大規模計算向け通信時間最適化ツール RMATT における実行時間の高速化

今出 広明† 平本 新哉† 三浦 健一† 住元 真司†
黒川 原佳‡ 横川 三津夫‡ 渡邊 貞‡

本論文では，RMATT(Rank Map Automatic Tuning Tool)における実行時間の高速化について述べる．RMATT は MPI アプリケーションにおけるランク配置を最適化することで通信処理時間を短縮することができるが，実行に長時間を要することが問題であった．この問題を解決するため，変更されたランクの通信処理のみを再計算する他，通信しないランク間のテーブル作成を省くことで計算量を大幅に削減する方法を開発した．評価の結果，4,096 ランクの Allgather bruck アルゴリズムの最適化に従来 16 時間かかっていた実行時間を 7.4 分に短縮できることを確認した．また，NAS Parallel Benchmark におけるクラス B，プロセス数 1,024 の CG に本 RMATT を適用し，京コンピュータ上において CG の実行時間を 7%削減することを確認した．

Reduction of Execution Time of RMATT for Communication Time Optimization for Large Scale Computation

HIROAKI IMADE† , SHINYA HIRAMOTO† , KENICHI MIURA† , SHINJI SUMIMOTO† ,
MOTOYOSHI KUROKAWA‡ , MITSUO YOKOKAWA‡ and TADASHI WATANABE‡

This paper presents optimization of execution time of RMATT (Rank Map Automatic Tuning Tool). RMATT realizes reduction of MPI communication time in MPI application programs by optimizing location of MPI ranks. However, RMATT has an issue that the execution time takes too long. To resolve this issue, we have developed a method with re-calculation of communication processing among only ranks replaced, and elimination of table creation among ranks without communication. The evaluation of the method shows that it dramatically reduces the execution time of 4,096 rank allgather bruck algorithm from 16 hours to 7.4 minutes, and RMATT reduces 7% of application execution time of NAS parallel benchmark CG B class 1,024 rank on K computer.

1. はじめに

高性能計算に対する要求はとどまるところを知らず，Linpack 演算性能の世界ランクである Top500[1]において，2011 年 6 月のリストではトップ 10 までが 1 ペタフロップスを超える演算能力を持っている．演算能力に対する要求は今後も続くことが予想され，2018 年にはエクサフロップスクラスの計算システムが登場するといわれている．

こうした計算性能に対する要求に従い，高性能計算システムを構成する計算ノードの数も増加の一途をたどっており，TSUBAME2.0[2]や T2K[3]，RICC[4]などの間接網システムに加え，京コンピュータ[5]や BlueGene/P[6]，Jaguar[7]などの Torus，Mesh など

† 富士通株式会社
Fujitsu, Ltd.
‡ 理化学研究所
Inst. of Physical and Chemical Research

の直接網のシステムも多く利用されるようになった. 計算システムが大規模になるほど間接網のシステムの構築が難しくなることから, 今後直接網のシステムが増加すると予想される.

直接網のシステムで MPI アプリケーションを実行する場合, 通信処理内容(通信パターン)に応じランクを適切なノードに配置(ランク配置)することで, 経由するノード数(ホップ数)や輻輳を抑え通信処理時間を短縮することができる. これをランク配置最適化と呼ぶ.

我々は, 大規模なアプリケーションにおける通信の最適化の手段として, ランク配置最適化を自動で行う Rank Map Automatic Tuning Tool(RMATT)を開発している[8]. RMATT は分割処理と焼きなまし法(Simulated Annealing: SA)[9]を用いることで大規模なアプリケーションに対応しているが, 実行に長時間を要し, 利用者への利便性に問題があった.

本論文ではチューニング時間を短縮するための RMATT の実行時間の最適化について述べる. RMATT の実行時間が長い原因は, ランク配置の評価においてホップ数と通信経路の計算に時間がかかるためである. そこでランク配置を新しく評価する際に, 以前に評価したランク配置から変更されたランクの通信処理のみを再計算することで計算量を大幅に削減する方法を開発した. さらに, すべてのランク間が通信するとは限らないため, 通信しないランク間のテーブルの作成を省くことにより, 計算時間の削減とテーブルに必要なメモリ量を削減した. これにより, 4,096 ランクの Allgather bruck アルゴリズム[10]のチューニングにおいて 16 時間かかっていたチューニング時間を 7.4 分まで短縮することができた. また, NAS Parallel Benchmark[11]におけるクラス B, プロセス数 1,024 の CG に本 RMATT を適用し, 現在開発中の京コンピュータ上において 4.5 分のチューニング時間で CG の実行時間を 7%削減することができた.

