Vol.2014-HPC-145 No.18
2014/7/29

XcalableMPによる効率的なFFTの実装方法

下坂 健則[1]　村井 均[1]　佐藤 三久[1,2]

概要：大規模 HPC システムに対する代表的なベンチマーク指標の一つである高速フーリエ変換について，PGAS 言語 XcalableMP による効率的な実装方法を検討した．その結果，XcalableMP による実装では，京の 82944 ノード実行で，京向けに最適化された G-FFT の実行性能の 90.6% となる 186.6TFLOPS の性能を得ることができた．また，XMP プログラムに改変したことで，ローカルノードに依存した配列表現を，シンプルで可読性が高いグローバル配列表現にでき，記述性が向上したと考えられる．その他，京の場合，数万ノードの規模になると，プロセス制御に関係する MPI 関数の実行時間が急激に増加する問題があることが分かった．

1. はじめに

高速 Fourier 変換 (FFT) は，大規模並列システムで実行するアプリケーションやベンチマーク課題などで広く利用されており，最適化に関する研究も盛んである．一方で，最適化は，一般にコードが複雑になる場合が多く，開発時の生産性を落とす要因になっている．そのため，従来より低コストでアプリケーションを最適化できるプログラミングモデルおよびその手法が求められている．

近年，MPI [1] を使って書かれたプログラムよりも低コストで並列プログラムを記述できる Partitioned Global Address Space (PGAS) 言語が提案されている．そこでも高生産性，高性能の両立は重要な課題でおり，FFT の PGAS 言語による最適化もその一つに含まれる．

PGAS 言語 XcalableMP(XMP)[4] は，C，Fortran 言語で書かれた逐次プログラムを基に，最小限の指示文を用いて効率よく並列プログラムを開発するための言語である．XMP でも高性能，高生産性の両立は，必須の課題であることから，そのノウハウ蓄積の一環として，様々なベンチマーク課題，およびアプリケーションの評価に取り組んでいる．本稿では，高性能計算分野の代表的なベンチマークの一つである HPC チャレンジベンチマーク [2] に含まれている Global-FFT (G-FFT) について，XMP による効率的な実装方法を述べる．なお，今回は，性能に直結する部分だけを取り上げている．実行性能は，京コンピュータ上での HPC Challenge Award Competition (HPCC) [3] Class1 の G-FFT 82944 ノードにおける性能値 205.9TFLOPS が公表されているため，それを目標とした．毎年 11 月に開催される HPC Challenge Award Competition には，Class1 と Class2 の 2 種類のカテゴリーが存在する．Class1 は高性能を競い，Class2 は実装のエレガントさと高性能を合わせた評価で競う．本稿では，Class2 のルールを基に，XcalableMP による実装を試みている．

以下では，第 2 章で XcalableMP の概要，第 3 章で G-FFT で用いられている six-step FFT アルゴリズムの概要を，第 4 章で XcalableMP による G-FFT の実装と評価，第 5 章でまとめと今後の課題を述べる．

2. XcalableMPの概要

XcalableMP は，PC クラスタコンソーシアムの XcalableMP 規格部会で検討されている並列プログラミング言語である．XMP は C，Fortran 言語に対応し，C 向けの XMP を XMP/C，Fortran 向けを XMP/Forran と呼ぶ．XMP のリファレンスコード Omni XcalableMP は，理化学研究所と筑波大学が共同で開発に取り組んでいる．XMP は，グローバルビューとローカルビューという 2 つのプログラミングモデルを備えていることを特徴とする．グローバルビューでは，逐次プログラムに対して指示文を挿入するだけで，各プロセス内のローカルなアドレスを意識することなく，並列プログラムを開発することができる．ローカルビューは，グローバルビューでは対応しきれない複雑な分散プログラムを書く場合に用いるもので，Fortran2008 の Coarray 機能の仕様を取り込んでいる．XMP では，Coarray 機能を C 言語の場合でも使用可能としている．その他，既存の並列プログラムとの親和性を高めるために，MPI プログラムと XMP プログラム間の相互の呼び出しや，同一関数内に XMP 指示文，OpenMP 指示

