富士通 PRIMEHPC FX100 上で利用できる ２つの Coarray Fortran 実装の実用性の評価

岩下英俊[†1]　中尾昌広[†1]　藤井裕也[†2]　村井均[†1]　佐藤三久[†1]

概要：
Coarray Fortran (CAF)は Fortran2008 仕様の一部となっている並列言語である．富士通の Fortran 処理系は，PRIMEHPC FX100 システム向けから CAF 仕様の先行サポートを始めた．また，PGAS 言語 XcalableMP(XMP)は CAF 仕様を包含していて，その実装である理化学研究所と筑波大学による Omni XMP は FX100 システム上でも利用できるようになった．これらの先駆的な実装を用いて，実用性の観点から CAF プログラミングとコンパイラの現状を評価する．EPCC Fortran Coarray micro-benchmark を使った評価では，データのサイズによらず CAF で MPI に近い性能が出せることが示された．また，正確な実装と性能のトレードオフに関して議論された．両コンパイラの比較によって抽出された課題は，両者の今後の改善にフィードバックできる．

キーワード：Fortran，Coarray，コンパイラ，MPI

1. はじめに

Coarray Fortran (CAF)は，Fortran2008 仕様の一部となっている並列言語である．富士通 Fortran は，PRIMEHPC FX100 システム向けから，Fortran2008 仕様の中で最初に CAF 関連仕様を実装しリリースしている．また，PGAS 言語 XcalableMP（XMP）[1]は CAF 仕様を包含していて，その実装である理化学研究所と筑波大学による Omni XMP は FX100 システム上でも利用できるようになった．これらの先駆的な実装を用いて, 実用性の観点から CAF プログラミングとコンパイラの現状を評価し，今後の課題について議論する．

本稿は, FX100 上での富士通 Fortran の CAF 機能（以降，FJ CAF と呼ぶ）と，Omni XMP の CAF トランスレータの実装[2]（以降，Omni CAF）の評価を通して CAF 言語を評価し，課題と将来性について議論する．本稿の構成は以下の通りである．2 章で CAF の文法と FJ CAF，Omni CAF について紹介し，3 章で評価に使用した理化学研究所の PRINEHPC FX100 システムとベンチマークテストセットを紹介する．評価は，4 章で基本性能について，5 章でアプリケーションプログラムの開発効率と性能について．FJ CAF と Omni CAF を比較しながら評価する．6 章では派生した課題について議論し，7 章で関連研究を紹介し，8 章をまとめとする．

2. CAF 文法と実装

2.1 CAF の基本

OpenMP や XcalableMP 指示文のように, 全体の仕事を分割するという考え方（グローバルビュー）ではなく，MPI のように，個々の実行手続と互いの通信・同期を記述するという考え方（ローカルビュー）である．そのため MPI プログラムと親和性が高い．CAF プログラミングは，MPI プログラムからの等価変換するアプローチが有効である[3]．

CAF の並列処理単位はイメージと呼ばれる．FJ, Omni ともイメージは MPI のプロセスに 1 対 1 に対応付けられ，そのIDであるイメージインデックスはMPI_COMM_WORLD に対するランク番号に 1 を足した値である．

CAF では，coarray として宣言された変数について，他のイメージからの参照と定義が許される．coaray 変数の宣言は，従来の変数の宣言に codimension の宣言を付加したものである．coarray 変数は save 属性をもって静的に割付けられるか, allocatable 属性をもって実行時に ALLOCATE 文で割付けられる．実装ではどちらの場合も割付け時に通信ライブラリに登録し，片側通信が可能な状態にする．イメージ k の coarray データ a(i,j)を参照するには a(i,j)[k]と記述する．この参照の実行によって起こる通信を以下では GET 通信と呼ぶ. イメージ k の coarray データ a(i,j)の値を val に更新するには代入文を使って

```
a(i,j)[k] = val
```

と記述する．この定義の実行によって起こる通信を以下では PUT 通信と呼ぶ.

2.2 FJ CAF の実装の特徴

通信ライブラリとして，片側通信の一部には ARMCI を使い，集団通信には富士通 MPI を使っている[4]．ARMCI は Tofu ライブラリを介して FX100 の Tofu ネットワークを効率よくアクセスするように実装されている．

2.3 Omni CAF の実装の特徴

GASNet または FJ-RDMA または MPI3 を片側通信のための通信ライブラリとして使用するが，FX100 上の実装では FJ-RDMA を使用する．FJ-RDMA とは「拡張 RDMA インタフェース」と呼ばれる富士通 MPI の独自拡張機能であり，Tofu ネットワーク上の片側通信をサポートする．

