlocked_by_neighborは自分のPE番号でロックする．その後，mat_send_decisionを1にセットし，次の段階に入る．この段階で，「Lock_request()」と「Lock_neighbors()」を呼び出す．これにより，受信側PEをロックし，隣接PEをロックする．それらが成功したら，mat_send_decisionを2にセットされる．ロックリクエストが返ってきて結果が判明するまでmat_send_decisionは1のままで，反復計算をつづける．mat_send_decision=2の段階で，ブロック行の移動処理を行う．最後のIF文は，mat_send_decision=1以外のときにアンロックリクエストを出して，様々な行列転送に関わったパラーメターを初期化する．mat_send_decision=1の場合はロックリクエストの反復計算をしながら返信を待てるように，パラメータの初期化は行わない．

3.1 非同期の送信と受信

行列転送処理以外の処理，例えばロックリクエストの送受信や，通信テーブルに基く解情報の送受信などは以下の図1ようなMPIの操作で実装した．送信側は，以前に送信したデータがMPI_TEST()によって終了できれば，送信処理に入るが，そうでない場合は送信処理には入らず関数を終了することになる．

3.2 行ブロックの移動処理

行ブロックの移動処理については我々が作成した行列再分散ルーチン[3]にもとづいて実装した．

本研究では，連番のPE間でしか行ブロックの移動が発生しないと仮定しているので，送信側と受信側の2PEでのみ同期し，行ブロックを転送して通信テーブルを修正するという処理になっている．通信テーブルの修正は送受信を行うPEの隣接PEにも及ぶ．その

```
!C 送信側
CALL MPI_TEST(req, flag, status, ierr)
IF(flag) THEN
  CALL MPI_ISSEND(sendbuf, size, buf_type, PE_rank, TAG, COMM, req, ierr)
END IF

!C 受信側
CALL MPI_IPROBE(PE_rank, TAG, COMM, flag, status, ierr)
IF(flag) THEN
  CALL MPI_IRECV(recvbuf, size, buf_type, PE_rank, TAG, COMM, req, ierr)
  CALL MPI_WAIT(req, status, ierr)
END IF
```

図 1　MPI での非同期の送受信

Algorithm 2 MPI 版：非同期反復解法

```
count = 0, TRY_LB_TIMES = 20
lb_done = false, sender_flag = false
locked_by_neighbor = -1
mat_recv_decision = 0
mat_send_decision = 0
repeat
  Exchange(ld, ld_array)
  Lock_neighbors_reception(locked_by_neighbor)
  Comm_modify_reception()
  /* 受信側の処理 */
  Algorithm 3 参照
  if count = 0 then
    /* 送信側の処理 */
    Algorithm 4 参照
  else
    count = count - 1
  end if
  x_i = Solve(A_i,x_i,b_i)
  AsyncSENDRECV(x_i)
  if 収束条件を満たす then
    終了処理
  end if
until 収束条件を満たす
```

Algorithm 3 MPI 版：受信側の処理

```
Lock_request_reception(lb_done, locked_by_neighbor)
if lb_done=true AND sender_flag = false then
  if mat_recv_decision=0 then
    Lock_neighbors() を実行
    ロックが失敗したら mat_recv_decision=1
    ロックが成功したら mat_recv_decision=2
  end if
  if mat_recv_decision > 0 then
    if mat_recv_decision = 2 then
      Matrix_transition()
    end if
    Unlocking_neighboring_PEs()
    locked_by_neihbor=-1
    lb_done=false
    mat_recv_decision = 0
    count = TRY_LB_TIMES
  end if
end if
```

隣接 PE 上の通信テーブルの修正が終わるまで，このブロック行移動処理は終了しない．

4. 数値実験

非同期反復解法のオーバーヘッドや同期的反復解法と比較したときの利点をみるため，直方体領域の 3 次元ポアソン方程式を題材に以下の 2 つの数値実験を行った．

(1)　各 PE で問題サイズを $25 \times 25 \times 25$ に固定して分散し，PE 数と問題サイズで比例させた場合．

(2)　全体問題サイズを $60 \times 60 \times 90$ に固定し，PE 数を変化させて解いた場合．但し，問題は，2×2×3 の 12PE に分散して与えられるとする．

反復解法としては，それぞれの PE でガウスザイデル法を実行し，反復ごとに解の値を更新する通信をするカオティックガウスザイデル法を選択した．収束判定条件は，残差の無限ノルムが 10^{-2} 以下となるように設定した．それぞれの解法で最終的な解の収束状態を確認するため，収束後に相対残差を同期的に通信をして求めている．負荷分散指標は，自分の担当しているブロック行の行数で与えている．実験環境は，IBM xSeries サーバ (Xeon 2.8GHz × 2, 主記憶 1GB) を 1000Base-T により 12 ノード接続したクラスタを利用した．

実験 1　はじめから全 PE の行数はバランスが取れているため，PE 間で行ブロックの移動は起きない．

Algorithm 4 MPI 版：送信側の処理

```
if mat_send_decision = 0 then
  Decision(ld_array, lb_done, sender_flag)
  if sender_flag = true then
    mat_send_decision = 1
  else
    mat_send_decision = 0
  end if
end if
if mat_send_decision = 1 then
  Lock_request() と Lock_neighbors() を実行
  ロック成功したら mat_send_decision=2
  ロック失敗したら mat_send_decision=0
end if
if mat_send_decision = 2 then
  Matrix_transition()
end if
if mat_send_decision /= 1 then
  Unlocking_neighboring_PEs()
  locked_by_neihbor=-1
  lb_done=false
  mat_send_decision = 0
  sender_flag = false
  count = TRY_LB_TIMES
end if
```