[1] 理化学研究所 計算科学研究機構
[2] 筑波大学 計算科学研究センター

文 [5], OpenACC 指示文 [6], および MPI 関数を混在させることが可能である．MPI プログラムから XMP プログラムを呼び出すときには，xmp_init_mpi と xmp_finalize_mpi 関数を，それぞれ MPI プログラム内で XMP の初期化，終了処理ルーチンとして呼び出す必要がある．XMP プログラムから MPI プログラムを呼び出すときには，XMP プログラムで定義したデータマッピング情報を問い合わせ関数を使って取り出し，MPI サブルーチンの実引数に渡すことで接続が可能となる．

3. six-step FFT アルゴリズム

オリジナルの G-FFT は，1 次元 FFT を six-step FFT アルゴリズム [7][9] に基づいて解くように実装されている．six-step FFT アルゴリズムは，(1) に示す 1 次元離散フーリエ変換 (DFT) を，2 次元表現で表した式から導かれる．

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk}, 0 \le k \le n-1 \quad (1)$$

ただし，$\omega = e^{-2\pi i/n}, i = \sqrt{-1}$.

(1) に対して，$n = n_1 \times n_2, j = j_1 + j_2 n_1, k = k_2 + k_1 n_2$ として，x_j, y_k を (2), (3) に示す 2 次元配列に置き換えると (4) が導かれる．

$$x_j = x(j_1, j_2), 0 \le j_1 \le n_1 - 1, 0 \le j_2 \le n_2 - 1 \quad (2)$$

$$y_k = y(k_2, k_1), 0 \le k_1 \le n_1 - 1, 0 \le k_2 \le n_2 - 1 \quad (3)$$

$$y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2} \omega_{n_1}^{j_1 k_1} \quad (4)$$

six-step FFT アルゴリズムは，(4) に対して，以下の手順で計算していくアルゴリズムである．

- 転置：$x_1(j_2, j_1) = x(j_1, j_2)$
- n1 組の n2 点 FFT：
  $x_2(k_2, j_1) = \sum_{j_2=0}^{n_2-1} x_1(j_2, j_1) \omega_{n_2}^{j_2 k_2}$
- ひねり係数の乗算：$x_3(k_2, j_1) = x_2(k_2, j_1) \omega_{n_1 n_2}^{j_1 k_2}$
- 転置：$x_4(j_1, k_2) = x_3(k_2, j_1)$
- n2 組の n1 点 FFT：
  $x_5(k_1, k_2) = \sum_{j_1=0}^{n_1-1} x_4(j_1, k_2) \omega_{n_1}^{j_1 k_1}$
- 転置：$y(k_2, k_1) = x_5(k_1, k_2)$

4. XcalableMP による G-FFT の実装と評価

本稿での XMP による FFT の実装は, FFTE ライブラリである fft-5.0 [8] に対して京コンピュータ向けに最適化されたコード [9] を基に行った．

4.1 実験環境

表 1 に，本稿での測定環境を示す．

実行時には，従来 [10] と同様に，MPI_Alltoall 関数の性能改善に有用なオプション -mca coll_tuned_use_6d_algorithm, -mca coll_tuned_prealloc_size を使用した．coll_tuned_use_6d_algorithm は，京の 6 次元メッシュ・トーラスネットワークの構造を利用して最適化されたアルゴリズムを利用する．coll_tuned_prealloc_size は，集団通信の内部で使用する静的作業領域の大きさを指定するものである．

表 1 京コンピュータでの測定条件

演算性能	128GFLOPS(16GFLOPS x 8 cores)
キャッシュ	L1I : 32 KB (2way) L1D : 32 KB (2way) L2 : Shared 6 MB (12way)
メモリ帯域	64GB/Sec
コンパイラ	K-1.2.0-16-2
最適化オプション	-Kfast,ocl,ilfunc,preex,openmp
実行時オプション	-mca coll_tuned_use_6d_algorithm 1 -mca coll_tuned_prealloc_size xxxx