†1 理化学研究所 計算科学研究機構
†2 富士通(株) 次世代テクニカルコンピューティング開発本部 言語開発統括部

ⓒ2017 Information Processing Society of Japan

3. 評価環境

3.1 理化学研究所 HOKUSAI GreatWave システム

超並列演算システム, アプリケーション演算サーバ群（大容量メモリ演算サーバと GPU 演算サーバ），フロントエンドサーバとストレージ群から構成されるシステムである．本稿の評価では計算に超並列演算システムを使用した．これは以下の諸元をもつ富士通 PRIMEHPC FX100（以降 FX100）である．

- CPU: SPARC64XIfx(1.975GHz)×1080 個. 16SIMD×32 コア．ノード当り 1CPU, 理論ピーク 1.092PFLOPS
- メモリ: CPU 当り 32GB. CPU 当りバンド幅 480GB/s
- インターコネクト: Tofu2. ノード間 12.5GB/s×双方向

4. 基本性能の評価

以下では，特に断りのない限り，富士通 Fortran と Omni XMP のコマンドと使用した最適化オプションはそれぞれ以下の通りである．

```
mpifrtpx -Cpp -Ncoarray -Kfast,reduction
xmpf90 -cpp -Kfast,reduction
```

Omni XMP に与えた-K オプションは, CAF トランスレータによる CAF プログラムからのソース-to-ソース変換の後，後続する富士通 Fortran にそのまま渡される．

本節の基本性能の評価には，EPCC Fortran Coarray micro-benchmark（以降 CAF ベンチマーク）を使用した[5]．CAF ベンチマークは CAF の基本的な性能を測定するベンチマークセットであり，cafpt2pt，cafsync と cafhalo の 3 つの部分から成る[6]．cafpt2pt は，様々なパターンの ping-pong 通信について，データサイズまたは粒度を変えてレイテンシ（1 ping-pong 当りの実行時間の半分）とバンド幅（1 秒当りの通信量）を測定し，PUT 通信，GET 通信と MPI の send/recv 通信を比較することができる．cafsync は SYNC ALL 文と SYNC IMAGES 文の性能を測定する．cafhalo はステンシル計算の通信性能を測定するが，本稿では取り上げない．

4.1 Ping-Pong 通信によるレイテンシ／バンド幅の比較

図 1 は，富士通（FJ）と Omni の CAF コンパイラと，富士通 MPI による，ping-pong 通信の性能を比較している．CAF には PUT で記述した版と GET で記述した版がある．同図(a)はレイテンシ，(b)はバンド幅を示す．横軸はバイト単位のデータ量を示す．通信の対象は倍精度実数型（8 バイト）の 1 次元配列であり，通信範囲は配列の先頭から長さ datasize とし，これを 8B から 32GB まで変化させた．ジョブは同一ノード内の 4 プロセスによる実行である．

(1) MPI Send/Recv（比較のため）

CAF ベンチマークの MPI の測定では，rank 番号 0 を image1，3 を image2 とし，ping-pong を以下の 2 ステップで実行している．

1. image1 で MPI_Send，image2 で MPI_Recv を実行
2. image2 で MPI_Send，image1 で MPI_Recv を実行

図 1(a)の折れ線グラフが示すように，MPI のレイテンシは 2KB 以下と 4KB 以上で変化し，2KB 以下では CAF の実装に比べて小さい（1.3～7.0 倍良い）．これは eager 通信が有効に働いているためであると考えられる．eager 通信は，送信者が送信先アドレス（固定的なバッファ領域）をあらかじめ知っていてアドレスの交換なしで無条件に送信できるため性能が良いが，メモリ量の制約からそのようなバッファは大きく確保できないので，通信データが大きいときには利用できない．4KB 以上で使われる rendezvous 通信では，バッファを介することなく目的のアドレスに直接送信するが，送信前に受信ノードからアドレスの通知を受けて通信ライブラリに登録する必要があるため，レイテンシが大きくなる．

MPI の実装として，eager と rendezvous 通信の閾値にはチューニングの余地がある．

(2) Omni XMP による PUT

CAF ベンチマークの PUT の測定では, イメージ番号 1 を image1，4 を image2 とし，ping-pong を以下の 4 ステップで実行している．