2 章ではランク配置最適化と, RMATT の概要と現状の問題点, RMATT 高速化の目標について述べる. 3 章では RMATT の高速化として, 方針と高速化の内容について述べる. 4 章では高速化した RMATT の評価と, 実際のアプリケーションと計算機環境に対して RMATT を用いたチューニングについて述べる. 5 章では結論と今後の予定について述べる.

2. ランク配置最適化と RMATT

2.1 ランク配置最適化

直接網において MPI アプリケーションを実行する場合, ランクをどのノードに配置し実行するかにより通信処理時間は異なる. これはランク配置によりランク間の通信時のホップ数や輻輳の発生頻度が異なるためである. 直接網では各ランクの通信パターンに応じランク配置を適切に設定することで, 通信処理時間の短縮が可能である. これをランク配置最適化と呼ぶ.

RMATT では, Open MPI などの MPI ライブラリで利用可能なランク配置を指定するファイル(ランクマップファイル)を用いたランク配置最適化を想定している. ランクマップファイルを用いたランク配置最適化では, ランク配置の決定と評価を試行錯誤的に行う必要がある. N ランクの MPI アプリケーションにおけるランク配置の総組み合わせ数は $N!$ 個となる. 開発者の経験や知識により必ずしもすべてのランク配置を評価する必要は無いため, 小規模なアプリケーションにおいては手作業でも可能である. しかし, 大規模なアプリケーションにおいては手作業では非常に面倒な作業となる. 大規模なアプリケーションにおける開発者へのランク配置最適化の負担を軽減するために, RMATT はランク配置最適化を自動で行う機能を提供している.

2.2 RMATT の概要と課題

RMATT の概要

RMATT は大規模なアプリケーションにおいてランク配置最適化を実行するために, 以下の要件を持つ.

・大規模な問題への対応
　数千～数万ランクのアプリケーションに対応する.
・多様な問題への対応
　多様な通信パターンに対応する.

これらの要件に対応するため, RMATT は以下の 2 つの処理から構成される.

・ランク分割処理 BISEM
　ランクを, よく通信を行うもの同士で複数のグループに分割し, グループ単位でネットワーク上に配置する. グループ単位で配置することで総組み合わせ数を大幅に削減することができ, 大規模なアプリケーションに対応することができる. この分割処理を BISEM(BISEction rankMap)と呼ぶ.
・ランク配置最適化処理 OPTIM
　BISEM の実行結果を初期解とし, 焼きなまし法(SA) を用いてランク配置最適化を行う. SA は問

題に依存しない最適化手法であるため，様々な通信パターンを持つアプリケーションに対応できる．この最適化処理を OPTIM (OPTImization rankMap)と呼ぶ．

アプリケーション開発者は RMATT を用いて以下の流れでランク配置最適化を行うことができる．

1. 通信パターンとネットワーク構成の決定
   MPI アプリケーションの通信パターンと，ネットワーク構成を決定する．
2. RMATT の実行
   ステップ 1 で決定した通信パターンとネットワーク構成を入力値とし，RMATT を実行する．RMATT の実行では，まず BISEM が実行され，その結果を元に OPTIM が実行される．RMATT の実行結果として MPI のランクマップファイルが出力される．
3. MPI アプリケーションの実行
   ステップ 2 で作成されたランクマップファイルを指定し MPI アプリケーションを実行する．

アプリケーション開発者の余暇時間に余剰 CPU 上で RMATT を実行することにより，開発者は時間や計算資源を有効に利用することができる．

プロトタイプ版 RMATT の問題点と目標

RMATT のプロトタイプ版を Java で作成し，4,096 ランクの Allgather bruck アルゴリズムに適用したところ，ネットワークシミュレータ OpenNSIM[12]上で通信処理時間を 75%削減することができた[8]．しかし，最適化には 20 台の計算機を用いて並列実行しても 16 時間を要してしまった．これは，京コンピュータのような大規模なペタフロップスクラスの計算システムでは実行不可能に近いことを示している．RMATT の実行時間の大幅な短縮が必要である．