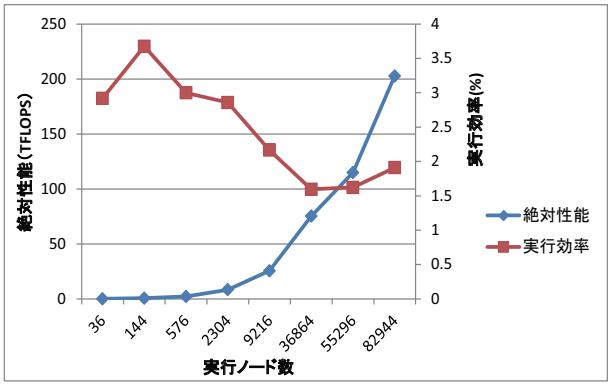

図 1 オリジナルコードの絶対性能と実行時間

図 1 は，オリジナルコードの性能グラフである．82944 ノードの性能が，20.2.7TFLOPS であることから，HPCC Class1 の G-FFT の性能は，図 1 のグラフで再現できていると考えられる．

4.2 XMP による実装の概要

今回は，XMP を使った場合に発生する性能的な問題を中心に解決することを目的にしたため，カーネルルーチンのプロセス間並列化を制御している PZFFT1D0 関数に絞って実装した．オリジナルの PZFFT1D0 関数では，以下の最適化が施されている．

- 転置処理のブロック化
- 複数組の FFT 計算に対する手続き間スレッド並列化
- ひねり係数の乗算とブロック化転置ループの融合

したがって，PZFFT1D0 関数は，以下の順序のアルゴリズムとなっている．

- ブロック化転置
- スレッド並列化された n1 組の n2 点 FFT
- ひねり係数の乗算 + ブロック化転置

- スレッド並列化された n2 組の n1 点 FFT
- ブロック化転置

上記アルゴリズムに対して，実装方法を検討した結果，XMP/Fortran で可能な以下の手法を利用することとした．
(1) MPI プログラムから XMP プログラムへの呼び出し
(2) XMP 指示文と OpenMP 指示文の混在
(3) ブロック化転置処理に配列処理関数を適用
(4) 分散宣言指示文の module 化
(1)，(2) は，[10] のときに適用済みの内容である．(3) は，[10] では，ブロック化転置処理の MPI_Alltoall 関数を使う実装にノード間に跨った配列同士のコピーが可能な gmove 指示文を利用して，転置処理を実装していた．しかし，この場合，入力となるベクトル長相当の作業領域が追加となる．HPCC class2 の G-FFT では，入力となる倍精度複素数ベクトルの要素数を m としたとき，$32m$ バイトという値は，少なくともシステムメモリの 1/4 以上でなければならないという制約がある．gmove 指示文を使う場合には，オリジナルと比べると，一つ多い入力ベクトル長相当の作業領域が必要だったため，この制約を満たせていなかった．そこで，この制約を満たすべく，配列処理関数を使う実装に変更した．

図 2 に，PZFFT1D0 関数を，XMP プログラムに書き換えた例を示す．24-30 行目は，n1 組の n2 点 FFT に対する XMP 指示文と OpenMP 指示文の混在，22 行目はブロック化転置処理に対する配列処理関数の適用，1-3 行目は分散宣言指示文の module 化を示している．

以降では，(1)〜(4) について，説明する．

4.3 MPI プログラムから XMP プログラムへの呼び出し

PZFFT1D0 関数だけを XMP 化する場合，XMP プログラムは，C プログラムから呼び出されることになる．本実装では，XMP プログラムにした PZFFT1D0 関数が分散配列を引数に持つ必要があったため，C プログラムから呼び出すときの実引数は，この分散配列に対応したローカルアドレスを指定している．main プログラムが XMP プログラムでない場合，XMP の初期化関数として，MPI_Init，MPI_Finalize 関数の代わりに，xmp_init_mpi，xmp_finalize_mpi 関数を呼び出す必要がある．main プログラムが XMP プログラムのときには，初期化処理，終了処理は，XMP プログラムで暗黙的に行われるため，MPI_Init，MPI_Finalize 関数，xmp_init_mpi，xmp_finalize_mpi 関数といった関数は必要ない．本実装では，リンク時の容易さを考慮し，main プログラムに XMP プログラムを使うことにした．

