Vol.2016-HPC-153 No.32
2016/3/3

FX100における MPI RMA ロック同期の実装と評価

畑中 正行[1]　堀 敦史[1]　石川 裕[1]

概要：京の継承機種である FX100 では，Tofu インターコネクト上のアトミック操作が追加されている．また MPI-3 仕様では，RMA の公開・非公開ウィンドウに対し一体型のメモリモデルが追加され，ハードウェアを活かした RMA 操作が可能になっている．本稿では FX100 における，ロック同期に対する最適化実装，並びに予備的な性能評価を報告する．

1. はじめに

MPI RMA (Remote Memory Access) ロック同期はこれまで，共有メモリシステムや専用のグローバルな共有メモリを有する一部のシステムでの RMA 操作の高速化のために用意されていた．そのため，RMA ロック同期は，制約も多い上，一般のシステムでは極端な性能劣化を招く場合も多く，プログラミングモデルとして積極的に推奨されて来なかった．しかしながら，RDMA アトミック操作をサポートするインターコネクトが普及し，特定のシステムでなくても，インターコネクト上で高速なロック機構を実装できるようになってきた．このため，MPI-3 仕様では RMA 操作について，近年のインターコネクト・ハードウェアに適した，大幅な改訂がなされている．

本稿では，RDMA アトミック操作をサポートした富士通 PRIMEHPC FX100 上での MPI RMA ロック同期の実装について検討する．

2. 関連研究

Gerstenberger ら [1] は，Cray Gemini/Aries 上のアトミック操作を使って，Backoff ロックベースの Readers-Writer ロックで `MPI_Win_lock` を実装した．彼らの目的は，MPI-3 仕様とその背景技術が適切かどうかを検証する目的と考えられ，実装も汎用的かつ教科書的で参考にするところが多い．一方で，細かな実装に着目すると，ロック待ちが発生すると，リモートメモリのポーリングにより，スループットが劣化する恐れがある等，課題もある．

Li ら [2] は，Infiniband 上のアトミック操作を使って，分散キューベースの Readers-Writer ロックで `MPI_Win_lock` を実装したが，`MPI_Win_lock_all` については，ウィンドウに属するすべてのプロセスに対し `MPI_Win_lock(MPI_LOCK_SHARED)` を発行する実装であり，まったくスケールしない．しかしながら，アサーションに `MPI_MODE_NOCHECK` フラグが指定された場合，内部のロック処理を NOP にすることで，一部のケースで，この問題を回避している．

Kumar ら [3] は，BlueGene/Q の Messaging Unit 上のアトミックを操作を使ったロック実装の性能評価を行った．しかしながら，ベンチマーク及びアプリケーションの性能評価が中心であり，実装は概要のみで詳細は不明である．

上記のいずれの研究も，主に 2 ノード間のロックのレイテンシの評価はあるが，複数プロセスが同時にロック獲得要求した場合，どのような性能特性を示すかの観点がない．

3. 背景

3.1 MPI-3 仕様

MPI-3 仕様では，非公開ウィンドウ及び公開ウィンドウに分かれた従来型の `MPI_WIN_SEPARATE` メモリモデルに加え，非公開/公開ウィンドウが一体となった `MPI_WIN_UNIFIED` メモリモデルが追加された．CPU コアからの load/store 命令と，RDMA エンジン経由の RDMA Read/Write に対し，キャッシュ一貫性を提供するシステムでは，重複しないメモリ領域に対し同時アクセスが可能であり，非公開ウィンドウと公開ウィンドウを分離する必要がなく，`MPI_WIN_UNIFIED` メモリモデルが有効である．この場合，非公開と公開ウィンドウの間のデータ同期のためのオーバヘッドを削減できる．また，同じメモリ領域への同時アクセスについては，そうした操作自体の禁止ではなく，「結果は未定義」と仕様上の規定が緩和された．但し，例外として同じメモリ領域への同時アクセスについては，一部の `accumulate` 操作については，結果は保証されることに注意が必要である．

[1] 理化学研究所
RIKEN

MPI RMA (Remote Memory Access) 操作のうち，MPI_Get，MPI_Put，MPI_Accumulate，及びその亜種のような RMA 通信呼出しは，基本的に，RDMA Read, Write, 及び Atomic 操作に対応づけることができる．

