Vol.2010-HPC-126 No.34
2010/8/4

通信と計算の負荷を考慮した並列疎行列ベクトル積の動的負荷分散技術

草場 健一郎[†1] 南里 豪志[†2] 藤野 清次[†2]

本研究では，反復法の中で繰り返し用いられる疎行列ベクトル積の並列化において，各プロセスへの負荷の割り当てを動的に調整する技術を提案する．従来より，疎行列ベクトル積の計算時間を考慮した動的負荷分散技術は提案されてきたが，反復法の中では反復毎に行われる通信の時間も性能に大きく影響を与える上，通信の時間にもプロセス間でばらつきがあるため，負荷分散において考慮する必要がある．そこで本研究で提案する負荷分散技術では，行列の各行の計算時間及び通信時間の見積もりを実行中の処理時間の計測結果より算出し，それをもとに次回の反復で負荷が均等になるようにプロセスへの負荷の割り当てを行単位で調整する．複数の行列データを用いて提案技術の効果を計測した結果，ほとんどの行列で従来の計算時間のみを考慮した動的負荷分散技術以上，もしくは同等の効果が得られることを確認した．

Effect of Runtime Technique for Balancing Load of Communication and Computation in Sparse-Matrix Vector Multiply

Kenichirou KUSABA ,[†1] Takeshi NANRI[†2]
and Seiji FUJINO[†2]

This paper proposes a runtime load-balancing technique for sparse-matrix vector multiplication used in iterative methods. There have been some runtime load-balancing techniques proposed that only considers computation time. However, in iterative methods, the results of the multiplication on each process must be copied to other processes in each iteration. Therefore, the time communication should be also considered for better load-balancing. The technique in this paper estimates computation time and communication time of each row of the target matrix by using measured time of execution on each process at runtime. Then these estimations are used to adjust the distribution of rows to processes so that the entire cost of each process will be balanced in the sparse-matrix vector multiplication in the next iteration. By some experiments over several types of matrices, the proposed technique showed that, in most cases, the effect with this technique is better than or equal to the effect with the technique that only considers only computation time.

1. はじめに

科学技術シミュレーションの多くは，連立一次方程式の求解に帰着される．特に係数行列が大規模で疎である場合，計算やメモリ利用の効率から反復法が用いられることが多い．反復法では，疎行列ベクトル積が反復的に実行され，計算量の大半を占める．そのため，疎行列ベクトル積の実装については，対象のアーキテクチャに応じた様々な高速化手法が提案されている[1)–6)]．特に PC クラスタのような分散メモリ型並列計算環境を対象とした並列化では，負荷のバランスと通信コストが性能に大きく影響する．このうち負荷バランスについて Lee らは，実行時に各プロセスにおける計算時間を計測しながら，全プロセスの計算時間のばらつきを減少させるように反復的にプロセスへの計算の割り当てを調整する手法を提案している[6)]．しかしこの手法は，プロセス間通信の時間を考慮にいれていないため，プロセス毎の通信時間のばらつきが大きい行列では，全体的な負荷を均等にすることができない．

そこで本研究では，MPI で並列化された疎行列ベクトル積において，計算時間だけでなく通信時間まで考慮にいれた負荷分散を行う最適化技術を提案する．本手法では，実行中に各プロセスの計算時間を計測するとともに，各プロセスで必要となる送信，受信それぞれについて通信時間を予測する．これらの情報をもとに，全体の負荷が均等となるような各プロセスへの行の割り当ての調整を反復的に行うことにより，負荷分散の最適化を図る．

2. 並列疎行列ベクトル積の負荷分散と通信

2.1 並列疎行列ベクトル積

本研究では，CRS(Compressed Row Storage) 形式で圧縮格納された疎行列とベクトルの積を計算するプログラムを対象とする．CRS 形式は，行列の非零要素の値と位置を行方向に取りまとめて格納する形式であり，行方向のアクセスを高速に行えるため，行列ベクトル積に適している．

†1 三菱電機株式会社
Mitsubishi Electric Corporation
†2 九州大学
Kyushu University

Vol.2010-HPC-126 No.34
2010/8/4

ここで，A を CRS 形式で圧縮格納された疎行列，x，y をベクトルとすると，疎行列ベクトル積 $y = Ax$ を計算するプログラムでは，行列 A の各行について非零要素を取り出しながら，ベクトル x 内の対応する位置の要素との積を計算し，この積の行毎の総和をベクトル y の一要素とする，という流れで計算を進める．