4.4 XMP 指示文と OpenMP 指示文の混在

XMP では，OpenMP 指示文との混在を許す仕様となっている．本稿で扱う複数組の FFT では，手続き間並列が必要であるため，該当箇所には，XMP 指示文に加えて，

```
1  module nodep
2  !$XMP nodes p(*)
3  end module nodep
4
5        SUBROUTINE XMP_PZFFT1D0(A,B,CX,CY, &
6                  W,NX,NY,TIMINGS)
7        use nodep
8        ....
9        COMPLEX*16 A(NX,NY)
10       COMPLEX*16 B(NY,NX)
11       COMPLEX*16 CX(*),CY(*)
12       COMPLEX*16 W(NY,NX)
13       ....
14 !$XMP template tx(NX)
15 !$XMP template ty(NY)
16 !$XMP distribute tx(block) onto p
17 !$XMP distribute ty(block) onto p
18 !$XMP align A(*,i) with ty(i)
19 !$XMP align B(*,i) with tx(i)
20 !$XMP align W(*,i) with tx(i)
21
22       CALL XMP_TRANSPOSE(B,A,1)
23       ....
24 !$XMP loop on tx(I)
25 !$OMP parallel do private(ithreads)
26       DO I=1,NX
27          ITHREADS=OMP_GET_THREAD_NUM()
28          CALL ZFFT1D(B(1,I),NY,-1, &
29                CY(ITHREADS*(NY*2+8)+1))
30       END DO
31 .....
```

図 2 XMP による PZFFT1D0 の改変例

OpenMP 指示文が付加されている．Omni XcalableMP では，XMP プログラムの翻訳後にプライベート変数が発生した場合でも，処理系内部で解析しているため，自動的に OpenMP 指示文によるプライベート属性を与える．オリジナルでは，PZFFT1D0 関数全体を並列化領域としているが，Omni XcalableMP は XMP 関数そのものを並列化領域とすることに対応していないことから，並列化領域はサブルーチン内部で適宜使用する方針としている．本対応による性能的な問題は発生しなかった．

4.5 ブロック化転置処理に対する配列処理関数の適用

オリジナルの転置処理では，ブロック化と省メモリ化を考慮した実装が行われ，以下の手順で実装されている．
(1) 配列の線形化
(2) MPI_Alltoall 関数の実行
(3) ローカル配列同士のブロック化転置処理

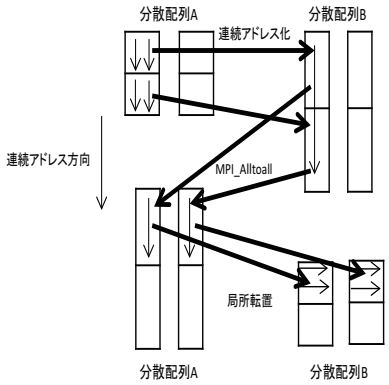

図 3 転置処理手順

オリジナルのブロック化コードは，L1 キャッシュのサイズに合わせてあり，右辺の配列要素をロードしたときに，同時にロードされるキャッシュライン上のデータを再利用している．キャッシュブロッキングのパラメータを NBLK とするとキャッシュブロックサイズの計算式は，NBLK × NBLK × 型サイズ ×2 で表される．オリジナルでは，キャッシュブロッキングのパラメータに 16 を採用している．京の L1 キャッシュサイズは 32KB のため，本研究で対象としている倍精度複素数の FFT の場合には，理論的には 32 まで拡張可能である．ただし，32 とする場合には，L1 キャッシュサイズと等しくなる．本稿では，キャッシュブロックサイズにある程度の汎用性を持たせるため，`xmp_transpose` 関数の実装でも，オリジナルと同様に 16 を使う．