利用者の利便性から，RMATT の目標とする実行時間は短ければ短いほどよい．まずは数千ランクから数万ランクのアプリケーションにおいて，RMATT の実行を数分から数十分で行うことを目標とする．例えば 4,096 ランクの Allgather bruck アルゴリズムにおける RMATT の実行を 10 分程度で行う場合，現状の RMATT に比べ実行時間を 96 分の 1 程度にする必要がある．

3. RMATT の高速化

3.1 RMATT 高速化の流れ

4,096 ランクの Allgather bruck アルゴリズムに対して 20 台の計算機を用いてプロトタイプ版 RMATT を実行した際の OPTIM と BISEM の実行時間を表 1 に示す．表 1 より OPTIM の実行時間が圧倒的に長いことから，これを高速化する．

表 1 OPTIM と BISEM の実行時間

	実行時間(秒)
BISEM	7,083
OPTIM	51,866

高速化を行う前に，仕様上の制約から C 言語に書き換えた．本論文では，この C 言語版 RMATT を対象に高速化を行っている．なお，C 言語に書き換えたことにより実行時間を 10%短縮することができた．この C 言語版 RMATT について，OPTIM 内の細かい処理時間を測定し，ボトルネックとなる処理を特定する．ボトルネックの原因を改善することで高速化を行う．

なお，BISEM については分割処理の精度を上げるために反復している処理が多くあり，プロトタイプ版 RMATT では必要以上に反復回数をとっていた．したがって，性能が落ちないところまで反復回数を下げることにより，BISEM の実行時間を短縮することができる．

3.2 OPTIM におけるボトルネックの調査

OPTIM は以下の流れで処理が行われるため，それぞれの実行時間を測定する．

1. スワップランクの決定
   現在のランク配置からランダムに数ランク選択する．現在のランク配置を親配置と呼び，選択されたランクをスワップランクと呼ぶ．
2. 子配置の生成
   スワップランクの配置をランダムに入れ替えた新しいランク配置をあらかじめ決められた個数生成する．生成したランク配置を子配置と呼ぶ．
3. 子配置の評価
   ステップ 2 で生成した子配置を評価する．
4. 新しい親配置の選択
   生成した子配置の中から新しい親配置を 1 個選択する．このとき，SA の条件により新しい親配置が選択されない場合もある．
5. OPTIM の終了条件が満たされるまでステップ 1 から 4 を繰り返す．

RMATT ではステップ 1 から 4 までの処理を 1 世代と呼ぶ．4,096 ランクの Allgather bruck アルゴリズムでの 1 世代あたりの各ステップの実行時間を測定した結果を表 2 に示す．表 2 から，子配置の評価処理が大

表 2 1 世代あたりの実行時間

	1 世代あたりの実行時間(秒)
スワップランクの決定	0.04×10^{-4}
子配置の生成	0.02
子配置の評価	82.60
親配置の選択	0.35×10^{-2}

部分を占めていることがわかる.

このことから, RMATT の高速化では子配置の評価処理の高速化が必須であることがわかる.

3.3 子配置評価処理におけるボトルネックの原因

子配置の評価処理では, 以下の評価式を計算する処理が行われている[8].

$$score = \sum_{i,j\in全ランク}(hop_{i,j}\cdot size_{i,j})\times \max_{link} \ldots(1)$$

$hop_{i,j}$ はランク i からランク j へ送信した際のホップ数を示し, $size_{i,j}$ はランク i からランク j への送信サイズを示す. $link$ はあるノード間を流れた転送量を示し, $\max_{link}$ は全ノード間における $link$ の最大値である. 評価式(1)を計算するためには, 各ランクにおける送信先ランクと送信サイズを求める処理と, 各ノード間の $link$ を保存するテーブル(ネットワーク状態テーブル)から最大値を検索する処理が必要となる.

子配置評価処理のボトルネックを特定するために, 高速化前の RMATT について, 4,096 ランクの Allgather bruck アルゴリズムでの子配置評価処理における, ネットワーク状態テーブルの初期化処理, 送信先ランクと送信サイズを求める処理, 評価式(1)を計算する処理の実行時間を測定した結果を, 表 3 に示す. 表 3 から, 子配置の評価処理がボトルネックとなっている原因が, 評価式(1)の計算処理と, 送信先ランクと送信サイズの検索処理にあるといえる. 以下にこれら 2 つの処理に時間がかかる原因についてまとめる.