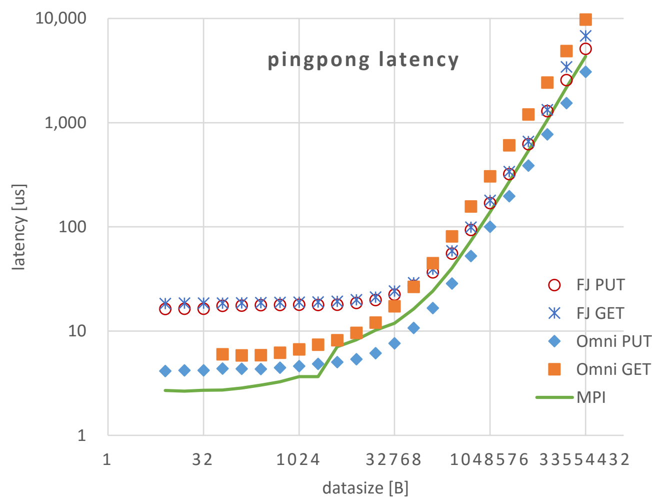
1. image1 で x(1:ndata)[image2] = x(1:ndata)
2. image1 と 2 で相互に SYNC IMAGES
3. image2 で x(1:ndata)[image1] = x(1:ndata)
4. image1 と 2 で相互に SYNC IMAGES

Omni XMP トランスレータは，ステップ 1 の代入文を以下のような実行時ライブラリの呼び出しに変換する．

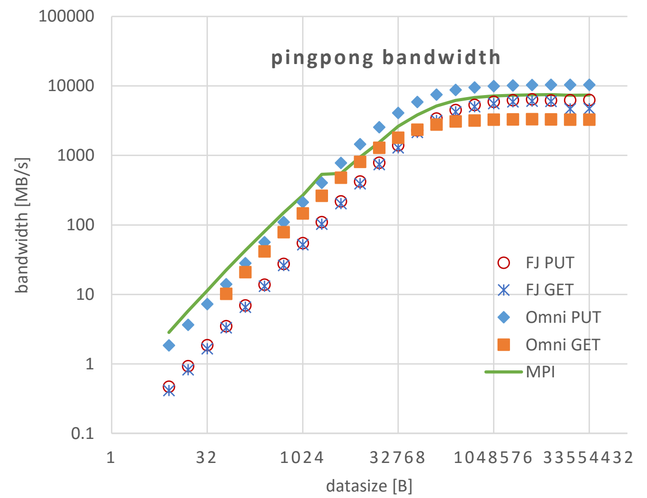
```
CALL xmpf_coarray_put_generic (
     xmpf_descptr_x ,      !! coarray 変数 x の記述子
     image2 ,              !! 送信先イメージ番号
     x ( 1 : ndata ) ,     !! 左辺の変数（送信先）
     x ( 1 : ndata ) )     !! 右辺式（送信データ）
```

実行時ライブラリの入口は Fortran 90 で書かれ形状引継ぎ配列を引数としているため，右辺式（第 4 引数）は余計な copy-in/out を生じることなく実行時ライブラリに伝えられる. 実行時ライブラリは, 割付け時に登録されて記述子（第 1 引数）が保持する coarray 変数 x のアドレスと，第 3 引数から得られる部分配列の形状から，送信先に問い合わせることなく目的のアドレスと通信パターンを知ることができる．データが 4GB 以上のとき MPI に対して 28～40% レイテンシが小さい理由は，MPI の rendezvous 通信で行われている目的のアドレスの受信と登録が必要ないことであると考えられる．

一方で, 2KB 以下で MPI と比べてレイテンシが 26～60% 大きい原因は，通信の完了確認のコストであると考えられる．eager 通信では受信側は MPI_Recv のライブラリの実行の中で完了を待つことができるが，PUT 通信では受信側は通信に全く関与しないので，送信側で何らかの手段で確認し，さらにそれを受信側に知らせる必要がある．Omni XMP



(a) レイテンシ



(b) バンド幅

図 1. PUT/GET と MPI send/recv の ping-pong 性能（同一ノード内）

の現在の実装では，PUT 通信（ステップ 1 と 3）のライブラリの中で完了通知を待ち，SYNC IMAGES 文の実行（ステップ 2 と 4）で受信側に知らせている．

(3) Omni XMP による GET

CAF ベンチマークの GET の測定では，イメージ番号 1 を image1，4 を image2 とし，pingpong を以下の 4 ステップで実行している．

1. image2 で x(1:ndata) = x(1:ndata) [image1]
2. image1 と 2 で相互に SYNC IMAGES
3. image1 で x(1:ndata) = x(1:ndata) [image2]
4. image1 と 2 で相互に SYNC IMAGES

Omni XMP トランスレータにより，ステップ 1 の代入文の右辺は以下のように配列関数（配列を返す関数）の引用に変換される．

```
x(1:ndata) = xmpf_coarray_get_generic (
     xmpf_descptr_x ,        !! coarray 変数 x の記述子
     image1 ,                !! 参照先イメージ番号
     x ( 1 : ndata ) )       !! 参照する変数
```