次にオリジナルの省メモリ実装について述べる．`PZFFT1D0` 関数内では，形式的に同じ容量だが異なる形状の 4 つの領域を使って，転置処理を行っている．実際には，4 つの領域は，実体である 2 つの領域のどちらかを指しているため，(1)〜(3) の 3 つの処理に対して，これら 2 つの領域を交互に更新しながら処理を進めている．

このブロック化と省メモリ化をともに XMP プログラムに取り込むため，配列処理関数 `xmp_transpose` を利用することにした．その際，ひねり係数の乗算にあたる処理は，`xmp_transpose` 関数では処理できないため，もともとの six-step アルゴリズムと同様に分離して処理した．

オリジナルの 3 段階の処理手順は，`xmp_transpose` 関数の実装でも同様である．しかし，`xmp_transpose` 関数では，作業領域用の引数を用意せず，作業領域が必要な場合は動的に領域確保する方針としている．オリジナルの省メモリ化を取り込むためには，`xmp_transpose` 関数に入力行列を壊す仕様を入れなければならない．`xmp_transpose` 関数は，入力行列を壊さない仕様をデフォルトとする必要があることから，図 2 の 22 行目に示す第 3 引数にフラグを用意し，第 3 引数が 1 のときには入力行列を保存せず，0 のときには保存する仕様としている．`xmp_transpose` 関数の本仕様を使うことにより，オリジナルと同等のメモリ使用量とすることができ，HPCC Class2 の入力ベクトル長に対する制約を満たすことが可能となる．本対策後のメモリ使用量を表 2 に示す．いずれも HPCC class2 の制約となる入力ベクトル長の下限値を超えていることがわかる．

表 2 入力ベクトル長とメモリ使用量

ノード数	入力ベクトル長	入力ベクトル長 下限値	メモリ使用量 (MiB/node)
36	6635520000	4831838208	11520
144	24186470400	19327352832	10376
576	99532800000	77309411328	10708
2304	398131200000	309237645312	10816
9216	1528823808000	1236950581248	10492
36864	6522981580800	4947802324992	11068
55296	9784472371200	7421703487488	11312
82944	12899450880000	11132555231232	10216

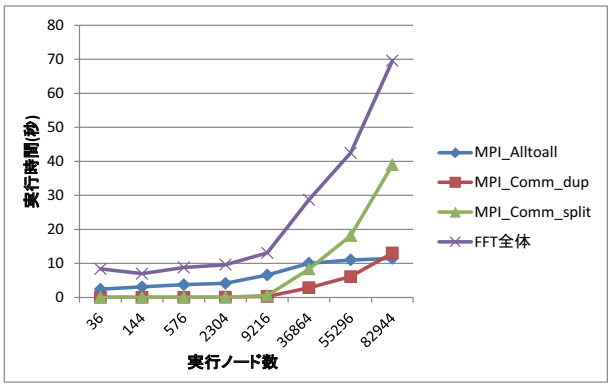

図 4 4.6 節対策前コードに対する主な MPI 関数の実行時間

4.6 分散宣言指示文の module 化と `MPI_Comm_split` の排除

4.2 節 (1)〜(3) まで実装した後で，G-FFT の性能を確認したところ，図 1 のオリジナルの性能とは，9216 ノードから乖離が発生し，82944 ノードでは，5 倍もの差となっていることが分かった．そこで，詳細プロファイラで原因を調べたところ，オリジナルでは出てこない `MPI_Comm_dup` 関数と `MPI_Comm_split` 関数が原因であることが分かった．図 4 は，京の詳細プロファイラで MPI の主要関数の実行時間をグラフにしたものである．