ランク配置の評価における計算量

評価式(1)の各項について, ランク数 N における計算量を求める. ランク i の通信処理について, 送信先ランクの数は $O(N)$ であり, dim 次元トーラスにおけるホップ数は $O(N^{1/dim})$ となる. したがって, ランク i における $\sum(hop_{i,j}\cdot size_{i,j})$ の計算量は $O(N\times N^{1/dim})$ となり, 全ランクにおける $\sum(hop_{i,j}\cdot size_{i,j})$ の計算量は $N\times O(N\times N^{1/dim})$ となる. $\max_{link}$ は $N\times N$ のネットワーク状態テーブルから最大値を求めるため, 計算量は $O(N^2)$ となる. ネットワーク状態テーブルの初期化処理についても $O(N^2)$ となる. これらの処理の計算オーダは大規模なアプリケーションにおいては非常に大規模な計算量となってしまう.

表 3 ランク配置評価処理における実行時間

	実行時間(ミリ秒)
評価処理全体	93.5
ネットワーク状態テーブル初期化	18.2
送信ランクと送信サイズの検索	29.5
評価式(1)の計算	44.0

送信先ランクと送信サイズの参照回数

これまで RMATT ではランク間の通信内容を記した通信パターンを送信サイズテーブルと呼ぶテーブルで管理していた. 送信サイズテーブルでは送信元ランクを列と, 送信先ランクを行とし, 送信サイズが格納されている. 値が 0 のとき通信が行われないことを意味する. あるランクの送信先ランクを求める場合, そのランクのすべての行について 0 でない箇所を検索する必要がある. N ランクからなるランク配置を評価する際, 検索には送信サイズテーブルを N^2 回参照する必要があるため大規模なアプリケーションにおいては非常に大きな参照回数とテーブルサイズとなる.

3.4 高速化の方針

ランク配置評価処理における計算量の削減

OPTIM において一度に変更するスワップランクは数個である. したがって, 親配置と子配置では大部分のランクの通信処理は同じホップ数と通信経路となる. そこで, 子配置生成時に変更されたスワップランクのホップ数と通信経路のみ再計算し, 残りのランクの通信処理については親配置の情報をそのまま利用することとする. これにより計算量の大幅な削減が可能となる.

送信先ランクと送信サイズの参照回数の削減

MPI アプリケーションにおいて, 一般にあるランクがすべてのランクと 1 対 1 通信を行う場合は少ないと考えらえることから, 送信サイズテーブルのすべてのエントリが 0 以外になる確率は低いと考えられる. 実際に 4,096 ランクの Allgather bruck アルゴリズムの場合, 送信サイズの 78.7%が 0 であった. このことから, 送信サイズテーブルにおける 0 の部分を詰めて小さくすることで, 参照回数とテーブルサイズの削減が期待できる.

3.5 RMATT の高速化

3.5.1 ランク配置評価処理の高速化

高速化した子配置評価処理の疑似コードを図 1 に示す. この評価処理を行う前に, 親配置から N_child 個の子配置が生成されているものとする. network_table には全ランクの通信処理における, $\sum(hop_{i,j} \cdot size_{i,j})$ の値と, ノード間を流れた転送量 $link$ を記録する. 評価式(1)の値は network_table に記録されている情報から求めることができる.

1 行目の init_comm_info( )では, まず, 引数として渡された親配置の全ランクの通信情報について, $\sum(hop_{i,j} \cdot size_{i,j})$ と $link$ を計算し network_table にセットする. 次に, 親配置のスワップランクの通信情報について, $\sum(hop_{i,j} \cdot size_{i,j})$ と $link$ を計算し, network_table から削除する. このとき削除した値を保存しておく.

5 行目の add_comm_info( )では, 引数として渡された子配置 i のスワップランクの通信処理における, $\sum(hop_{i,j} \cdot size_{i,j})$ と $link$ を計算し, network_table に追加する. このとき追加した値を保存しておく.

7 行目の eval( )では network_table の情報から評価式(1)の値を求める.

