Vol.2017-HPC-159 No.1
2017/4/17

京コンピュータにおける2.5次元アルゴリズムを用いた分散並列行列積の実装と評価

椋木 大地[1,a)]　今村 俊幸[1]

概要：分散並列環境における行列積計算の通信削減手法として，2.5 次元アルゴリズムが提案されている．本稿では 2.5 次元アルゴリズムを使用した倍精度の分散並列行列積（2.5D-PDGEMM）を実装し，京コンピュータにおいてその性能を分析した．2.5D-PDGEMM を利用するには行列データを 3 次元形状に分散する必要があるが，我々は 2.5D-PDGEMM を従来型の 2 次元アルゴリズムを採用する PDGEMM の代用とすることを想定し，行列が 2 次元で分散されている状態に対して 2.5 次元アルゴリズムを実行する実装した．京コンピュータの 16384 ノード（131072 コア）を使用した性能評価では，ScaLAPACK の PDGEMM ルーチンを含む従来の 2 次元アルゴリズムによる実装と比べ，行列の再分散コストを考慮しても，我々の実装した 2.5D-PDGEMM が高速となるケースがあり，2.5 次元アルゴリズムが PDGEMM の強スケーリング性を改善するために有効であることが確認された．

1. はじめに

スーパーコンピュータの性能向上に向けて分散並列計算機におけるプロセス数（ノード数）は増加傾向にある．これは 1 ノードあたりの性能向上に限界があり，性能向上を並列数に依存せざるを得ないことが背景にある．このような状況においては，ある一定規模の問題の計算について，並列数を増加させた場合の性能向上（いわゆる強スケーリングにおけるスケーラビリティ）が重要となる．強スケーリング性能を考える上で問題となるのは，1 ノードあたりの計算量が小さくなりすぎることによって，相対的に通信コスト，特に通信レイテンシのコストが性能のボトルネックとなることである．

このような問題は，$O(n^2)$ のデータアクセスに対して $O(n^3)$ の演算を行い，演算インテンシブな処理とみなされる行列積においても生じうる．分散並列環境における行列積（ScaLAPACK[1] における PDGEMM ルーチン等）の場合，1 ノードあたりの問題サイズが十分であれば，実行時間の大半を演算コストが占めることになるため，性能は計算機の演算性能に律速する．しかしある問題サイズの計算に対してノード数を増やしていくと，やがて実行時間に占める通信コストの割合が増加していくようになる．

このような問題は次世代エクサスケール世代の計算機ではさらに顕著となることが懸念されており，各種の通信削減・通信回避（Communication Avoiding, CA）手法が研究されている．行列積については，Solomonik と Demmel が 2011 年に提案した 2.5 次元アルゴリズム [2] が知られており，幾つかの研究において理論的なコストや性能モデル，性能が報告されている [2] [3] [4] [5].

我々は 2.5 次元アルゴリズムを使用した並列行列積（2.5D-PDGEMM）を実装し，理化学研究所・計算科学研究機構に設置されているスーパーコンピュータ「京」を用いてその性能を分析した．2.5 次元アルゴリズムは行列データが 3 次元形状に分散されている必要がある．本研究では，2.5 次元アルゴリズムを従来型の 2 次元アルゴリズムを採用する PDGEMM の代用とすることを想定し，行列データが 2 次元プロセスグリッド上に 2 次元分散されている状態から行列を再分散して，2.5 次元アルゴリズムを実行する実装とした．最大 16384 ノード（131072 コア）を用いた性能評価の結果，問題サイズに対してノード数が十分に多いケースにおいては，再分散のコストを考慮しても ScaLAPACK の PDGEMM を含む従来の 2 次元アルゴリズムを用いた実装を上回る性能を実現できることが確認された．本稿ではプロファイラによる実行時間の内訳を示しながら，2.5 次元実装の有効性と課題を議論する．

2. 2.5 次元行列積

2.5 次元行列積の詳細および理論的なコストについては Solomonik と Demmel の論文 [2] に詳しく述べられているため，ここではその概念の簡単な説明にとどめる．