`MPI_Comm_dup` 関数と `MPI_Comm_split` 関数の実行時間の割合は 36864 ノードから急上昇していることがわかる．

従来の XMP の実装では，部分ノード配列を扱う場合を含めて汎用的な実装とするため，分散宣言時に常に `MPI_Comm_split` 関数を実行していた．本稿で扱う FFT は，部分ノード配列を扱うことはないため，部分ノード配列を扱う場合に限り，`MPI_Comm_split` 関数を発行するように実装を変更した．

`MPI_Comm_dup` 関数については，Omni XcalableMP 処理系では，ユーザプログラムが起動するコミュニケータとの

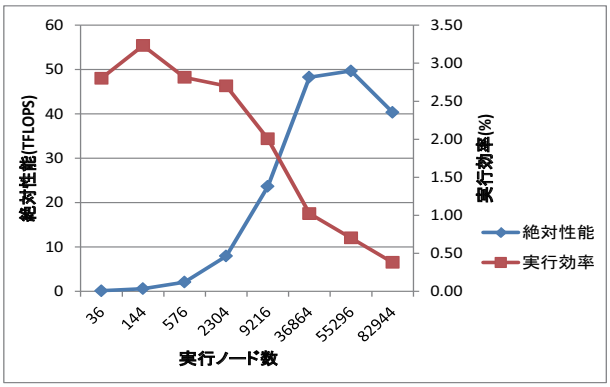

図 5 4.6 節対策前コードの G-FFT の絶対性能と実行効率

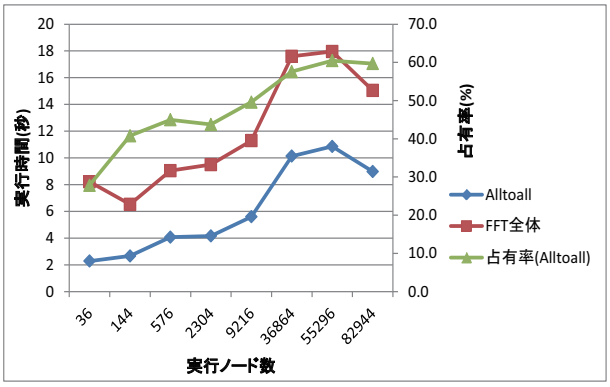

図 6 G-FFT の最終実行時間と MPI_Alltoall の実行時間と占有率

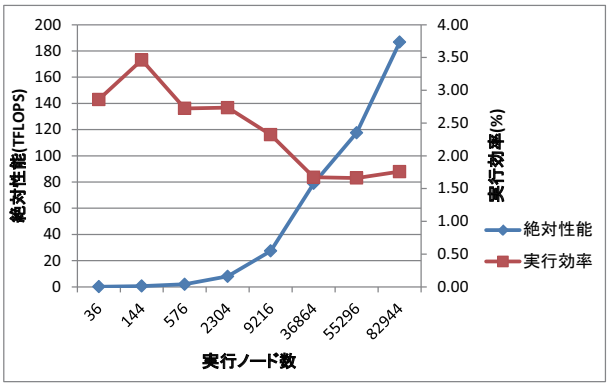

図 7 G-FFT の最終実行性能と実行効率

干渉を避けるため，ノード指示文の処理を実行する際に，最初に MPI_Comm_dup 関数で MPI_COMM_WORLD 相当のコミュニケータを複製する．もともとは，XMP/F サブルーチンにそのままノード指示文を書いていたため，PZFFT1D0 を実行するタイミングで MPI_Comm_dup 関数が発行されていた．そのため，module を使い大域的にノード指示文を使うよう変更することで，PZFFT1D0 を実行するタイミングで，MPI_Comm_dup 関数の実行を回避することができた．