この疎行列ベクトル積の並列化手段としては，y を分割して各プロセスに割り当て，その領域を計算させる，という手法が一般に用いられる．この場合，行列 A の行単位でプロセスに割り当てることとなる．また，それぞれのプロセスは互いに独立して担当領域の計算を完了させることができる．

この手法による並列疎行列ベクトル積のアルゴリズムを図 1 に示す．図中で `val`, `rowptr`, `colind` は，それぞれ CRS 形式による疎行列の非零要素の値，各行の開始位置ポインタ，各非零要素の列方向の位置を格納したベクトルである．また，このアルゴリズムは連続した行をプロセスに割り当てており，`myrank` はプロセスのランク番号，`start(rank)` はランク `rank` に割り当てられた領域の開始行を表す．なお，本研究で対象とするプログラムでは，行列の分割配置は行わず，全プロセスに行列の全データを保持させるものとする．

2.2 反復法内の疎行列ベクトル積に伴うプロセス間通信

2.1 節の並列化を MPI を用いて行う場合，計算結果が各プロセスに分散しており，そのままでは他のプロセスの計算結果を参照することができない．ところが，反復法の多くでは，ある反復における計算結果 y を，次の反復における x としてそのまま用いるか，もしくは x の算出に用いる．そのため，次の反復で疎行列ベクトル積を実行する前に適切にプロセス間通信を行い，各プロセスに必要なデータのコピーを配置しておく必要がある．

このプロセス間通信の方法として最も簡単なのは，各プロセスが自分の計算結果を全プロセスに送信するものである．この場合，集団通信関数 `MPI_Allgatherv` を用いることができるため，ネットワークの特性に適した通信パターンで通信できる可能性がある．しかし

```
for i = start(myrank) to start(myrank + 1) - 1
  temp = 0.0
  for j = rowptr(i) to rowptr(i+1) - 1
    temp = temp + val(j) * x(colind(j))
  end for
end for
```

図 1 並列疎行列ベクトル積のアルゴリズム

`MPI_Allgatherv` はプロセス数の増加に伴って通信量が増加し，並列化効率の低下を招く．

そこで本研究では，文献[6] で紹介されている通信方法を用いる．これは，通信が本当に必要なプロセス間で，必要最小限の連続領域を転送するものである．図 2 に，この通信方法による通信の例を示す．