RMA 同期呼出しは 3 種類提供されている．

- MPI_WIN_POST,START,COMPLETE,WAIT は，active target 通信であり，アクセスされるウィンドウを所有するプロセス (target process) が明示的に，ウィンドウの開示/非開示 (expose/unexpose) のための RMA 同期呼出しが必要であり，RMA 通信呼出し側プロセスと共に明示的に，データ通信の準備/後始末に関与する必要がある．
- 同様に，MPI_WIN_FENCE は active target 通信かつ集団的であり，RMA 通信呼出し側か否かに関係なく，集団的に RMA 同期呼出し MPI_WIN_FENCE を呼出し，データ通信に関与する必要がある．
- これに対し，MPI_WIN_LOCK,UNLOCK は，passive target 通信，つまり RMA 通信呼出し側プロセスしか，データ通信に関与しない．よって，アクセスされるウィンドウを所有するプロセスは何も呼び出す必要はないし，ウィンドウの開示/非開示の管理は不要である．

これまで，ノード間のバリア同期を高速化するハードウェア支援機構を有する場合，MPI_WIN_FENCE の使用が一般には好まれてきた．また，共有メモリを有するような一部のシステムでは，MPI_WIN_LOCK,UNLOCK が使われてきたが，近年，インターコネクトでアトミック操作が普及したことにより，ロックに対する実装のハードルが下がってきた．このため，passive target 通信，つまり RMA ロック同期呼出しの利点である，RMA 通信呼出し側プロセスしかデータ通信に関与しないという，真な意味での片側通信に期待が寄せられている．

3.2 MPI RMA ロック同期呼出し詳細

図 1 は，RMA ロック同期呼出しの MPI_Win_lock の形式を示す．*lock_type* 引数は，MPI_LOCK_EXCLUSIVE または MPI_LOCK_SHARED のいずれかの値を指定できる．つまり，MPI の RMA 同期呼出しでは，Readers-Writer ロックと呼ばれるタイプのアルゴリズムを実装する必要がある．*rank* 引数は，アクセスするウィンドウを所有する MPI プロセスを指す．よって，ステンシル計算のように複数のプロセスの袖領域にアクセスするためには，複数のロック呼出しが必要である．

これに対し，図 2 の，MPI-3 仕様で導入された MPI_Win_lock_all では，そのウィンドウ *win* に関連するすべての MPI プロセスの所有するウィンドウに，MPI_LOCK_SHARED でアクセスできる．

```
int MPI_Win_lock(int lock_type, int rank, int assert,
        MPI_Win win);
int MPI_Win_unlock(int rank, MPI_Win win);
```

図 1 MPI_Win_lock/unlock の形式

```
int MPI_Win_lock_all(int assert, MPI_Win win);
int MPI_Win_unlock_all(MPI_Win win);
```

図 2 MPI_Win_lock_all の形式

また，アプリケーション・プログラマは，相互排他の必要なロックを保持していたり，そうしたロックを獲得しようとするプロセスがいないことを保証できる場合，MPI_MODE_NOCHECK を RMA ロック同期呼出しの *assert* 引数に指定することにより，RMA ロック同期の内部処理を NOP にするよう MPI 実装に指示することができる但し，*assert* 引数を使って最適化するかどうかは，実装依存である．図 3 に，MPI_MODE_NOCHECK の使用例を示す．

```
int assert = MPI_MODE_NOCHECK;
MPI_Win_lock_all(assert, win);
/* RMA 通信呼出し */
MPI_Win_unlock_all(win);
```

図 3 MPI_MODE_NOCHECK の使用例

例えば，各プロセスのアクセスするメモリ領域がオーバラップせず，MPI_LOCK_EXCLUSIVE つきの MPI_Win_lock 呼出しはなく，かつ put や get のような RMA 通信呼出しを使うために，形式上 MPI_Win_lock_all を使う場合 (RMA アクセス期の開始) には，一つの最適化オプションになる．

3.3 課題

図 4 及び図 5 は，RMA 操作を使ったステンシルコードの例であり，それぞれ fence 同期呼出し及び lock_all 同期呼出しの例である（図 5 の例は，MPI-3.1 仕様の例 11.7 をステンシル用に修正したものである）．

```
while (--iterations) {
  compute();
  MPI_Win_fence(..., win);
  for (i=0;i<nbrs;i++)
    MPI_Put( ..., win);
  MPI_Win_fence(..., win);
}
```