図 4 および図 5 は，本節の対策前の測定値を示している．これらに対して，本節の対策をした結果を，それぞれ図 6 および図 7 に示す．本節の対策により，実行時間の伸びが止まり，実行効率についても 36864 ノード以降，下げ止まっていることが分かる．82944 ノードの性能では，オリジナルの約 9 割の性能を得た．

```
1         SUBROUTINE PZFFT1D0(A,AXPY,AXY,AYXP,B,BXYP,BYPX,
2        1                    BYX,CX,CY,W,NX,NY,NPU,TIMINGS)
3         IMPLICIT REAL*8 (A-H,O-Z)
4         INCLUDE 'mpif.h'
5         INCLUDE 'param.h'
6         COMPLEX*16 A(NX,*),AXPY(NX/NPU,NPU,*),
7        1           AXY(NX/NPU,*),AYXP(NY/NPU,NX/NPU,*)
8         COMPLEX*16 B(NY,*),BXYP(NX/NPU,NY/NPU,*),
9        1           BYPX(NY/NPU,NPU,*),BYX(NY/NPU,*)
10        COMPLEX*16 CX(*),CY(*)
11        COMPLEX*16 W(NY/NPU,NPU,*)
12        ....
```

図 8 PZFFT1D0 の抜粋

4.7 生産性評価

図 8 は，オリジナルの PZFFT1D0 関数の冒頭部分の抜粋である．6-11 行目の配列宣言部を図 2 の 9-12 行目と比較すると，XMP プログラムの方がグローバルなインデックスだけを考えればよい分，シンプルで可読性が高くなっていることが分かる．また，PZFFT1D0 関数そのもののコード量は，XMP プログラムにすることで 121 行から 64 行に半減している．ただし，コード量の減少は，転置処理に組込関数を使ったことの影響が大きい．

4.8 HPCC 計測範囲外を含む MPI 関数情報と job 実行時間の推移

ここまでは，HPCC で対象とする範囲の性能を確認し，性能的に問題のある MPI 関数については，極力性能測定範囲から排除するように最適化を進めていた．本節では，参考までに，ジョブ全体として，MPI 関数の実行時間がどのような振る舞いをしているか確認する．

G-FFT では，プログラム全体として主に以下の処理が行われている．

(1) 入力ベクトル生成 (本領域は作業領域としても利用)
(2) FFT 計算 (G-FFT の性能測定対象箇所)
(3) (2) の計算結果を入力として再 FFT 計算
(4) 入力ベクトルを再生成
(5) (3) の計算結果と入力ベクトル値の比較による結果検証

ジョブ全体の実行時間は，図 9 に示す通り，9216 ノード以降，急激に増加していることがわかる．

図 10 と図 11 は，G-FFT の性能改善前後のジョブ全体に対する主要な MPI 関数の実行時間をグラフにしたものである．なお，図 10，図 11 の判例に書かれている MPI 関数名についている括弧内の数字は，ジョブ内での呼び出し回数を示している．性能改善前は，MPI_Comm_split 関数，MPI_Comm_dup 関数，MPI_Init 関数，MPI_Finalize 関数の影響が大きい．性能改善後は，MPI_Comm_split 関数がな