[1] 理化学研究所 計算科学研究機構
[a)] daichi.mukunoki@riken.jp

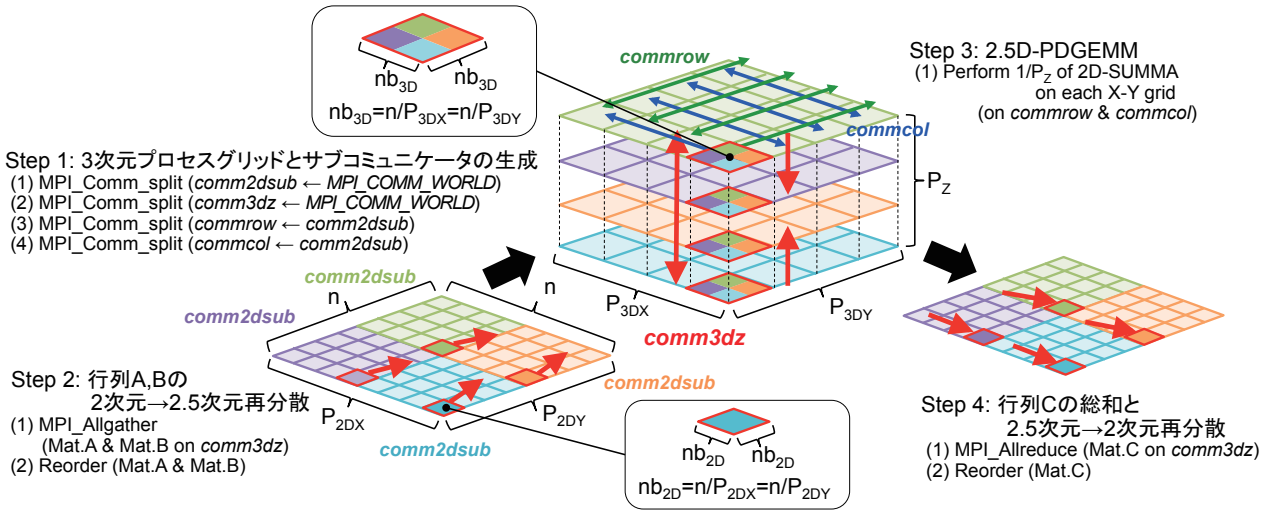

図 1: 実装の概要

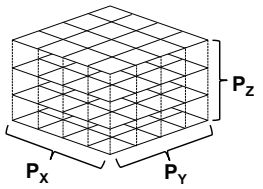

図 2: 3 次元プロセスグリッド

表 1: 2 次元および 2.5 次元行列積のコスト（問題サイズ:$n \times n$，総プロセス数 P，Z 方向のプロセス数 P_Z とした場合）

	2D	2.5D	2.5D/2D
演算	$O(n^3/P)$	$O(n^3/P)$	1
メモリ	$O(n^2/P)$	$O(n^2P_Z/P)$	P_Z
通信バンド幅	$O(n^2/P^{1/2})$	$O(n^2/(P_ZP)^{1/2})$	$1/P_Z{}^{1/2}$
レイテンシ	$O(P^{1/2})$	$O(P^{1/2}/P_Z{}^{3/2})$	$1/P_Z{}^{3/2}$

2.5 次元アルゴリズムでは，**図 2** に示すような 3 次元のプロセスグリッドを想定する．X, Y, Z 方向のプロセス数をそれぞれ P_X，P_Y，P_Z とすると，総プロセス数 P は $P = P_X \times P_Y \times P_Z$ である．ただし $P_Z \leq P^{1/3}$ である．一方，行列データの分散という観点では，従来の 2 次元アルゴリズムが 2 次元プロセスグリッド（$P_X \times P_Y$）上に行列データを 2 次元に分散させて計算を行うのに対し．2.5 次元アルゴリズムでは，$P_X \times P_Y$ に 2 次元分散された行列データを，Z 方向に P_Z だけ積層（複製）する．そして X–Y の各 2 次元プロセスグリッド上では，一般的な 2 次元アルゴリズム（Cannon[6] や SUMMA[7] など）の $1/P_Z$ ステップに相当する部分がそれぞれ実行される．最終的な計算結果は，各階層で計算された計算結果を Z 方向に P_Z プロセス間で総和をとることによって得られる．

表 1 に 2 次元および 2.5 次元行列積の理想的な場合のコストを示す（問題サイズ:$n \times n$，総プロセス数 P，Z 方向のプロセス数 P_Z とし，行列の再分散，Z 方向の通信コストを含まない）．2 次元アルゴリズムと比較すると 2.5 次元アルゴリズムは，演算量は変わらないがメモリ要求量が増大し，一方で通信コストを削減することができる．