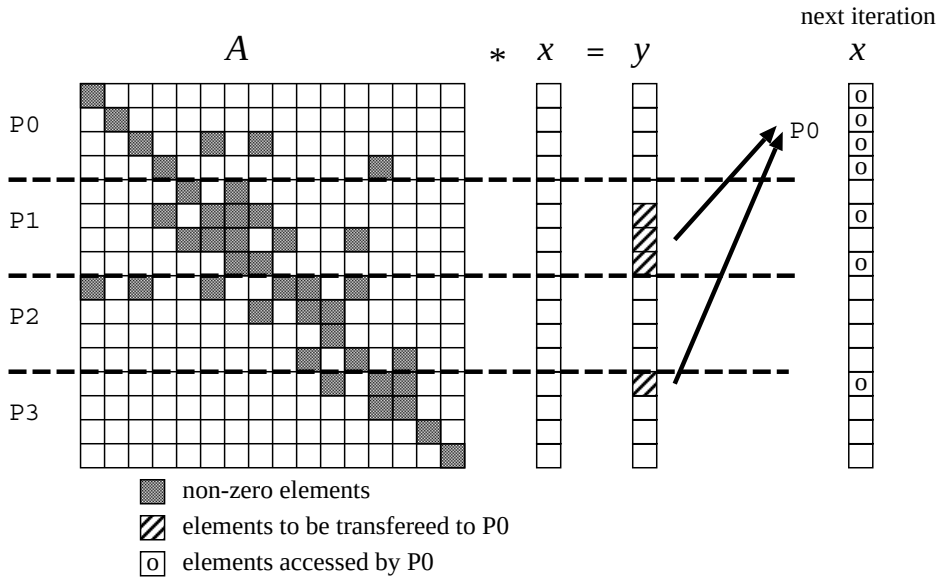

図 2 次回の反復で必要な領域のみを転送する通信方法の例

この図では，16×16 の行列に対する計算を 4 プロセスに分割している．`P0`〜`P3` はランク番号であり，それぞれに対して行列 A が 4 行ずつ割り当てられている．また，簡単のためランク `P0` のプロセスへ向けた通信のみを，矢印で示している．ここで，ベクトル x の中でランク `P0` がアクセスするのは，`P0` に割り当てられた行の非零要素の配置より o で示された要素のみとなる．そのため，`P0` に対して転送が必要なのは，これらの要素に対応する y を計算するランク `P1`, `P3` のプロセスのみである．さらに，斜線で示されている通り，各プロセスが必要最小限の連続領域を転送するようにすることで，通信量を削減できる．

2.3 並列疎行列ベクトル積の動的負荷分散技術 NRET

2.1 節の手法で疎行列ベクトル積を並列化した場合，各プロセスの計算量は割り当てられた行の非零要素の数や配置によって決まる．そのため，最適な割り当てを実行前に決定するのは困難であり，疎行列の非零要素の配置パターンが判明してから割り当てを調整する必要がある．そこで Lee らは，各プロセスの計算時間が全プロセスの計算時間の平均値に近づくように反復的に行のプロセスへの分配を調整する NRET(Normalized Row-Execution-Time-based iteration-to-process mapping) アルゴリズムを提案した[6]．

NRET アルゴリズムでは，ある反復における各プロセスの計算時間を計測し，その時間をそのプロセスに割り当てられた行数で割ることによって，各行の見積もり計算時間を算出する．さらに，全プロセスの計算時間の平均値を目標時間とし，各プロセスに割り当てられる行の見積もり時間の合計が目標時間にできるだけ近づくように，割り当ての調整を行う．具体的には，まずランク 0 のプロセスに行列の 1 行目から順に割り当てながら，そのプロセスの合計見積もり計算時間を増加させていき，目標時間に到達したら割り当て対象を次のプロセスに変更する，という手順を繰り返して，プロセスへの行の割り当てを決定する．

この手法は，簡潔な手段で行毎の見積もり計算時間や目標時間の算出を行うことにより，最適化にともなうオーバヘッドを小さく抑えることができる，という特徴がある．しかし，この手法では負荷分散において通信時間を考慮していない．例えば図 2 において，4 番目の行の割り当て対象を `P0` から `P1` に変更した場合，`P0` は `P1` からのみデータを受信すれば良くなるが，`P1` は自分を除く全プロセスからデータを受信しなければならなくなる．このように，行の割り当てを変更することによって通信時間が変化する可能性があるため，計算時間だけを考慮した負荷分散では十分な性能が得られない場合がある．

3. 通信時間を考慮した動的負荷分散アルゴリズム BRECT

3.1 BRECT アルゴリズムの概要

本研究では，通信時間と計算時間の両方を考慮して行列ベクトル積の負荷分散を行う BRECT(Balanced Row Execution and Communication Time) アルゴリズムを提案する．このアルゴリズムは，毎反復での疎行列ベクトル積を計算した後，行毎の通信時間と計算時間を見積もり，それらの値をもとに，次の反復でプロセス毎の通信と計算の見積もり時間の合計が均等になるように，プロセスへの行の割り当てを決定する．

BRECT アルゴリズムにおける各行の計算時間の見積もりは，2.3 節で紹介した NRET と同様に，疎行列ベクトル積のプロセス毎の計算時間をもとに算出する．一方，各行の通信時間の見積もりは，その行を対象プロセスに割り当てることによって増加する通信回数及び通信量を解析することにより，算出する．この通信時間見積もりの詳細は後述する．

図 3 に，BRECT アルゴリズムの流れを示す．この中で `ncol` は行列の次元数，`NPROCS` はプロセス数，`start(k)` は，プロセス `k` に割り当てられる最初の行の番号を，それぞれ表す．また，`ST(i)`, `RT(i)` は，それぞれその行をプロセス `k` に割り当てることによって必要となる送信，及び受信に要する時間を表す．

この図の流れによる疎行列ベクトル積を，反復回数が規定回数に到達するか，もしくはプロセス間の所要時間のばらつきが基準値以下となるまで繰り返す．本論文の実験では，BRECT アルゴリズム反復の回数を 20 回まで，ばらつきの基準値をプロセス毎の総実行時間の最大値の 5% と規定した．