データ量が大きいときのバンド幅は PUT の 32%前後である．バンド幅が小さい理由は，ローカル側で 2 回のデータコピーが行われるためと考えられる．一つは，GET 通信の受信バッファから配列関数の結果変数へのコピー，もう一つは，結果変数から左辺式へのコピー（配列代入）である．受信バッファを介さず結果変数へ直接受信するためには結果変数を通信ライブラリに登録する必要があるが，結果変数は Fortran システムが管理するヒープ領域に割付けられるため FJ-RDMA の制限事項に該当し登録することができない．

このオーバヘッドコストは，ライブラリインタフェースに配列関数を用いたために生じたものだが，トランスレータ方式では回避するのが難しい．一般に，coindexed object（coarray 変数に角括弧を付けた表記）は一般の式の中に出現できるので，その式の形状に応じた配列関数の形で実行時ライブラリを用意するのが自然である．このベンチマークに現れたような，右辺全体が coindexed object である配列代入文に限るなら，PUT 通信と同じように代入文全体をサブルーチン呼出しに変換することはできる．

(4) FJ CAF による PUT

FJ PUT の測定で用いた CAF ベンチプログラムは，Omni PUT で用いたものと同じである．

レイテンシを Omni PUT と比較すると，データサイズが小さいとき 4.0 倍，大きいとき 1.66 倍である．この差の理由を調査した結果，定性的には coarray を含む配列代入文についてのコンパイラの最適化が不完全であるためと分かった．PUT 通信は以下の配列代入文で記述されている．

x(1:ndata)[image2] = x(1:ndata) (1)

富士通コンパイラはこれを，長さ ndata の一時配列 tmp を介して

tmp(:) = x(1:ndata) (2)

x(1:ndata)[image2] = tmp(:) (3)

と変換している．この冗長な変換は，CAF ベンチマークプログラムがたまたま代入文両辺に同じ ccoarray 変数を引用していたために生じている．もしも代入文(1)の両辺の変数の名前が違っていれば，左辺と右辺でデータ依存がないことが明らかなので，このような変換を生成することはなく，性能低下はなかった．このケースでは両辺の変数の名前が同じだが，添字のアクセスがずれなく完全に一致しているので，(1)を(2),(3)に変換する必要はなかった．

(5) FJ CAF による GET

FJ GET の測定で用いた CAF ベンチプログラムは，Omni

GET で用いたものと同じである．

FJ GET を FJ PUT を比較すると，レイテンシは差が±15%以内でありほぼ等しく，Omni PUT に対して同程度の性能の開きがある．この理由も FJ PUT と同様であった．GET 通信は以下の配列代入文

x(1:ndata) = x(1:ndata) [image2] (4)

で記述されていて，富士通コンパイラはこれを一時変数を介した GET に変換している．

4.2 ノードを跨ぐバインディングの影響

前節と同じ測定を，image1 と image2 が別々のノードにある場合で行った．レイテンシの測定結果を図 2 に示す．ノードを跨がない場合と比べて，グラフの形は大きくは違わないが，以下のような傾向は見られた．

- MPI 版のバンド幅は，データが大きいときノードを跨ぐ方が良い（40%程度高い）．
- Omni のレイテンシは，データが小さいときノードを跨ぐ方が悪い（10～22%大きい）．データが大きくなると数%の差に縮まる．
- 富士通コンパイラの性能は大きな差がない（最大 14%の上下）．

MPI 通信でノード内よりノード間のバンド幅が大きい理由は，ノード内では共有メモリを使い，ノード間では Tofu ライブラリを使って実装されているためである．逆に FJ-RDMA でノード内の方が性能がよい理由は，常に Tofu ライブラリを使用しているためである．ノード内プロセス間で Tofu ライブラリ通信を行うと，Tofu ネットワークに出ることなく，ノードに付いているルータでループバックされるので，レイテンシが小さい．

4.3 Coarray 変数の引数渡しによる得失

coarray 変数をサブルーチン引数として受渡しするときの性能への影響を評価する．図 3 の FJ put と Omni put のグラフ（実線）は，0 節で述べた PUT による ping-pong のレイテンシの図 1(a)を非対数軸に置き換えたものである．PUT 通信は 0 節の配列代入文(1)で記述されている．図 3 の FJ subput と Omni subput（点線）は，これをサブルーチン呼出し

call cafput(x, x, 1, ndata, image2) (5)

に置き換えただけのプログラムの結果である．サブルーチン cafput は以下の通り定義されている．

```
subroutine cafput(target, source, disp, count, image)
  double precision, dimension(:), codimension[*] :: target
  double precision, dimension(:)                 :: source
  target(disp:disp+count-1)[image] = source(disp:disp+count-1)
end subroutine cafput
```