3. 実装

ScaLAPACK の PDGEMM と同様に $C = \alpha AB \times \beta C$ を計算し，行列は 2 次元プロセスグリッド上に 2 次元分散されている状態を想定する．簡単のためプロセスグリッドおよび行列サイズは正方に限定し，行列の分散は 2 次元ブロック分割とする．並列行列積アルゴリズムには SUMMA[7] を用いた．SUMMA における通信は行方向および列方向の MPI_Bcast のみで実装できるが，特に京コンピュータにおいては Tofu インターコネクトに最適化された高速な MPI_Bcast[8] が活用できるメリットがある．

C 言語で記述した実装例を**図 3** に示す．このコードは $P_Z = 1$ である場合に余計な操作を行わずに通常の 2 次元アルゴリズムの SUMMA と等価となるように実装されている．このコードにおいて PX2D, PY2D はそれぞれ 2 次元実装における 2 次元プロセスグリッドの X 方向，Y 方向のプロセス数を表している．また PX3D, PY3D, PZ3D はそれぞれ 2.5 次元実装における 3 次元プロセスグリッドの X 方向，Y 方向，Z 方向のプロセス数を表している．**図 1** はこの実装の概略を示したものである（$P = 64$，$P_Z = 4$ の場合を示している）．この図に示したように実装は以下の 4 ステップから構成される．

3.1 3 次元プロセスグリッドとサブコミュニケータの生成

まず 2 次元プロセスグリッドから 3 次元プロセスグリッドを構成し，2.5 次元アルゴリズムを動作させるためのサブコミュニケータを準備する．図 1 の例では $P_Z = 4$ とするため，2 次元プロセスグリッドを 4 つにブロック分割し色分けして，サブコミュニケータ comm2dsub を 4 個作成し，それぞれが 3 次元プロセスグリッドの X–Y 平面となる．各 comm2dsub 上では SUMMA を実行するための行方

```
1  MPI_Comm_rank (MPI_COMM_WORLD, &myprank2d);
2  coords3d[3] = {myprank2d % PX2D, myprank2d / PY2D, 0};
3  if (PZ3D > 1) {
4      coords3d[2] = (coords3d[1] / (PY2D/sqrt(PZ3D))) * sqrt(PZ3D) + (coords3d[0] / (PX2D/sqrt(PZ3D)));
5      MPI_Comm_split (MPI_COMM_WORLD, coords3d[2], myprank2d, &comm2dsub);
6      MPI_Comm_rank (comm2dsub, &myprank2dsub);
7      MPI_Comm_split (MPI_COMM_WORLD, myprank2dsub, myprank2d, &comm3dz);
8      MPI_Comm_rank (comm3dz, &myprank3dz);
9  } else {
10     MPI_Comm_rank (MPI_COMM_WORLD, &myprank2dsub);
11 }
12 myrow = myprank2dsub / PX3D; mycol = myprank2dsub % PY3D;
13 MPI_Comm_split ((PZ3D > 1) ? comm2dsub : MPI_COMM_WORLD, myrow, myprank2dsub, &comm_row);
14 MPI_Comm_split ((PZ3D > 1) ? comm2dsub : MPI_COMM_WORLD, mycol, myprank2dsub, &comm_col);
15 nb2d = n / PY2D;  size2d = nb2d * nb2d; nb3d = n / PX3D; size3d = nb3d * nb3d;
16 csqrt = sqrt (PZ3D);
17 // 2D -> 2.5D: duplicate and reorder
18 if (PZ3D > 1) {
19     MPI_Allgather (matA, size2d, MPI_DOUBLE, matAw3d, size2d, MPI_DOUBLE, comm3dz);
20     MPI_Allgather (matB, size2d, MPI_DOUBLE, matBw3d, size2d, MPI_DOUBLE, comm3dz);
21     for (step = 0; step < PZ3D; step++) {
22         mat_copy (nb2d, matAw3d, 0, nb2d * step, nb2d, matA3d, nb2d * (step / csqrt), nb2d * (step % csqrt), nb3d, 0.);
23         mat_copy (nb2d, matBw3d, 0, nb2d * step, nb2d, matB3d, nb2d * (step / csqrt), nb2d * (step % csqrt), nb3d, 0.);
24     }
25 }
26 // 2.5D-SUMMA
27 nstep = sqrt (P / (PZ3D * PZ3D * PZ3D));
28 for (step = nstep * coords3d[2]; step < nstep + nstep * coords3d[2]; step++) {
29     if (step == mycol) dlacpy_ (N, &size3d, 1, matA3d, &size3d, matAw3d, &size3d);
30     MPI_Bcast (matAw3d, size3d, MPI_DOUBLE, step, comm_row);
31     if (step == myrow) dlacpy_ (N, &size3d, 1, matB3d, &size3d, matBw3d, &size3d);
32     MPI_Bcast (matBw3d, size3d, MPI_DOUBLE, step, comm_col);
33     ibeta = (step == nstep * coords3d[2]) ? ((PZ3D == 1 && coords3d[2] == 0) ? beta : 0.) : 1.;
34     dgemm_ (N, N, &nb3d, &nb3d, &nb3d, alpha, matAw3d, &nb3d, matBw3d, &nb3d, &ibeta, matC3d, &nb3d);
35 }
36 // 2.5D -> 2D: reduce
37 if (PZ3D > 1) {
38     MPI_Allreduce (MPI_IN_PLACE, matC3d, size3d, MPI_DOUBLE, MPI_SUM, comm3dz);
39     mat_copy (nb2d, matC3d, nb2d * (myprank3dz / csqrt), nb2d * (myprank3dz % csqrt), nb3d, matC, 0, 0, ldc, beta);
40 }
```