図 4 MPI_Win_fence の例

```
MPI_Win_lock_all( , win);
while (--iterations) {
  compute();
  MPI_Barrier(com);
  for (i=0;i<nbrs;i++)
    MPI_Get( ..., win);
  MPI_Win_flush_all(win);
}
MPI_Win_unlock_all(win);
```

図 5 MPI_Win_lock_all の例

これらの例では，ステンシル計算の計算部 compute が完了したことを保証するために，fence または barrier のような集団操作を使用する．それから通信部では，put または get を使用して袖領域の交換を行い，通信が完了したこ

とを確認し，また計算部に戻るという処理を繰り返す．

RMA 通信呼出し put や get を発行するためには，なんらかの RMA 同期呼出しを使って RMA アクセス期に入る必要がある．図 5 の例では，ステンシル計算の繰り返し全体を，RMA 同期呼出し lock_all と unlock_all で囲むことによって RMA 同期呼出しのオーバヘッドを削減している．また flush_all は，呼出したプロセスが発行した全ターゲットに対する未完了の RMA 通信操作を完了する．fence が集団的にプロセス間の実行同期とデータ同期を両方実行するのに対し，flush_all は呼び出したプロセスに閉じたデータ同期のみを実行する．アプリケーション・プログラマが 2 種類の同期を使い分けることで，より低オーバヘッドの通信を実現することが可能となる．

課題は，アプリケーション・プログラマが MPI 仕様書に従って，例えば図 5 のような，ステンシル計算コード書いたとして，数万ノード以上の大規模実行で，問題なくそのコードが動作するだろうかと問題である．

本稿では，PRIMEHPC FX100 上での MPI RMA ロック同期実装について，大規模実行の面から検討する．

4. 設計と実装

4.1 ToFu インターコネクト 2

富士通 PRIMEHPC FX100 は，京コンピュータ，PRIMEHPC FX10 に継ぐ，3 世代目の計算機である．FX100 でサポートされるインターコネクトは，Tofu インターコネクト 2 (以降 Tofu2 と呼ぶ)[4] であり，SPARC64 XIfx チップに統合されている．Tofu2 では，既存の RDMA Read/Write 操作に加え，RDMA エンジンに Atomic Read Modify Write (ARMW) 操作が追加されている．この ARMW 操作は，CPU コアからのアトミック命令，及び 4 基ある RDMA エンジンからの同操作による同時アクセスに対し，アトミック性を有する．ARMW 操作には，compare_and_swap，fetch_and_store，fetch_and_add，fetch_and_or，fetch_and_and 等の基本的なアトミック操作がある．

4.2 ロックアルゴリズムの選択

ロックアルゴリズムについては多くの研究があり，インターコネクト上での RDMA 操作を使ったロックアルゴリズムの評価についても増えてきた (節 2 参照)．しかしながら，大規模での MPI_Win_lock_all の同時発行については，あまり研究されていない．

本稿は，大規模での MPI_Win_lock_all の同時発行における全体の振舞いの特性を踏まえた上で，課題を探ることが目的である．よって，唯一のグローバルロック変数に対し，一斉に RDMA 操作を使ったリモートポーリングが発生しうるようなナイーブなロック・アルゴリズムを採用することは避け，まずは，保守的なロックアルゴリズムを選択することとした．ここでは，分散キューロックアルゴリズムである MCS Readers/Writer ロックアルゴリズム [5] を採用することにした．このアルゴリズムはさらに，reader と writer の競合時のスケジューリングにおいて，中立，reader 優先，writer 優先の 3 種類のアルゴリズムが提案されているが，HPC のユースケースにおいて，完全にシリアライズされる writer が大量に存在することは考えにくく，reader 優先で検討を進める（以降，MCS-RW-RP と呼ぶ）．

図 6 及び図 7 は，MCS ロックアルゴリズムベースの Readers/Writer ロック（Reader 優先）の疑似コードの [5] からの抜粋である．紙面の都合上，アルゴリズムの詳細については説明しない（詳細については，[5] を参照）．