3.2 通信時間の見積もり

本研究では，BRECT アルゴリズム中の通信時間の予測に，線形の通信性能モデル $T_{comm} = \alpha \times m + \beta$ を用いる．ここで m は倍精度実数の要素数である．また，α と β の二つの定数は，実行開始時に要素数を変化させながら ping-pong 通信の所要時間を計測し，その結果に対して最小二乗法を適用して算出しておく．

この通信性能モデルをもとに，行列の `i` 行目をランク `p` のプロセスに割り当てることによる送信時間と受信時間の増分 `ST(i)`, `RT(i)` を，以下のように算出する．

まず `ST(i)` は，ランク `p` に `i` 行目を割り当てることにより必要となる送信を導出し，それに基づいて算出する．この割り当てにより，`y(i)` をランク `p` が計算することになる．そのためランク `p` は，次の反復で `x(i)` を参照するプロセスに対して，`y(i)` を送信する必要がある．この，`y(i)` の送信先プロセスは，行列の `i` 列目の各非零要素について，その要素が存在する行が割り当てられているプロセスを調べ，その中からランク `p` を除外することによって得られる．ただし図 3 より，`ST(i)` を算出している時点で `i` 行目以降の次反復でのプロセスへの割り当て状況は未定であるため，それらの行については現反復での割り当て情報をもとに送信先プロセスを導出する．

図 4 に，送信先プロセスの導出の様子を示す．この図の時点では 6 行目をランク `P1` に割り当てようとしている．そこで，送信先プロセスを判定するため，行列の 6 列目の非零要素の位置を確認する．この例では 6 列目の 3, 6, 7, 9 行目に非零要素がある．このうち 6, 7 行目はランク `P1` に割り当てられているため通信は不要である．そこで，3 行目が割り当てられているランク `P0`, 及び 9 行目が割り当てられているランク `P2` が，`P1` からの送信先プロセスとなる．なお，この時点で 6 行目以降のプロセスへの次反復での割り当ては未定

Vol.2010-HPC-126 No.34
2010/8/4

1. 各プロセス k で，割り当てられた行の疎行列ベクトル積を計算した後，次の反復に向けた通信を発行する．
   - 計算時間 CT(k) を計測．
   - 発行した通信の量からプロセス毎の通信時間 COMM(k) を算出．
2. CT(k)，COMM(k) それぞれについて，MPI_Allreduce により全プロセス分を集計し，平均値を算出．この値を，次の負荷分散の目標値 tCT とする．
3. 行毎の計算時間の見積もり NRET(i) を算出する．

```
for k = 0 to NPROCS - 1
  for i = start(k), start(k+1) - 1
    NRET(i) = CT(k) / (start(k+1)-start(k))
  end for
end for
```

4. 次の反復で各プロセスの計算時間と通信時間の和が tCT に近づくように，行をプロセスに割り当てる．

```
start(0) = 1
for k = 0 to NPROCS - 1
  eCT(k) = 0
  while (eCT(k) < tCT && (i <= ncol))
    eCT(k) += NRET(i) + ST(i) + RT(i)
    i = i + 1
  end while
  start(k + 1) = i
end for
start(NPROCS) = ncol + 1
```