図 3: C 言語による実装（アルゴリズムの理解に影響しない部分は適宜省略している）

向サブコミュニケータ commrow および列方向 commcol を作成する．また comm2dsub から Z 方向の通信を行うためのサブコミュニケータ comm3dz を生成する．サブコミュニケータの生成は MPI_Comm_split 関数で行われる．

3.2 行列 A,B の 2 次元→ 2.5 次元の再分散

本実装では行列が 2 次元プロセスグリッド上に 2 次元分散されている状態から利用することを想定する．2 次元から 2.5 次元への再分散が必要となるのは行列 A と B である．2.5 次元アルゴリズムでは 1 プロセスが計算を担当する小行列の大きさは P_Z 倍に拡大する．このために comm3dz 上で MPI_Allgather により各プロセスが所持していた小行列を全プロセスが所有するように再分散する．このとき MPI_Allgather を動作させるために行列データを並び替えて一時領域にコピーするための mat_copy 関数を実装した．この関数はコピー元とコピー先の行列について，先頭アドレスから指定要素数分オフセットしてアクセスする．なおこの関数は OpenMP でスレッド並列化している．

3.3 行列積計算部分

2.5 次元アルゴリズムは Z 方向にスタックされた各 2 次元プロセスグリッド（comm2dsub コミュニケータ）において，既存の 2 次元アルゴリズムの一部（$1/P_Z$）がそれぞれ実行される．図 3 の 28 行目から 35 行目が 2.5 次元の SUMMA による計算部分であるが，28 行目の for ループにおいて開始位置・終了位置が Z 次元の位置によって異なる．なお dlacpy は LAPACK[9] が提供するシングルプロセス上で行列コピーを行う関数である．dgemm は BLAS[10] が提供するシングルプロセス上で行列積（$C = \alpha AB + \beta C$）を計算する関数である．

3.4 行列 C の総和と 2.5 次元→ 2 次元の再分散

最後に行列 C についての各 comm2dsub 上における一時計算結果を comm3dz で総和し再分散する．このステップ

表 2: 評価環境（1 ノードあたり）および実行条件

システム	京コンピュータ
プロセッサ	SPARC64 VIIIfx
	8 コア, 2.0 GHz, 128 GFlops（倍精度）
メモリ	16GB, 64GB/s
ネットワーク	Tofu インターコネクト
	6 次元メッシュトーラス（各方向 5GB/s）
言語環境	K-1.2.0-21（2017 年 1 月 10 日リリース）
コンパイラ	mpifccpx
コンパイル設定	-Xg -Kfast,parallel,openmp
	-O3 -MD -SCALAPACK -SSL2BLAMP
MPI 設定	1 プロセス/ノード
OpenMP 設定	8 スレッド/プロセス（1 スレッド/コア）
PJM 設定	#PJM –rsc-list ”node=PX2DxPY2D”
	（PX2D,PY2D は x,y 方向のプロセス数）