同期反復解法と比較して，負荷情報の交換や非同期にすることによる収束の悪化，収束検知の遅れなどのオーバーヘッドが存在する．表 1 に非同期反復解法と同期反復解法の結果が示されている．非同期反復解法も相対残差の 2 ノルムが 10^{-2} 未満になっていることが分かる．また非同期反復解法の収束までの時間は同期反復解法と比較して 5 %〜10 %程度長くなった．

実験 2 問題サイズ固定されているので，実行 PE 数を増やすことにより，動的に行ブロックが移動する．行ブロックが移動するオーバーヘッドと負荷が分散されたことによる性能向上の影響が収束時間に反映される．図 2 と表 2 に収束時間と収束後の相対残差の 2 ノルムの値が記されている．非同期反復解法の収束性の目安として 12PE での同期反復解法の収束性能が書かれている．行ブロックが反復的に 2PE 間を移動することにより，漸近的に負荷が分散されるという手法であるため，PE 数が増えるにしたがって行ブロックの移動が頻繁に行われる．そのため，24PE で非同期反復解法を実行すると，12PE で同期反復解法を実行する時間の約 50 %長くかかった．一方で，15PE で並列に実行したときは，約 10 %程度時間が短縮された．

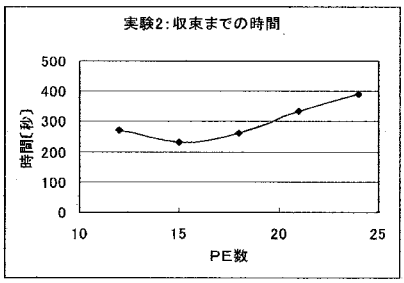

図 2 実験 2：負荷分散性能の評価

5. 評　　価

同期反復解法を非同期反復解法にすることによるオーバーヘッドは，本研究の実験ではせいぜい 10 %程度に抑えられておりそれほど大きくないことが判明した．また非同期反復解法における収束検知についても，2.2 節に記述した手法で同期的に判定するのと同程度の正確さの判定基準となりうることが確認された．また負荷分散については，最適な分散に対してずれが大きくなると，行ブロックの移動が大幅に増えるために，12PE で分散が最適な問題を 24PE で実行すると 12PE の同期反復解法の約 50 %増しの時間が必要となった．一方，12PE で最適に分散されている問題を 15PE で処理すると 10 %程度高速に収束した．この結果からも，大幅に分散を補正する必要がある場合は，一度全体で同期して行列を再分散しなおしたほうが良いと考えられる．

6. おわりに

本論分では，Jacques ら[1] によって提案されている非同期反復解法を不規則疎行列線形ソルバに対して適用し，評価を行った．その結果，非同期反復解法は通常の同期反復解法と比較して，負荷情報の交換や終了処理，収束検知のための処理など追加的に必要となる処理が多くあるが，オーバーヘッドはそれほど大きくなく実現できることを確認した．また，負荷分散による性能向上についても，分散が最適なものから大きくずれていない場合では性能向上が見込めることが分かった．

ただ，分散が最適なものから大きくずれていたときに，再分散を何度も行う必要があり，性能劣化が大くなった．この再分散処理についてはまだ改良の余地があると考えられる．その他にも，反復回数が PE ごとにずれて進むために CG 法が適用できないという問題や，テスト問題や計算環境の多様化などについても研究の余地がある．今後はこれらの問題について研究を進めて行きたい．

表 1　各 PE に問題サイズ固定：
上段が非同期反復解法，下段が同期反復解法

PE 数	2	4	8	16	24
時間 [秒]	11.5	11.6	51.2	214.3	220.0
相対残差	3.92E-03	4.68E-03	5.05E-03	5.25E-03	5.60E-03
時間 [秒]	11.0	11.0	46.9	201.4	207.0
相対残差	4.09E-03	4.50E-03	4.68E-03	4.71E-03	4.90E-03

表 2　全体問題サイズ固定：
上段が非同期反復解法，下段が同期反復解法

PE 数	12	15	18	21	24
時間 [秒]	271.4	233.0	261.6	334.4	389.7
相対残差	5.82E-03	5.54E-03	2.80E-03	1.52E-03	1.36E-03
時間 [秒]	262.6	—	—	—	—
相対残差	5.11E-03	—	—	—	—

謝辞　本研究を進めるにあたり，東京大学の須田礼仁先生から貴重なコメントを頂きました．また東京大学の中島研吾先生から問題生成ルーチンを提供頂きました．謹んで感謝いたします．

参 考 文 献

1) Jacques M. Bahi, Raphael Couturier: Dynamic Load Balancing and Efficient Load Estimators for Asynchronous Iterative Algorithms, IEEE Trans. Parallel and Distributed Systems, volume 16, No.4, 2005.
2) Jacques M. Bahi, Sylvain Contassot-Vivier, Raphael Couturier: A Decentralized Convergence Detection Algorithm for Asynchronous Parallel Iterative Algorithms, IEEE Trans. Parallel and Distributed Systems, volume 16, No.1, 2005.
3) A. Fujii, R. Suda, A. Nishida: Parallel Matrix Distribution Library for Sparse Matrix Solvers, Proceedings of The 8th International Conference on High Performance Computing in Asia Pacific Region (HPC Asia 2005), pp.432-436.
4) Jacques Bahi, Raphael Couturier, Philippe Vuillemin: Asynchronous Iterative Algorithms for Computational Science on the Grid: Three Case Studies, VECPAR 2004, LNCS 3402, pp.302-314, 2005.
5) K. Blathras, D. Szyld, Y. Shi: Parallel processing of linear systems using asynchronous methods, April 1997. Preprint, Temple University, Philadelphia, Pa.