8 行目の delete_comm_info( )では, 5 行目で保存した子配置 i のスワップランクの通信処理における $\sum(hop_{i,j} \cdot size_{i,j})$ と $link$ の値を network_table から削除する.

スワップランクにおける通信処理の計算では, スワップランクが送信元ランクである場合と, 送信先ランクである場合について計算を行う必要がある. あるスワップランクが送信先ランクである場合 $\sum(hop_{i,j} \cdot size_{i,j})$ の計算量は $O(N \times N^{1/dim})$ となり, 送信元ランクである場合の $\sum(hop_{i,j} \cdot size_{i,j})$ の計算量も $O(N \times N^{1/dim})$ となる. したがって, 全スワップランクにおける $\sum(hop_{i,j} \cdot size_{i,j})$ の計算量は, スワップランク数 $\times 2 \times O(N \times N^{1/dim})$ となる. 高速化前の計算量は $N \times O(N \times N^{1/dim})$ であることから, これにより計算量は (2×スワップランク数)/ N に削減されたことになる. 4,096 ランクの Allgather bruck アルゴリズムの場合, スワップランク数を 6 とすると高速化前に比べ 1/341 の計算量となることから, 目標とする値まで実行時間が短縮されることが期待できる. また, $\max_{link}$ を求める処理ではスワップランクの通信処理のみ考慮すればよいので, network_table の検索回数は $\sum(hop_{i,j} \cdot size_{i,j})$ の計算量と同様のスワップランク数 $\times 2 \times O(N \times N^{1/dim})$ となる.

```
1: init_comm_info( network_table,
2:                 親配置の全ランク,
3:                 親配置のスワップランク );
4: for( i = 0; i < N_child; i++ ){
5:     add_comm_info( network_table,
6:                    子配置 i のスワップランク );
7:     eval( network_table );
8:     delete_comm_info(  network_table,
9:                        子配置 i のスワップランク );
10: }
```

図 1.高速化した子配置評価処理の疑似コード

また, 親配置は一世代前の子配置の中から選ばれるか, 一世代前の親配置から変わらないかのどちらかになることから, 1 行目の init_comm_info( )における全ランクの通信処理の計算では以下のようにすることで計算量を削減する.

一世代前の子配置の中から選ばれた場合

選ばれた一世代前の子配置について, 5 行目の add_comm_info( )で保存したスワップランクの $\sum(hop_{i,j} \cdot size_{i,j})$ と $link$ の値を一世代前の network_table に追加する.

一世代前の親配置と変わらなかった場合

一世代前の親配置について, 1 行目の init_comm_info( )で保存したスワップランクの $\sum(hop_{i,j} \cdot size_{i,j})$ と $link$ の値を一世代前の network_table に追加する.

これにより, ステップ 1 のネットワーク状態テーブルの初期化処理の計算量は $\sum(hop_{i,j} \cdot size_{i,j})$ の計算量と同じスワップランク数 $\times 2 \times O(N \times N^{1/dim})$ となる.

3.5.2 送信先ランクと送信サイズの検索処理の高速化

送信先ランクと送信サイズの検索処理の高速化では, 図 2 に示す 5 種類のランクリストとサイズリストを用意した. 図 2(a)の DstList は各ランクにおける送信先ランクをまとめたリストであり, 図 2(b)の NumDstList は各ランクにおける送信先ランクの数をまとめたものである. 図 2(c)の SizeList は送信先ランクへの送信サイズをまとめたリストであり, DstList と対応する. また, 図 1 の add_comm_info( )では送信元ランクも求める必要があるため, ランクごとの送信元ランクをまとめた図 2(d)の SrcList と, 送信元ランクの数をまとめた図 2(e)の NumSrcList も用意した.