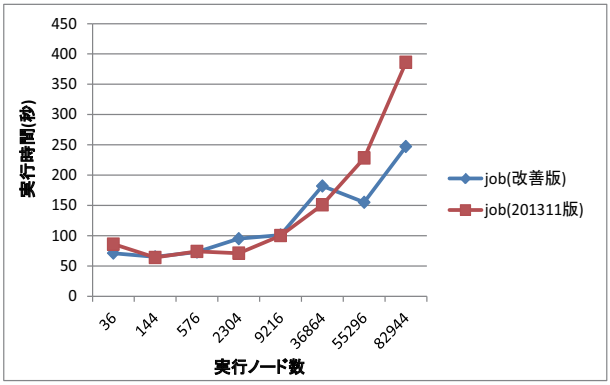

図 9 最終版と 4.6 節対策前コードとのジョブ実行時間の比較

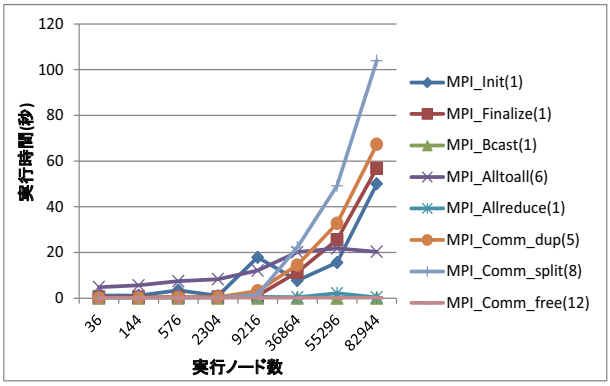

図 10 4.6 節対策前の G-FFT ジョブ全体に対する MPI 関数の実行時間の内訳

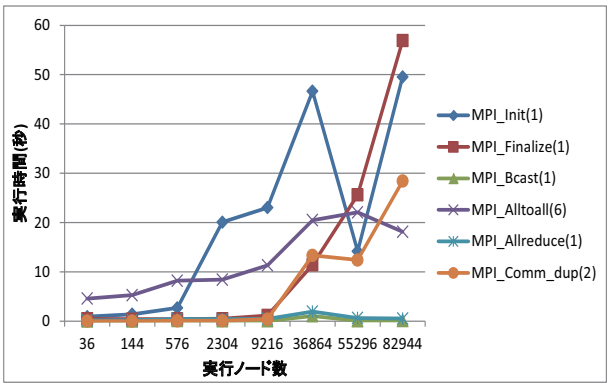

図 11 最終版での G-FFT ジョブ全体に対する MPI 関数の実行時間の内訳

くなり，`MPI_Comm_dup` 関数の実行時間が大きく減ったことから，`MPI_Init` 関数，`MPI_Finalize` 関数の影響が最も大きくなっている．

5. まとめと今後の課題

京向けに最適化された G-FFT のカーネルプログラムに対して，XMP による実装を行った結果，82944 ノードの性能で HPCC Class1 の G-FFT の公表値 [3] 205.9TFLOPS の 90.6% となる 186.6TFLOPS の性能を得ることができた．また，XMP プログラムに改変したことで，ローカルノードに依存した配列表現を，シンプルで可読性が高いグローバル配列表現にでき，記述性が向上したと考えられる．コード量では，XMP による改変の対象としたオリジナルの `PZFFT1D0` 関数に対して，半減させることができた．その他に，京では，プロセス制御に関係する MPI 関数で扱うノード数が数万ノードの規模になると，実行時間が急激に増加することが分かった．今後は，他の方法で実装された FFT についても，XMP による実装を試み，最適化ノウハウの蓄積と XMP 処理系へのフィードバックをしていく必要があると考える．

謝辞 本論文の結果の一部は，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す．

参考文献

[1] MPI Forum: MPI: A Message-Passing Interface StandardVersion 3.0, http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.
[2] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi: Introduction to the HPC Challenge benchmark suite, Technical Report LBNL-59493 (2005).
[3] HPC Challenge : http://www.hpcchallenge.org/index.html.
[4] XcalableMP Specification Working Group: Specification of XcalableMP, Version 1.2, http://www.xcalablemp.org/spec/xmp-spec-1.2.pdf.
[5] OpenMP Architecture Review Board: OpenMP Application Program Interface, http://www.openmp.org/mp-documents/OpenMP3.1.pdf.
[6] OpenACC: http://www.openacc-standard.org/.
[7] D. H. Bailey: FFTs in external or hierarchical memory, The Journal of Supercomputing, vol. 4, pp. 23–35 (1990).
[8] FFTE: http://www.ffte.jp/.
[9] D. Takahashi, A. Uno and M. Yokokawa: An Implementation of Parallel 1-D FFT on the K computer, High Performance Computing and Communication and 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), pp. 344–350 (2012).
[10] M. Nakao, H. Murai, T. Shimosaka, and M. Sato: Productivity and Performance of the HPC Challenge Benchmarks with the XcalableMP PGAS Language, Proceedings of the 7th International Conference on PGAS Programming Models, pp. 157–171 (2013).