図 6 の ReaderLock は，MPI_Win_lock_all() の実装の構成部品になりうる．このコードでは，writer がロック中 (WAFLAG) でなければ，1 回の RDMA アトミック操作 (fetch_and_add) で済む (*02* 行目)．writer がロック中 (WAFLAG) であれば，MCS の分散キューロックベースの待ちキューに登録する (*03*〜*11* 行目)．writer がアンロックし，blocked が false になるまでビジーループする (*09* 行目)．もし他にも待っているプロセスがいるなら，blocked を false に変更する (*10*〜*11* 行目)．しかし，*02* 行目の WAFLAG のチェックから *04* 行目の fetch_and_store までの間に，writer がアンロックする可能性があり，*05*〜*08* までで，WAFLAG を二重チェックし，競合状態を解消する．

```
01 procedure ReaderLock(L : *lock, I : *qnode)
02   if fetch_and_add(&L->rcnt, RC_INCR) & WAFLAG then
03     I->blocked := true
04     I->next := fetch_and_store(&L->r_hd, I)
05     if (*(&L->rcnt) & WAFLAG) == 0 then
06       head : *qnode := fetch_and_store(&L->r_hd, nil)
07       if head != nil then
08         head->blocked := false
09     repeat while I->blocked
10     if I->next != nil then
11       I->next->blocked := false
12 end
```

図 6 MCS-RW-RP ReaderLock の疑似コード

また，この疑似コードは，マルチプロセッサを意図したコードであり，インターコネクト上のロック実装では，ロック変数 L の実体は (通常) リモートにあるため，*05* 行目の rcnt の読出しと，*08* 及び *11* 行目の blocked への書込みは，それぞれ RDMA Read と Write 操作に置換えなければならないことに注意が必要である．

同様に，図 7 の ReaderUnlock は，カウンタ rcnt を $1 \rightarrow 0$ にデクリメントした最後の reader でもなく，writer のロック待ち (WIFLAG) もいないなら，1 回の RDMA アトミック操作 (fetch_and_add) で済む (*02*〜*03* 行目)．そうでないな

```
01 procedure ReaderUnlock(L : *lock, I : *qnode)
02   if fetch_and_add(&L->rcnt, -RC_INCR) ==
03     (RC_INCR + WIFLAG) then
04     if compare_and_swap(&L->rcnt,WIFLAG,WAFLAG) then
05       L->w_hd->blocked := false
06 end
```

図 7 MCS-RW-RP ReaderUnock の疑似コード

ら，rcnt のフラグを WAFLAG を立て，writer のロック中に状態を遷移する (*4* 行目)．それから，writer の待ちキューの先頭の writer である L->w_hd のビジーループを解くために blocked を false に変更する (*05* 行目)．しかし，*02* 行目の WIFLAG のチェックから *04* 行目の compare_and_swap までの間に，新たに reader がロック (rcnt をインクリメント) する可能性があり，compare_and_swap が失敗したら，何もしない．また，*05* 行目の w_hd の読出しと，その blocked への書込みは，それぞれ RDMA Read と Write 操作に置換えなければならないことに注意が必要である．

4.3 MPI_Win_lock_all 評価実装

Gerstenberger らの RMA 実装 foMPI [6] は，教科書的であるため，この RMA ロック実装について，簡単に触れる．foMPI 実装では，グローバルとローカルの 2 種類のロック変数を管理する．グローバル・ロック変数は，ウィンドウ・グループ全体で 1 個，ローカル・ロック変数は，ウィンドウ・グループのプロセスごとに 1 個ある．また，MPI_Win_lock(MPI_LOCK_EXCLUSIVE) は，MPI_Win_lock_all(MPI_LOCK_SHARED) との相互排他ためにグローバルロックを獲得し，それから同じ対象ランクに対する他の MPI_Win_lock(MPI_LOCK_EXCLUSIVE) との相互排他のためにローカルロックを獲得するため，2 つのロックを獲得する必要がある．これに対し，MPI_Win_lock_all は，MPI_Win_lock(MPI_LOCK_EXCLUSIVE) との相互排他のためにグローバルロックを獲得するだけでよいため，1 つのロックの獲得で済む．

しかしながら，foMPI の実装では，一旦ロックの待ちが発生すると，状態の変化の監視のためリモートメモリ・ポーリングを行う．しかし，これは一般に，ロック獲得までの時間が，同時のロック獲得要求の数に対し指数関数的に増大することが知られている [5]．大規模実行の評価面で，ハード特性が不明な段階で，最初の実装としてリモート・ポーリングを採用するのは非常にハードルが高い．