図 3 BRECT アルゴリズムを適用した疎行列ベクトル積の反復実行

であるため，現反復での割り当てを使って送信先プロセスを導出している．

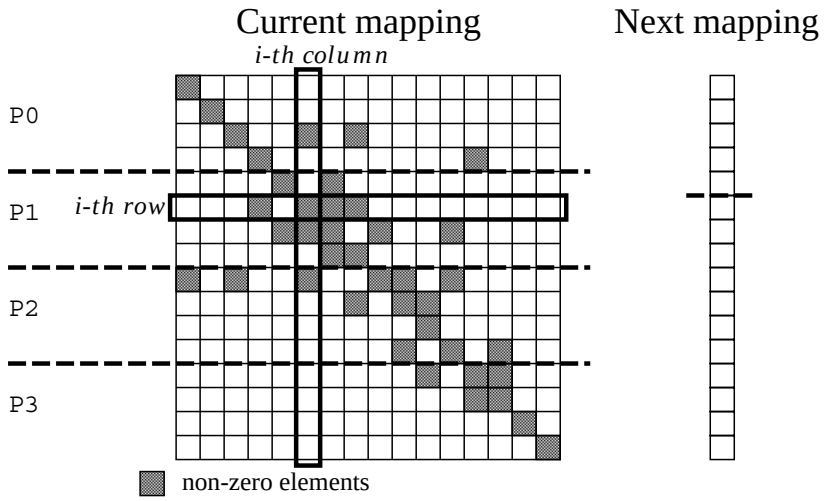

図 4 BRECT アルゴリズムにおける送信先プロセスの導出

i 行目の割り当てに対して送信先プロセスが確定すると，それぞれの送信先プロセスについて，今回の割り当てによって新規にランク p からの送信先プロセスとして追加されたプロセスである場合は，ST(i) に β を加算する．さらに，それぞれの送信先プロセスについて，この割り当てによって増加した送信要素数に α を乗じた値を ST(i) に加算する．なお，ST(i) の算出には行列の列方向の探査が必要なので，行列データとして CRS 形式だけでなく CCS 形式も保持しておく．

一方 RT(i) の算出は，ランク p に i 行目を割り当てた場合にランク p で必要となる受信の受信元プロセスを導出し，それに基づいて算出する．この受信元プロセスは，行列の i 行目の非零要素の位置より導出できる．その後，ST(i) と同様に α と β を適宜加算することによって RT(i) を算出する．

3.3 BRECT アルゴリズムの特徴

BRECT アルゴリズムの特徴は，NRET と同様に最適化に伴うオーバーヘッドを抑えながら，通信を含めた総合的な負荷のバランスを均等にすることによって，処理性能の向上

を図る点である．NRET アルゴリズムに対し追加されるオーバーヘッドは，図 3 における COMM(k) の算出と集計に要する時間，及び ST(i), RT(i) の算出時間である．このうち COMM(k) の算出に要する時間は最大でプロセス数に比例し，ST(i), RT(i) の算出に要する時間は行列の次元数に比例するが，どちらも疎行列ベクトル積の計算時間に比べれば無視できるとみなす．また，COMM(k) の集計には MPI_Allreduce を用いるが，実際には CT(k) の集計のための MPI_Allreduce について，要素数を 1 要素から 2 要素に増やすだけなので，所要時間はほとんど変化しない．

一方，各プロセスの計算時間や通信時間を予測する手段としては，より精度の高い方法を用いることも考えられる．例えば図 3 の 4. で ST(i) や RT(i) の算出時に，一部，古い割り当て情報を用いているため，正確な通信量や通信回数を予測できない．これに対し，割り当て情報を更新しながら 4. を繰り返すことによって，予測精度を向上させることが可能である．また，通信モデルとして，3.2 節で紹介した線形モデルではなく，より精度の高い LogP モデル[7] や LogGP モデル[8] 等を用いるという選択肢もある．しかし，一般に性能予測の精度向上にはオーバヘッドの増加が伴うため，最適化による効果とのトレードオフについての詳細な調査が必要であり，現時点では今後の課題としている．

4. 実　　験

4.1 実験の概要

本研究で提案した BRECT アルゴリズムが，反復法で用いられる疎行列ベクトル積の性能に与える影響を調べるため，疎行列ベクトル積の計算，及び次の反復に必要な通信を 1000 回行うプログラムを作成し，以下の 3 通りについて全体の所要時間と通信時間を計測して比較する．なお，最適化を行わない場合は，プロセスあたりの行数が均等になるように割り当てる．

- 最適化を行わない場合
- NRET アルゴリズムを適用した場合
- BRECT アルゴリズムを適用した場合

実験環境は 表 1 に示す通りである．また，実験に用いる入力データとしては，Florida 大学の疎行列データベース[9] からダウンロードした 5 個の疎行列 (表 2) について，それぞれ計測する．

4.2 実 験 結 果

表 3〜7 に，各行列についてプロセス数を変化させて 1 回の反復あたりの所要時間と通信時間を計測した結果を示す．

表 1　実験環境

CPU	Intel Xeon 3.0GHz 4 コア/ノード
メモリ	8GB/ノード
ノード数	16
ネットワーク	InfiniBand
OS	RedHat Enterprise Linux WS4
コンパイラ等	Fujitsu Fortran Compiler & MPI Library