あるランク i において, 送信先ランクの数が $N_{dst(i)}$ のとき, ランク i のすべての送信先ランクと送信サイズを検索するためには, NumDstList を 1 回, DstList と

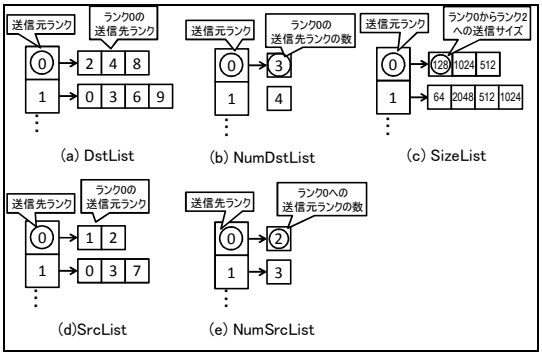

図 2.ランクリストとサイズリスト

SizeList をそれぞれ $N_{dst(i)}$ 回参照することになる. したがって, ランク数 N のランク配置を評価する際には, NumDstList を N 回, DstList と SizeList をそれぞれ $\sum_{i=0}^{N} N_{dst(i)}$ 回参照することになる. また, ランク数 N における図 2 の各リストの要素数は, DstList と SizeList, SrcList では $\sum_{i=0}^{N} N_{dst(i)}$ に, NumDstList と NumSrcList は N となる.

4. 高速化した RMATT の評価

本章では高速化した RMATT について, 計算処理の削減による高速化の効果と, 実アプリケーションにおけるチューニング効果を評価する.

4.1 プロトタイプ版 RMATT との比較

4.1.1 評価目的

3 章の高速化を行った RMATT について, プロトタイプ版 RMATT と実行時間を比較し, 目標を達しているか確認する.

4.1.2 評価目的

表 4 で示した Allgather bruck アルゴリズムに対し, 3 章の高速化を行った RMATT を適用し, 実行時間を測定する. なお BISEM の高速化では, BISEM 実行後の評価式(1)の評価値が数%落ちるまで反復回数を減らしている. RMATT の実行には表 5 の計算機を 1 台用いた.

4.1.3 評価結果

表 4 Allgather bruck アルゴリズム

ランク数	4,096 ランク
ネットワーク構成	16x16x16 ノード 3D Torus

表 5 実行環境

CPU	Intel Core i7 940 2.93GHz×2
メモリ	6GB

測定を行った結果, プロトタイプ版 RMATT の実行結果と同程度の評価値となるまでに, BISEM の実行に 44 秒, OPTIM の実行に 400 秒を要した. プロトタイプ版では 16 時間かかっていたが, 高速化により 7.4 分で終了したことから, 目標を達したといえる.

4.2 計算量削減による効果の確認

4.2.1 評価目的

OPTIM におけるランク配置評価処理の計算量の削減と, 送信先ランクと送信サイズの参照回数の削減について, それぞれの効果を確認する.

4.2.2 評価方法

測定対象となる OPTIM を表 6 に示す. 高速化 1 ではランク配置評価処理の計算量削減による高速化の効果を, 高速化 2 では送信先ランクと送信サイズの参照回数とメモリ量の削減による効果を, 高速化 3 では両者を合わせたことによる効果を確認する. 表 6 の各 OPTIM を表 4 の Allgather bruck アルゴリズムに適用し, OPTIM が 10 世代経過するまでの実行時間を測定する. 高速化なしの OPTIM では実行が完了するまでに非常に長い時間がかかることが予想される. いずれの OPTIM も世代あたりの実行時間は世代が進んでもほぼ一定であることから, 測定は 10 世代までとした. OPTIM の実行には表 5 の計算機を 1 台用いた.

4.2.3 評価結果

表 6 の各 OPTIM について, 10 世代経過するまでの実行時間を表 7 に示す. 高速化 1 の実行時間は高速化 2 より大幅に短縮されていることから, ランク配置評価処理における計算量の削減が OPTIM の高速化に大きな効果があったことがわかる.

表 6 高速化した OPTIM

高速化 1	評価式(1)の計算量を削減
高速化 2	送信先ランクと送信サイズの参照回数を削減
高速化 3	高速化 1 と高速化 2 を合わせたもの
高速化なし	高速化前の RMATT

表 7 10 世代の実行時間

	10 世代の実行時間(秒)
高速化 1	4
高速化 2	560
高速化 3	1
高速化なし	773

4.2.4 考察

高速化 1 による効果

ランク配置評価処理の計算量削減による高速化の効果を確認するために，表 6 の高速化 1 と高速化なしにおけるランク配置評価処理の実行時間を測定し，比較した．測定ではランク配置評価処理 1 回あたりの $\sum(hop_{i,j} \cdot size_{i,j})$ の計算時間と，$\max_{link}$ の検索に要した時間を測定した．また，$\max_{link}$ を検索するために network_table を参照した回数も測定した．