(1) Omni CAF のグラフ

同図(a)で，Omni put と subput のレイテンシの差は，サブルーチン呼出しによる一定のコスト増であると考えられる．

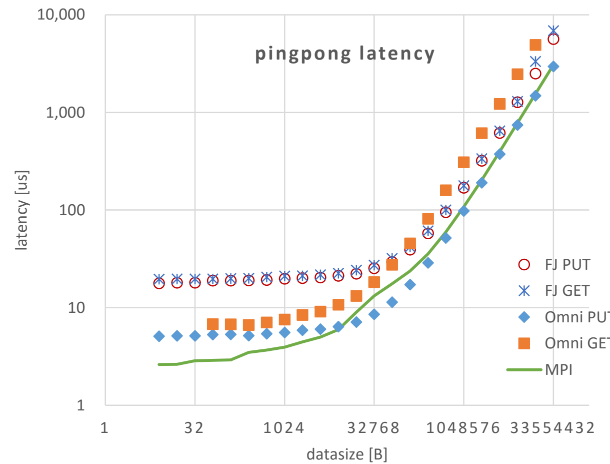
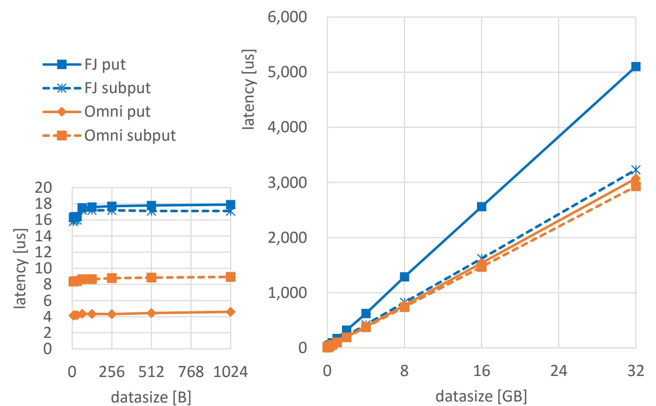


図 2 ノードを跨ぐときの ping-pong 性能



(a) サイズ 1KB まで (b) サイズ 32GB まで

図 3 引数渡しによるレイテンシの変化

サブルーチン入口では仮引数 target に対して記述子を得る処理が実施され，出口では sync memory 操作が実施される．データサイズが十分に大きな場合には，通信時間に対してこれらのコストが無視できることが同図(b)から分かる．

(2) FJ CAF のグラフ

FJ put と subput の比較では，同図(a)でほとんど性能差がないことから，サブルーチン呼出しによるコスト増は富士通コンパイラでは小さいと考えられる．しかし同図(b)を見ると，FJ put だけが他の三者と比べて 66%もレイテンシが大きいことが分かる．この主な原因は，coarray 変数を含む配列代入文についてコンパイラの最適化が未熟なためであると分かった．前述の配列代入文は長さ ndata の一時配列 tmp を介して

tmp = x(1:ndata) (3)

x(1:ndata)[image2] = tmp (4)

ⓒ2017 Information Processing Society of Japan

と計算される．tmp は PUT 通信のローカル側の対象となれる領域であり，Fortran システムが確保する．この冗長な変換は，CAF ベンチマークプログラムがたまたま代入文両辺に同じ ccoarray 変数を引用していたために生じている．もしも代入文(1)の両辺の変数の名前が違っていれば，左辺と右辺でデータ依存がないことが明白なので，このような冗長な変換が起こることはない．

4.4 同期系の比較

CAF の SYNC ALL 文は，sync memory 操作と全イメージ間のバリア同期を同時に行う．ここで sync memory 操作とは，自イメージのメモリのローカルとリモートのアクセスを確定させるローカルな処理であり，具体的には他イメージからの自イメージへの書き込みを完了させて他イメージからも参照可能な状態にするなど，きわめて実装に依存した処理を行う．また，コンパイラの内部論理にとっては，sync memory 操作を跨いでの code motion の最適化を止めるなどの意味をもつ．

SYNC IMAGES 文は，1 つ以上のイメージインデックスを引数にもち，その各イメージと自イメージの間の 1 対 1 の同期を行う．この文も sync memory 操作を同時に行う．

図 4 に SYNC ALL/IMAGES 文の実行時間の測定結果を示す．比較のために同じ条件で計測した MPI_Barrier の性能を添えている．横軸の sync images には以下のバリエーションがある：pairwise は 2 イメージずつのペアを作ってそれぞれ 1 対 1 の同期を行う．ring 2 は両隣りのイメージと 1 対 1 同期を作って全体でリング状になるよう構成する．rand 2 はランダムに選択した 2 イメージと 1 対 1 同期を行うので，ring 2 よりもネットワークが交錯しシグナルの到着順序が乱れやすい．ring n，rand n は，同様にこれらを n イメージとの間で行う．同期の実行時間は，ディレイを含むループについてディレイの後に同期を入れた場合と入れない場合で計測し，その差から求めているため，測定誤差はある程度大きい．