表 2　入力データ

行列名	次元数	非零要素数	行当たり平均非零要素数
matrix9	103,430	2,121,550	20.51
venkat50	62,424	1,717,792	27.52
ecl32	51,993	380,415	7.32
turon m	189,924	1,634,766	8.61
xenon1	48,600	1,181,120	24.30

まず，全ての行列において，負荷分散の最適化技術を適用しない場合に比べて，NRET もしくは BRECT を用いた動的負荷分散技術を適用した場合の方が高い性能が得られている．これにより，行列に応じた負荷分散の最適化が有効であることが示された．

一方，NRET を用いた場合と BRECT を用いた場合の優劣は，行列やプロセス数に依存する．まず matrix9 では，全体的に BRECT を用いることによって，NRET を用いた場合以上に高速化できていることが分かる (表 3)．特にプロセス数が 16, 及び 32 の場合に，NRET を用いた場合に対して 1.2〜1.4 倍の速度向上が見られる．

また，venkat50 でもプロセス数が 32 の場合は，BRECT を用いることにより NRET の 1.4 倍程度の速度向上が見られる (表 4)．しかし，それ以外のプロセス数ではほぼ同等か，若干低い性能が得られている．

ecl32 及び turon_m では，BRECT を用いた場合の性能は，プロセス数 32 までは NRET を適用した場合と同等かそれ以上である (表 5)．しかしプロセス数が 64 になると，どちらも NRET より低い性能となっている．

最後に xenon1 では，プロセス数が 8 の場合以外は BRECT よりも NRET を用いた場合の方が性能が高い (表 7)．

表 3 各手法による全体時間と通信時間 (行列 matrix9)

Procs	最適化無し		NRET		BRECT	
	通信	全体	通信	全体	通信	全体
1	0.725	11.037	0.002	9.741	0.001	9.708
2	1.483	9.377	0.758	8.947	0.756	8.928
4	2.265	7.294	0.746	5.370	0.611	5.297
8	2.701	5.294	0.945	2.982	0.678	2.669
16	2.960	4.337	1.171	1.761	0.412	1.190
32	3.187	3.962	0.764	1.033	0.388	0.796
64	3.861	4.301	0.757	0.957	0.564	0.837

(秒)

表 4 各手法による全体時間と通信時間 (行列 venkat50)

Procs	最適化無し		NRET		BRECT	
	通信	全体	通信	全体	通信	全体
1	0.568	11.534	0.002	10.453	0.002	10.602
2	0.866	6.274	0.453	5.710	0.441	5.697
4	1.337	5.092	0.580	4.157	0.595	4.182
8	1.526	3.355	0.638	2.146	0.671	2.191
16	1.560	2.410	0.609	1.031	0.444	1.008
32	1.549	1.971	0.732	0.937	0.332	0.668
64	2.540	2.738	0.601	0.753	0.493	0.790

(秒)

表 5 各手法による全体時間と通信時間 (行列 ecl32)

Procs	最適化無し		NRET		BRECT	
	通信	全体	通信	全体	通信	全体
1	0.320	2.423	0.001	1.912	0.001	1.934
2	0.726	2.001	0.300	1.380	0.262	1.351
4	1.036	1.843	0.474	1.111	0.467	1.110
8	1.181	1.554	0.812	1.109	0.800	1.067
16	1.135	1.334	1.074	1.231	0.956	1.143
32	1.185	1.332	1.143	1.235	0.960	1.101
64	2.100	2.191	1.484	1.574	1.267	1.526

(秒)

表 6 各手法による全体時間と通信時間 (行列 turon_m)

Procs	最適化無し		NRET		BRECT	
	通信	全体	通信	全体	通信	全体
1	1.993	15.140	0.002	12.972	0.002	13.313
2	3.802	11.659	1.823	9.691	1.854	9.774
4	4.958	9.656	2.271	6.892	2.251	6.900
8	5.852	8.397	3.256	5.552	2.648	5.154
16	5.984	7.427	3.585	4.609	2.128	3.417
32	6.413	7.314	2.094	2.474	1.060	1.678
64	6.858	7.478	0.971	1.193	0.814	1.301

(秒)

表 7 各手法による全体時間と通信時間 (行列 xenon1)