は comm3dz における MPI_Allreduce で実現できる．また行列 A,B の 2.5 次元への再分散時と同様に行列の並び替えを行うため mat_copy 関数を実行する．本実装ではこの時に $C = \beta C$ を計算する．

4. 性能評価

4.1 評価環境

京コンピュータにおいて性能を評価した．**表 2** に評価環境および実行条件の詳細を示す．参考として ScaLAPACK の PDGEMM の性能も測定した．ただしブロックサイズを $nb = n/\mathtt{sqrt}(P)$ とし，我々の実装と同様にブロック分割としている．我々の実装において P_Z は 1, 4, 16 の 3 種類のケースを測定した．例えば 16384 ノードの場合，確保されるノードは実行時の node オプションにより”node=128×128” であるが，$P_Z = 4$ では 64×64×4，$P_Z = 16$ では 32×32×16 に相当する擬似的な 3 次元プロセスグリッドが形成される．$P_Z = 1$ では $128 \times 128 \times 1$ となり，余計な操作は一切行わずに通常の 2 次元 SUMMA と等価な処理が行われる．なお $P_Z = 16$ はアルゴリズムの制約（$P_Z \leq P^{1/3}$）により 4096 ノード以上のみで実行可能である．

ルーチンの実行時間の測定時には各 MPI 関数の実行時間の内訳の取得を取得するために富士通詳細プロファイラ（fapp）[11] を適用している．プロファイラは各スレッド間の平均，最大，最小実行時間を返すが，本稿では平均値を結果として採用した．プロファイリングコストは最大でも実行時間の数パーセント程度であった．京コンピュータでは複数回のプログラムの実行時に実行時間が大きくばらつく場合がある．その影響を減らすために，測定対象のルーチンを 3 回実行するプログラムを 3 回実行するジョブスクリプトを実行し，そのうち 2 回目と 3 回目のプログラム実行の中から最良の結果を採用した．

4.2 実験結果

図 4 に，最大 16384 ノードにおける強スケーリング性能（問題サイズ n=32768，理論ピーク演算性能比）を示す．理論ピーク演算性能はノード数× 128GFlops である．破線は MPI サブコミュニケータの生成に関する関数群（MPI_Comm 関数）の実行時間を含まない性能を示している．**図 6** は図 4 の実行時間の内訳を示したものである．“Others”は測定区間の全体の実行時間から，MPI 関数群の実行時間を引いたものである．実際には dgemm_の実行時間が支配的であると考えられる．同様にして，**図 5** に，4096 ノードにおいて問題サイズを変えた場合の性能（理論ピーク演算性能比）を示す．また**図 7** にその実行時間の内訳を示す．

2.5 次元アルゴリズムの有効性が期待される，1 ノードあたりの計算サイズが比較的小さい場合の結果をみると，2.5 次元実装である SUMMA($P_Z = 4$) が 2 次元実装の SUMMA($P_Z = 1$) および ScaLAPACK を上回る性能を示した．我々の実装は 2 次元と 2.5 次元の再分散を行うが，そのコストを含めても 2.5 次元アルゴリズムが有効であることが確認された．一方で MPI_Comm_split のコストが最大で全体の実行時間の 40%近くを占めるケースがあり，無視できないことがわかる．MPI_Comm_split はルーチンを複数回呼び出す場合には 1 度だけ行えば良いため，そのコストを無視できるケースもあると考えられるが，実装上の工夫で MPI_Comm_split の使用を避けるべきであると言える．

図 4 および図 6 に示した n=32768 における強スケーリング性能において，SUMMA($P_Z = 4$) の MPI_Bcast のコストは SUMMA($P_Z = 1$) に対して，約 1/1.6（P=256）から約 1/6.5（P=16384）であった．理論的には表 1 よりバンド幅が 1/2，レイテンシが 1/8 となることが期待されるため，この結果は概ねそれに従ったものであると考えられる．図 5 および図 7 に示した P=4096 のケースにおいても同様に約 1/2.0（n=65536）から約 1/7.8（n=8192）であった．MPI_Comm 関数群のコストを除いた場合の全体の実行時間については，SUMMA($P_Z = 4$) が SUMMA($P_Z = 1$) に対して，n=8192，P=4096 で約 4.7 倍，n=32768，P=16384 で約 3.3 倍の性能向上となった．