測定結果を表 8 に示す．高速化後の $\sum(hop_{i,j} \cdot size_{i,j})$ の計算量は，高速化前に比べ 1/341 となっている．これに対し，高速化 1 の実行時間は高速化なしに比べ 1/351 となっている．したがって，$\sum(hop_{i,j} \cdot size_{i,j})$ における計算量の削減の割合に応じて計算時間も削減されていることがわかる．また，$\max_{link}$ を求めるための network_table の参照回数は，高速化 1 は高速化なしに比べ 1/18,355 となっていることに対し，$\max_{link}$ の検索時間は 1/5,922 となっている．この違いについてメモリの参照処理に原因がある可能性があるが，現在詳細に調査中である．

高速化 2 による効果

送信先ランクと送信サイズの参照回数の削減による高速化の効果を確認するため，表 6 の高速化 2 におけるランクリストとサイズリスト(図 2)と，高速化なしにおける送信サイズテーブルについて，それぞれ参照回数と参照に要した時間を測定し，比較した．また，メモリサイズの削減量も確認するため，高速化 2 におけるランクリストとサイズリストの要素数と，高速化なしにおける送信サイズテーブルの要素数を求め，比較した．

表 8 ランク配置評価処理の実行時間

	高速化 1	高速化なし
$\sum(hop_{i,j} \cdot size_{i,j})$ の実行時間(マイクロ秒)	28	9,845
$\max_{link}$ の計算時間(マイクロ秒)	3	17,768
network_table の参照回数	914	16,777,216

表 9 送信先ランクと送信サイズの検索処理

	高速化 2	高速化なし
参照回数	102,400	16,777,216
参照に要した時間(マイクロ秒)	2,344	29,944
要素数	155,648	16,777,216

測定結果を表 9 に示す．表 4 の Allgather bruck アルゴリズムはランク数が 4,096，各ランクの送信先ランク数はすべて $\log_2(4,096)=12$ となることから，高速化 2 におけるランクリストとサイズリストの参照回数と，高速化なしにおける送信サイズテーブルの参照回数は計算により求めることができ，表 9 の測定結果と同じであることが確認できた．しかし，高速化 2 の参照回数は高速化なしに比べ 1/164 となっているのに対し，参照に要した時間は 1/12 でしかない．これもメモリの参照処理に原因がある可能性があるが，現在調査中である．

4.3 大規模アプリケーションへの適用

4.3.1 評価目的と評価方法

高速化した RMATT における大規模なアプリケーションでの実行時間を測定するため，大規模な Allgather bruck アルゴリズムに高速化 3(表 6)の高速化を行った RMATT を適用する．ネットワーク構成は 128×128(16,384)ノードの 2D Torus とし，最適化が終了するまでの時間を測定した．OPTIM の終了条件は，OPTIM 実行開始時に比べ評価値が 1%以上下がらない状態が 1,000 世代続いた場合とした．RMATT の実行には，表 5 の計算機を 1 台用いた．

4.3.2 評価結果

BISEM の実行時間は 1,225 秒，OPTIM の実行時間は 960 秒であった．36.4 分で RMATT の実行が終了していることから，目標を達しているといえる．

4.4 NAS Parallel Benchmark への適用

4.4.1 評価目的と評価方法

高速化した RMATT について，実際のアプリケーシ

表 10 NAS Parallel Benchmark の問題設定

問題	CG
NPROCS	1,024
SIZE	B

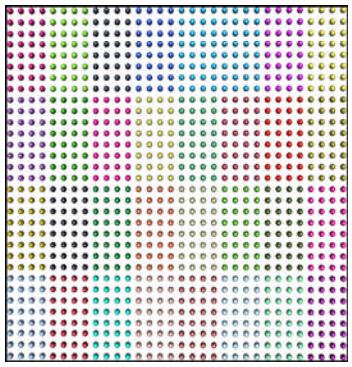

図 4 RMATT による NPB CG のランク配置

ョンと実環境上でのチューニング効果を確認するために，NAS Parallel Benchmark[11]に対して高速化3(表 6)の RMATT を用いランク配置最適化を行い，現在開発中の京コンピュータ[5]上で実行時間を測定した．NPB の問題設定を表 10 に示す．京コンピュータ上におけるネットワークの構成は 32×32(1,024)ノードの 2D Torus とした．RMATT の実行には，表 5 の計算機を 1 台用いた．