Procs	最適化無し		NRET		BRECT	
	通信	全体	通信	全体	通信	全体
1	0.347	7.436	0.002	7.013	0.002	7.016
2	0.719	5.237	0.177	4.573	0.156	4.555
4	0.969	3.573	0.262	2.620	0.260	2.628
8	1.089	2.220	0.298	0.970	0.301	1.024
16	1.091	1.532	0.197	0.412	0.179	0.441
32	1.126	1.345	0.139	0.274	0.156	0.355
64	1.777	1.892	0.261	0.369	0.285	0.488

(秒)

5. 考　　察

本節では，実験結果で示された BRECT アルゴリズムの効果について，分析を行う．

5.1 計算時間と通信時間を考慮した行の割り当て

提案手法による効果を解析するため，BRECT アルゴリズムと NRET アルゴリズムでの行の割り当て方を比較する．図 5 は，matrix9 を 16 プロセスで実行した場合について，BRECT アルゴリズム，及び NRET アルゴリズムを用いた動的負荷分散後に各プロセスに割り当てられた行の数を，それぞれ示している．

図より，BRECT アルゴリズムを用いた場合，ランク 14 及び 15 のプロセスに対する行の割り当てが他のプロセスよりも極端に少ない．これは，BRECT が matrix9 の非零要素の分布により予測される通信時間を考慮したためである．matrix9 の非零要素は，対角項付

近，及び最右端の列に配置されている．このうち最右端の列に非零要素があることにより，全プロセスは行列の最下行を担当したプロセスからのデータを必要とする．行列の最下行は必ず末尾のプロセス，すなわち図 5 におけるランク 15 に割り当てられるので，ランク 15 は他のプロセスよりも多くの通信を担当する．

NRET アルゴリズムの場合，この通信時間を考慮に入れないため，ほぼ他のプロセスと同程度の数の行をランク 15 に割り当てている．一方 BRECT アルゴリズムは，通信時間を考慮して他のランクへの割り当てを増やした結果，ランク 15 に少ない行数が割り当てられ，全体の負荷をより均等にすることが出来ている．なお，ランク 14 への割り当ても減少している原因は，現在調査中である．

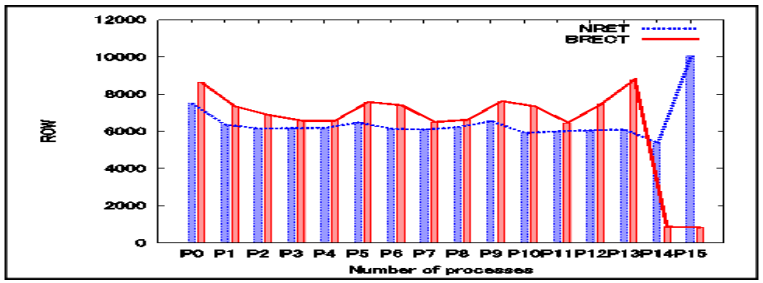

図 5 matrix9 における行のプロセスへの割り当て (プロセス数 16)

5.2 BRECT アルゴリズムにおける性能低下の要因

表 7 より，xenon1 においては BRECT アルゴリズムを用いた場合の性能が，NRET アルゴリズムを用いた場合に対して低下することが判明した．xenon1 は，非零要素が対角項付近のみに配置されているため，行あたりの非零要素の分布は，最上部と最下部の行が他の行よりも少ない．そのため NRET アルゴリズムでは，先頭と末尾のプロセスに他の行よりも多い行数を割り当てることで，計算時間の均等化を図れる．一方通信は，隣接するランクとの間でしか発生しない．すなわち先頭と末尾のプロセスは他のランクより通信回数が少なくなるため，BRECT アルゴリズムでも，やはり先頭と末尾のプロセスへの割り当てを増加させる．その結果，双方のアルゴリズムがほぼ同じ割り当てを選択することになり，NRET アルゴリズムに対する BRECT アルゴリズムの優位性が低くなる．さらに，プロセス数が増えて各プロセスの実行時間が減少してくると，BRECT アルゴリズムのオーバヘッドが無視できなくなり，全体の所要時間が NRET よりも長くなる．このように，NRET アルゴリズムによる負荷の調整が計算だけでなく通信の時間の均等化にも働く場合は，BRECT アルゴリズムの優位性が薄れる．