一方，SUMMA($P_Z = 16$) に関しては，MPI_Bcast のコストについて期待されるほどの削減効果は得られていない．また 2 次元から 2.5 次元への再分散に必要な MPI_Allgather と結果の総和および再分散に必要な MPI_Allreduce のコストが $P_Z = 4$ の場合と比べて増大したため，結果として全体では $P_Z = 4$ の方が良い性能を示した．

5. 今後の検討課題

以下に我々の今後の検討課題を示す．

- アルゴリズムおよび P_Z の自動選択：2 次元・2.5 次元アルゴリズムの切り替えと，2.5 次元実装において P_Z

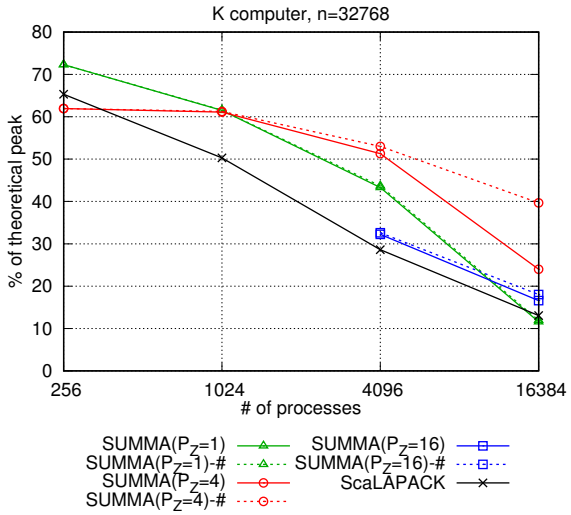

図 4: 京コンピュータ，問題サイズ n=32768 においてノード数を変えた場合の性能（理論ピーク演算性能比，破線（-#）は MPI_Comm 関数群のコストを含まない性能）

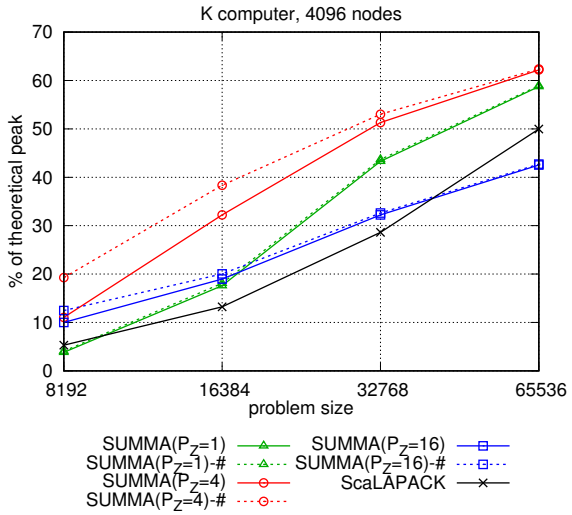

図 5: 京コンピュータ，4096 ノードにおいて問題サイズを変えた場合の性能（理論ピーク演算性能比，破線（-#）は MPI_Comm 関数群のコストを含まない性能）