4.4.2 評価結果

OPTIM の実行時間は 240 秒，BISEM の実行時間は 30 秒であった．4.5 分で最適化を終了していることから，目標を達しているといえる．

高速化 3 の RMATT によるランク配置最適化されたネットワークを図 4 に示す．各ランクは 32 ランクごとに 1 色ずつ色分けされている．RMATT によるランク最適化では 32 ランクずつ長方形に分けられ，各長方形は規則的に並んでいることがわかる．

ランク配置最適化を行わない場合と，高速化した RMATT により最適化した場合との NPB CG の実行時間を表 11 に示す．表 11 から，高速化した RMATT により最適化された NPB CG は，最適化なしの場合に比べ実行時間が 7%短縮されていることがわかる．CG では大部分がランク内の計算処理であるが，通信処理を十分短縮することにより実行時間を短縮することができたと考えらえる．

表 11 NPB CG の実行結果

	実行時間(秒)
ランク配置最適化なし	1.33
高速化 3 の RMATT による最適化	1.24

5. おわりに

RMATT は MPI アプリケーションにおけるランク配置を最適化することで通信処理時間を短縮することができるが，実行に長時間を要することが問題であった．この問題を解決するため，変更されたランクの通信処理のみを再計算する方法を開発した．これにより，Allgather bruck アルゴリズムにおいて 16 時間かかっていたチューニング時間を 7.4 分に短縮でき，16,384 ランクでも 36.4 分で実行できることも確認した．さらに，高速化した RMATT を用いて NAS Parallel Benchmark におけるクラス B，プロセス数 1,024 の CG に本 RMATT を適用し，京コンピュータ上において CG の実行時間を 7%削減することができた．

今後はさまざまな実アプリケーションや実機上で数万ランク規模の評価を行い，最適化性能の向上を図るとともに，一般への提供を目標にユーザビリティの充実も行っていく予定である．

参 考 文 献

[1] http://www.top500.org/.
[2] 松岡 聡，遠藤 敏夫，丸山 直也，佐藤 仁，滝澤 真一朗，"TSUBAME 2.0 の全貌"，TSUBAME e-Science Journal，(1):2-4，2010.
[3] Hiroshi Nakashima，"T2K Open Supercomputer: Inter-University and Inter-Disciplinary Collaboration on the New Generation Supercomputer."Intl. Conf. Informatics Education and Research for Knowledge-Circulating Society (ICKS'08)，Kyoto，Japan，Jan，2008.
[4] http://accc.riken.jp/ricc.html.
[5] http://www.nsc.riken.jp/index_j.html.
[6] Alam，S.，Barrett，R.，Bast，M.，Fahey，M.R.，Kuehn，J.，McCurdy，C.，Rogers，J.，Roth，P.，Sankaran，R.，Vetter，J.S.，Worley，P.，Yu，W.，"Early evaluation of IBM BlueGene/P，" High Performance Computing，Networking，Storage and Analysis，2008. SC 2008. International Conference for，pp.1-12，Nov，2008.
[7] http://www.nccs.gov/.
[8] 今出広明，平本新哉，三浦健一，住元真司，”大規模計算環境のためのランク配置最適化手法 RMATT”，SACSIS2011，pp.340-347，Mar，2011.
[9] Scott Kirkpatrick，"Optimization by simulated annealing: Quantitative studies，"Journal of Statistical Physics，Vol.34，Number 5-6，pp.975-986，Mar，1984.
[10] Bruck，J.，Ching-Tien Ho，Kipnis，S.，Upfal，E.，Weathersby，D.，"Efficient Algorithms for All-to-All Communications in Multi-Port Message-Passing Systems，"Parallel and Distributed Systems，IEEE Transactions on，Vol.8，Issue 11，pp.1143-1156，Nov，1997.
[11] Bailey，D，H.，Barszcz，E.，Barton，J.，T.，Browning，D.，S.，Carter.，R.，L.，Fatoohi，R.，A.，et al.，“The NAS Parallel Benchmarks”，the International Journal of Supercomputer Applications，1991.
[12] https://ngarch.isit.or.jp/taas/opennsim/.