一方，プロセス数が増えると NRET アルゴリズムを用いた方が性能が高くなる場合がある．例えば turon_m (表 6) では，プロセス数が 32 までは BRECT アルゴリズムを用いた方が高速であったが，プロセス数が 64 になると逆転し，NRET アルゴリズムを用いた方が速くなる．この原因として，割り当ての変更によって全体の通信量の変化したことが考えられる．表 8 に，turon_m に対して NRET アルゴリズムと BRECT アルゴリズムを用いた場合の通信量を示す．これは，各プロセスが送信もしくは受信した要素数の合計である．この表から明らかなように，プロセス数が 32 の場合は BRECT アルゴリズムを用いた方が通信量が少ないのに対し，プロセス数が 64 の場合は NRET アルゴリズムを用いた方が通信量が少ない．

BRECT アルゴリズムは，通信時間を考慮した負荷分散を行うだけであり，通信量全体の増減は考慮していない．そのため，通信量が増加するような割り当てを選択した場合に性能が低下し，結果的に NRET アルゴリズムの方が高速になる場合がある．疎行列ベクトル積における通信量や通信時間を削減するための割り当て方法は，いくつか提案されている[10),11)]．これらを参考にした BRECT アルゴリズムの改良は，今後の課題である．

表 8 行列 turon_m に対する各手法を用いた場合の通信量

プロセス数	NRET	BRECT
32	1,909,506	1,841,986
64	2,045,782	2,223,138

6. むすび

反復法の中で用いられる疎行列ベクトル積の並列化において，動的に負荷分散を行う最適化技術として，計算だけでなく通信時間も考慮した BRECT アルゴリズムを提案した．実験の結果，計算のみを考慮した動的負荷分散アルゴリズム NRET に対し，多くの場合で性能を向上できることを示した．

また，今後の課題として，より精度の高い性能予測手法や，全体の通信量を削減する最適化技術の検討をあげた．他に，実際の反復法における効果の確認等も行う予定である．

Vol.2010-HPC-126 No.34
2010/8/4

参 考 文 献

1) J. Bolz, I. Farmer, E. Grinspun and P. Schröoder: Sparse Matrix Solvers on the GPU, Conjugate Gradients and Multigrid, ACM SIGGRAPH 2003 Papers, pp. 924-932, 2003.
2) E.J. Im, K. Yelick and R. Vuduc: Optimization Flamework for Sparse Matrix Kernels, International Journal of High Performance Computing Applications, Vol. 18, No. 1, pp. 135-158, 2004.
3) G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis and N. Koziris: Understanding the Performance of Sparse Matrix-Vector Multiplication, In PDP'08, Proceedings of the 16th Euromicro International Conferencce on Parallel, Distributed and Network-Based Processing, 2008.
4) 工藤 誠, 黒田 久泰, 片桐 孝洋, 金田 康正: 並列疎行列ベクトル積における最適なアルゴリズム選択の効果, 第 147 回アーキテクチャ研究会, 情報処理学会研究報告 2002-ARC-147, pp. 151-156, 2002.
5) S. Williams, L. Olikera, R. Vudecc, J. Shalfa, K. Yelick and J. Demmel: Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms, Parallel Computing, Vol. 35, pp. 178-194, 2009.
6) S. Lee and R. Eigenmann: Adaptive Runtime Tuning of Parallel Sparse Matrix-Vector Multiplication on Distributed Memory Systems, Proceedings of the 22nd Annual International Conference on Supercomputing 2008, pp. 195-204, 2008.
7) D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian and T. von Eicken: LogP: Towards a Realistic Model of Parallel Computation, Primciples Practice of Parallel Programming, pp. 1-12, 1993.
8) A. Alexandrov, M. Ionescu, K. Shauser and C. Scheiman: LogGP:Incorporating Long Messages into the LogP Model - One Step Closer Towards a Realistic Model for Parallel Computation, In 7th Annual Symposium on Parallel Algorithms and Architectures, May 1995.
9) University of Florida Sparse Matrix Collection: http://www.cise.ufl.edu/research/sparse/matrices/
10) U.V. Catalyurek, C. Aykanat: Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication, IEEE Transactions on Parallel and Distribuited Systems, Vol. 10, No. 7, pp. 673-693, 1999.
11) E.G. Boman and U. Catalyurek: Constrained Fine-Grain Parallel Sparse Matrix Distribution, SIAM Workshop on Combinatorial Scientific Computing, 2007.