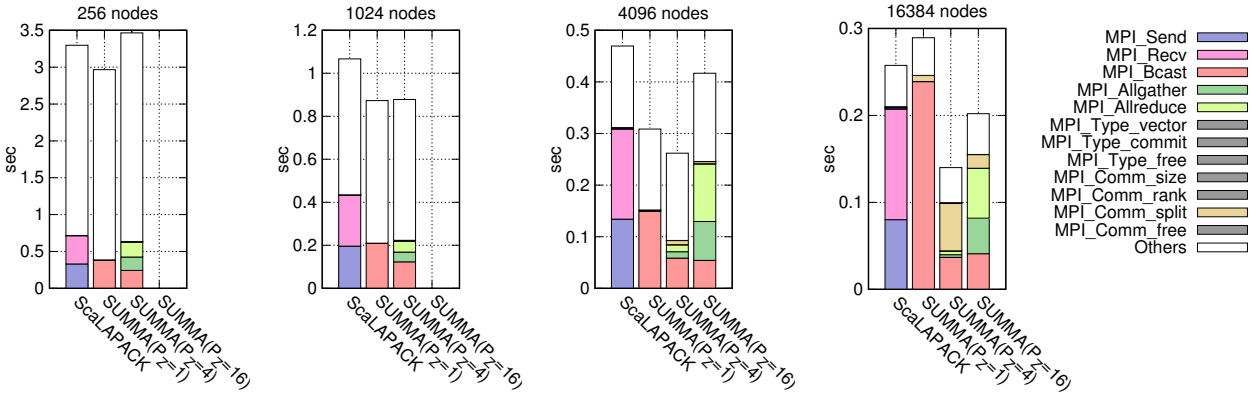

図 6: 京コンピュータ，問題サイズ n=32768 においてノード数を変えた場合の性能（図 4）の実行時間内訳

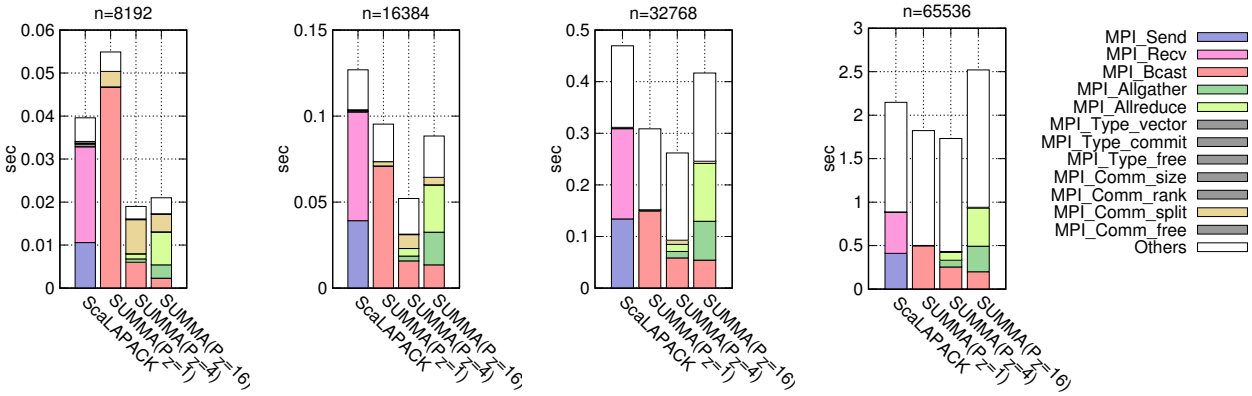

図 7: 京コンピュータ 4096 ノードにおいて問題サイズを変えた場合の性能（図 5）の実行時間内訳

を選択する余地があり，これらの自動選択機能を搭載した実装が求められる．これらの決定にはオフライン自動チューニングのほか，性能モデルに基づくモデルドリブンな手法も適用できると考えられる．

- 物理ノードとプロセスグリッドの対応：割り当ての仕方で性能が変化する可能性がある．本稿で示した $P_Z = 16$ のケースのように Z 方向の通信コストが支配的となるケースでは，それに最適化したプロセス配置を行うことで性能が改善する可能性がある．また京コンピュータでは Tofu インターコネクトに特化したプロセス配置も検討できる．
- MPI_Comm_split の削減：現実装では 2 次元プロセス

グリッドから 2.5 次元に対応するプロセスグリッドを構築するために MPI_Comm_split を用いているが，このコストが無視できないことが明らかとなった．MPI サブコミュニケータを用いない実装を工夫する必要がある．

- メモリ消費量に関する議論 : 2.5 次元アルゴリズムは 2 次元アルゴリズムと比較するとメモリ使用量が P_Z 倍となる．また今回の我々の実装ではさらに一時的な作業領域を必要としており，メモリ消費量について議論が必要である．ただし 2.5 次元アルゴリズムが有効となるのは性能が通信律速となるケースであり，そのようなケースは 1 プロセスあたりの問題サイズは小さいため，メモリ量の圧迫が問題となるケースは限定的ではないかと考えられる．