(1) FJ CAF の実装と評価

sync memory 操作とバリア同期相当を以下のように行う．

- ARMCI_AllFence の呼出し：自イメージに送られているすべての書き込みを完了させ，自イメージからのすべての送信と送信元データの参照を完了させる．
- MPI_Iallreduce と MPI_Test の呼出し：“stat=” 指定子（同期の異常を検出しエラー値を返す）の実現のため，全対全同期を reduction 演算で実装している．

前者は，PUT 通信を非同期で実装しているために必要となる．後者については，“stat=” 指定子の指定がない場合には同期の異常を無視したより高速なバリア同期に切り替える方法もあるが，現在の実装では常に reduction 演算で実装している．

図 4 に示した測定結果では，SYNC ALL の実行時間は MPI のバリアと比較してそれぞれ 3.4 倍と 8.6 倍大きい．SYNC IMAGES は SYNC ALL の実行時間よりも常に小さいので，FJ CAF では同期はなるべく局所的に行う方がよいと分かる．

(2) Omni CAF の実装と評価

現在，GET/PUT は常に同期通信で実装されているため，sync memory 操作で待ち合せるものはない．また，“stat=” 指定子は未サポートなので，イメージ間で収集する情報はない．その結果，SYNC ALL は単純に MPI_Barrier の呼出しで実現できる．SYNC IMAGES は相手メモリのカウンタをアトミックにインクリメントする post 操作と，自メモリのカウンタが 0 である間を非 0 になるまで待ってデクリメントする wait 操作だけで実現している．

図 4 の測定結果では，同期相手が 4 以上になると，SYNC IMAGES よりも SYNC ALL を使う方が高速である．

5. アプリケーション性能の評価

Himeno ベンチマークについて，MPI 版プログラム himenoBMTxpr.f90 を元に，等価な Coarray 記述に置き換えて CAF に移植し，元の MPI 版と比較して FJ　CAF と Omni CAF を評価した．

5.1 CAF プログラム・生産性の比較

FJ CAF と Omni CAF はどちらも Coarray 機能の記述と

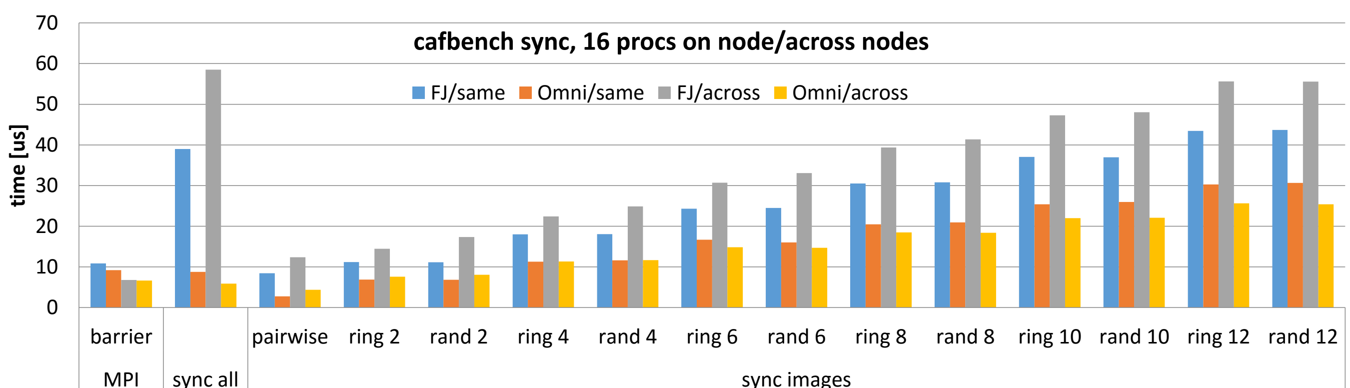


図 4　SYNC ALL/IMAGES 文の実行時間（16 プロセス，同一ノード内／8 ノード分散）

MPI ライブラリ呼出しを同じプログラムの中で混在できるので，移植の中間段階でも動作確認を行いながら進めることができる．最終的に評価した CAF プログラムのソース行数は，コメント行・空行を除いて 457 行であり，元の MPI の 610 行に比べて 25%削減された．MPI 版では実行文中の改行が多いなどスタイルが違うので単純には比較できないが，MPI では MPI_Type_vector 関数を使って派生型の再帰的な定義によって表現される通信パターンが，CAF 版ではコンパクトな添字三つ組 lbound:ubound:stride で表現できたことなど，簡潔な表現に代わることで行数が減る傾向は見られた．