6. 関連研究

2.5 次元アルゴリズムについては LU 分解等，行列積以外の行列計算に対する研究も行われているが，ここでは行列積についての研究事例を紹介する．

まず文献 [2] は 2.5 次元アルゴリズムの名で 3 次元アルゴリズムを再定義した最初の論文である．Cannon アルゴリズムに基づく実装と BlueGene/P システム最大 16384 ノード（65536 コア）を用いた性能が示されているが，理論的な議論が中心であるため，詳細な性能分析や実装に関する議論は行われていない．文献 [3] では SUMMA および Cannon アルゴリズムについて，通信と演算のオーバーラップを行った 2 次元実装および 2.5 次元実装の性能を Cray XE6 の最大 1024 ノード（24576 コア）を用いて比較している．オーバーラップは通信隠蔽の手法であり，通信削減が行われている 2.5 次元アルゴリズムにおける効果は限定的であることが示されている．またこの論文では実行時間に占める通信と演算の内訳が示されている．文献 [4] では SUMMA に基づく 2.5 次元実装についてそのコストに関する議論と BlueGene/P の最大 2048 ノード（8192 コア）における最大性能を示している．この論文では P_Z を変化させた場合の性能が示されている．文献 [5] では疎行列と密行列の行列積であるが，2.5 次元および似た発想による通信削減実装について性能の分析がなされている．

7. おわりに

本稿では SUMMA アルゴリズムに基づく 2.5D-PDGEMM を実装するとともに，従来の 2D アルゴリズムを採用した PDGEMM の代用とするための 2 次元–2.5 次元の行列データ再分散機能を持った実装を行った．そして京コンピュータ最大 16384 ノードを用いて強スケーリング性能および実行時間の内訳を示した．実験の結果，2 次元–2.5 次元の行列再分散コストを含めても 2.5 次元実装が 2 次元実装を上回る性能を示すことが確認された．一方で 2.5 次元実装が有効となるケースにおいて，MPI サブコミュニケータを生成するための MPI_Comm_split に要するコストが無視できなくなる場合があり，実装上の工夫が必要であることが明らかとなった．今後は実装上の課題を解決するとともに，実アプリケーションにおいて有効性を示したいと考えている．

謝辞 本研究はフラグシップ 2020 エクサスケールコンピューティング開発プロジェクトにおける開発成果の一部であり，本論文の結果は理化学研究所のスーパーコンピュータ「京」を利用して得られたものである（課題番号 : RA000022）．本研究の遂行にあたりご議論・ご意見をいただきました富士通株式会社・山中栄次様，同・末安直樹様，同・竹重和明様（順不同）にお礼申し上げます．

参考文献

[1] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. and Whaley, R. C.: *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA (1997).

[2] Solomonik, E. and Demmel, J.: *Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms*, pp. 90–109 (2011).

[3] Georganas, E., González-Domínguez, J., Solomonik, E., Zheng, Y., Touriño, J. and Yelick, K.: Communication Avoiding and Overlapping for Numerical Linear Algebra, *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pp. 100:1–100:11 (2012).

[4] Schatz, M., Poulson, J. and Geijn, R. V. D.: Parallel Matrix Multiplication: 2D and 3D, *Flame Working Note #62* (2012).

[5] Koanantakool, P., Azad, A., Bulu, A., Morozov, D., Oh, S. Y., Oliker, L. and Yelick, K.: Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication, *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 842–853 (2016).

[6] Cannon, L. E.: A Cellular Computer to Implement the Kalman Filter Algorithm, PhD Thesis, Montana State University (1969).

[7] van de Geijn, R. A. and Watts, J.: SUMMA: Scalable Universal Matrix Multiplication Algorithm, Technical report, Austin, TX, USA (1995).

[8] 住元真司，川島崇裕，志田直之，岡本高幸，三浦健一，宇野篤也，黒川原佳，庄司文由，横川三津夫：「京」のための MPI 通信機構の設計，先進的計算基盤システムシンポジウム論文集，pp. 237–244 (2012).

[9] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A. and Sorensen, D.: *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition (1999).

[10] Lawson, C., Hanson, R., Kincaid, D. and Krogh, F.: Basic Linear Algebra Subprograms for FORTRAN usage, *ACM Trans. Math. Soft., Vol.5*, pp. 308–323 (1979).

[11] Ida, K., Ohno, Y., Inoue, S. and Minami, K.: Performance Profiling and Debugging on the K computer, *Fujitsu Scientific and Technical Journal*, Vol. 48, No. 3, pp. 331–339 (2012).