仕様のサポート範囲には Omni と FJ で違いがあったため，マクロ#ifdef による切り分けが必要だった．例えばイメージ間の集計演算を行う Fortran2015 の組込みサブルーチン co_sum では，FJ は 1 引数のもの（入力変数に戻り値を上書きする）だけがサポートされ，Omni では 2 引数のもの（入力変数は参照のみで出力変数に値を戻す）だけがサポートされている．

エラーメッセージの表現や行番号の適切さは FJ 版の方が信頼できた．しかしバッチ処理でなく卓上の PC で動作確認とデバッグが行える点では，マルチターゲットである Omni の方が便利であった．今回は使用しなかったが，並列実行でしか再現しない同期の漏れやデッドロックへの対応では，FJ 版の “stat=”指定子が活用できるかもしれない．

5.2 実行結果

図 5 に，FX100 上で測定した MPI 版の結果と，FJ CAF と Omni CAF による CAF 版の結果を示す．縦軸は逐次実行に対する加速率を示し，逐次版 Himeno を FX100 の 1 スレッドで最も高速になるようにオプション –Kfast,reduction を付けて翻訳し実行した結果を 1 としている．スレッド並列は使っていない（Flat-MPI）．32 プロセスまでの Omni と FJ の性能は殆ど変わらず，MPI とほぼ同等と言える（32 プロセスでそれぞれ MPI の 96%と 95%）．ノードを跨ぐ 64 プロセス以上になると三者で少し差が開く傾向がある（256 プロセスでそれぞれ MPI の 83%と 68%）．問題サイズが小さくなると 3 者の差がより開く傾向がある．

同図(b)のように，ノードに 2 プロセスずつを割り当て，各プロセスを 16 スレッドの自動並列とした．Flat-MPI よりもこのようなハイブリッド列の方が，Omni と FJ の性能差は縮まる傾向がある．図 2 に見られるようにサイズ XL ではわずかに逆転している（256 スレッドでそれぞれ MPI の 94%と 97%）．サイズ M と L では FJ より Omni の方が性能が高かった．

6. 議論

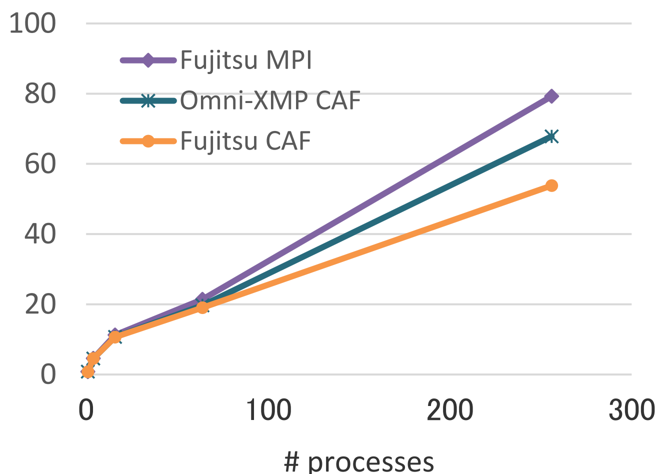
6.1 EPCC CAF ベンチマーク

Fortran2008 の文法では，SYNC IMAGES 文に指定されるイメージインデックスは，同じものが複数回現れてはならないが，cafsync と cafhalo では実行イメージ数が少ないとき同じ番号が 2 度現れる場合がある．FJ CAF ではこれを実行時エラーとして検出して実行を止めている．Omni CAF ではエラー検出は行わず，イメージインデックスの出現回数だけ同期を実行しているので，プログラムの意図通り動作している．Makefile とソースプログラムを見る限りでは，Cray と古い g95 でもこのプログラムは動作しているようだ．

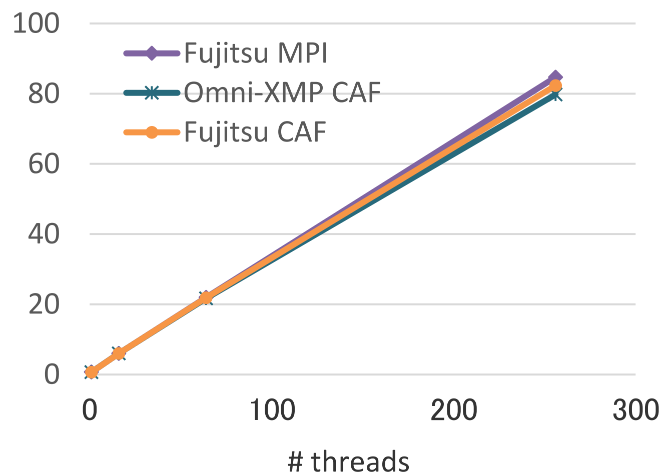
6.2 CAF 言語仕様と性能

4.1 節(4)で述べたように，正確な実装と最適化は相反する場合がある．たとえ非常に稀であっても誤動作する可能性が少しでもあれば，コンパイラは最適化を止める方を選択しなければならないので，解析の精度を上げることは重要である．

従来のデータフロー解析では，添字を無視して名前の一致があれば重なりがある可能性があると判断するのが一般



(a) L サイズ／Flat-MPI（ノード内 32 プロセス）



(b) XL サイズ／ハイブリッド（プロセス当り 16 スレッド）

図 5 FJ/Omni CAF と MPI による Himeno ベンチマークの結果（逐次版 Himeno の結果を 1 とする）

ⓒ2017 Information Processing Society of Japan

的だった．また，ループ中のデータを対象とするデータ依存解析では，配列各次元の添字の一致やずれの数を判断に加えるものであった．さらに今後は，イメージ番号の違いまで考慮に入れた解析技術が必要になるかもしれない．

7. 関連研究

CAF の実装は，フリーのものでは， Houston 大学の OpenUH コンパイラを開発基盤とした UH-CAF[7]と，Rice 大学の ROSE コンパイラを開発基盤とした caft[8]が有名である．ただし caft のサポートする書式は Fortran の標準のものとは少し異なっている． 近年リリースされた OpenCoarray は, V5.1 以降の GCC にリンクできるライブラリの形で公開されている. これらの実装では CAF アプリケーションの性能はベースの Fortran コンパイラの性能に大きく依存する[2]．Omni CAF はトランスレータ方式の採用によってベースコンパイラを自由に選択できるようにした．

ベンダでは Cray と Intel が古くから提供している．富士通も含めこれらのベンダのアプローチは, 自社の Fortran コンパイラの新機能として直接手を入れる方法である．

8. まとめ

EPCC の CAF ベンチマークと Himeno ベンチマークを使い，2 つの CAF コンパイラを比較評価した．結果は両者の特徴を際立たせるものとなった．

Omni XMP の CAF 機能は性能最優先で開発されていて，今回初めて評価した FX100 でも高い性能を示している．MPI の両側通信との比較では，PUT 通信と SYNC IMAGES 同期を合わせたレイテンシで，2KB 以下の eager 通信に対して 26～60%遅いが, 4KB 以上の rendezvous 通信に対して 28～40%速い．Himeno ベンチでは flat-MPI の 32 プロセスで MPI の 96%，256 プロセスで 83%，16 スレッド自動並列×16 プロセスで MPI の 94%の性能を得た．GET 通信で性能が出ないケースでは，手続間インタフェースでのトランスレータ方式の不利な点が明らかになった．

富士通 Fortran は網羅的な誠実な実装が優先されている．coarray 変数に絡む配列代入文の実装と, SYNC ALL 文の実装に関しては，レアケースを正しく実現するために全体の性能を落とさないようにすることの難しさを再認識した．さらに最適化を進めることで解決できる問題なので，今後のバージョンアップに期待できる. Himeno ベンチの一部など, Omni の実装よりも性能が出るケースもあった．今後さらに分析を加え，双方のコンパイラの改善にフィードバックしたい．

参考文献

[1] XcalableMP. http://www.xcalablemp.org/ja/
[2] Hidetoshi Iwashita, Masahiro Nakao, Mitsuhisa Sato. *Preliminary Implementation of Coarray Fortran Translator Based on Omni XcalableMP.* Proc. of 9th International Conference on Partitioned Global Address Space Programming Models (PGAS2015), P.70-75, Washington, D.C. USA, September, 2015.
[3] 岩下英俊，中尾昌広，佐藤三久．NAS Parallel ベンチマークの CAF への移植と Omni XcalableMP CAF コンパイラを用いた評価．第 153 回 HPC 研究会，2016 年 3 月．
[4] 住本真司，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．SACSIS 2012 – 先進的計算機基盤システムシンポジウム．情報処理学会．2012 年 5 月．
[5] EPCC Fortran Coarray micro-benchmark suite https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-co-array-fortran-micro
[6] David Henty. A Parallel Benchmark Suite for Fortran Coarrays. In *Applications, Tools and Techniques on the Road to Exascale Computing*, Advances in Parallel Computing, Vol.22, pp.281-288. 2012
[7] Deepak Eachempati, Hyoung Joon Jun and Barbara Chapman. *An Open-Source Compiler and Runtime Implementation for Coarray Fortran.* PGAS'10 Proc. of 4th Conference on PGAS Programming Models, No.13, 2010
[8] Cristian Coarfa. Portable High Performance and Scalability of Global Address Space Languages. Ph.D. Thesis, Rice University, January 2007.

ⓒ2017 Information Processing Society of Japan