このため，本実装では，Li ら [2] の実装とは逆に，MPI_Win_lock_all の評価のために，ローカルロックをなくし，グローバルロックのみで MPI RMA ロック同期を実装する．この方式では，MPI_Win_lock(MPI_LOCK_SHARED) も MPI_Win_lock_all も MCS-RW-RP の ReaderLock を呼出し，対となる MPI_Win_unlock(MPI_LOCK_SHARED) も MPI_Win_unlock_all も ReaderUnlock を呼出す．また，MPI_Win_lock(MPI_LOCK_EXCLUSIVE) は MCS-RW-RP の WriterLock を呼出し，対となる MPI_Win_unlock(MPI_LOCK_EXCLUSIVE) は WriterUnlock を呼出す．Gerstenberger らの教科書的な 2 段ロックは不要になる反面，グローバルロックの reader を待たせている場合でさえ，writer のキューはグローバルに 1 つしかないため，対象ランクが異なっても同時に一つしか処理されない問題もある．

5. 評価

5.1 測定環境

理研情報基盤センターに設置されている HOKUSAI-GW の PRIMEHPC FX100 システムを使用した．

5.2 測定結果

図 8 は，FX100 上 Tofu2 の RDMA アトミックを使った MCS-RW-RP 実装おける，reader (共有) ロック呼出しオーバヘッド（ReaderLock-ReaderUnlock）を測定したものである．128 ノード (ppn=32) の環境で，同時に ReaderLock-ReaderUnlock を発行するプロセス数を 4~4096 と変えて測定した．ロックとアンロックを 1000 回実行し，総経過時間が最も長かったプロセスの 1 回あたりの平均値をプロットした．

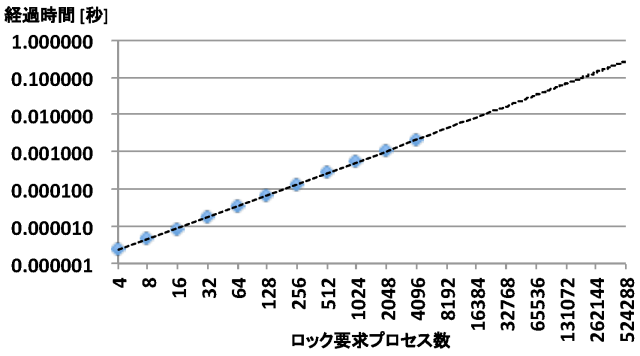

図 8 FX100 MCS-RW-RP 性能

この測定では，4 プロセスの場合に最短の 2.3[μ s] を達成しているが，同時プロセス数に比例して，ロック・アンロック時間が延びている．外挿すると，京の実行規模 663,552 プロセス（京の計算ノード数 82,944×ppn 8）で，約 1 秒かかることがわかる．もちろん，これは，RDMA アトミック操作のない Tofu1 を搭載した京コンピュータや FX10 に比べ，約 1000 倍の改善効果が得られることも同時に示している（図 9 参照．但し，MCS-RW-RP は Tofu ライブラリ直上で実装されており，MPI 実装のオーバヘッドは含まれていないことに注意．そうだとしても，プロセス数 256 で比較すると，FX10 と FX100 で，それぞれ 114[s] と 130[μ

s] の桁違いの効果がある).

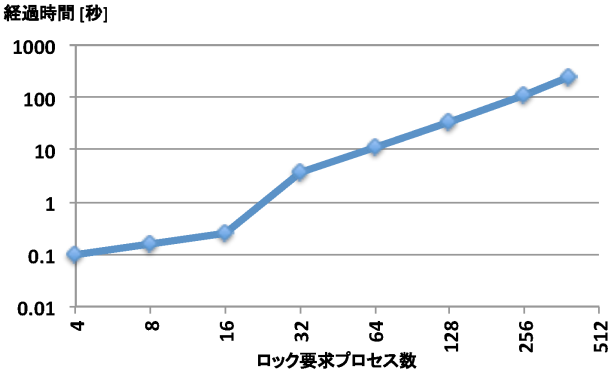

図 9 FX10 上の MPI_Win_lock/unlock
図 8 の測定での ReaderLock/ReaderUnlock の評価コードを MPI_Win_lock/unlock(MPI_LOCK_SHARED) 呼出しに置換えて, FX10 上 24 ノード (ppn=16) の環境で測定したものである

図 10 の測定は, 図 8 の測定と同様にロックとアンロックを 1000 回実行しているが, 1000 回の総経過時間が最も長かったプロセスだけでなく, 全プロセスの総合経過時間を表示している (つまり, 1 回あたりの平均でなく, 1000 回分の経過時間であることに注意). また, 比較のためにノード数を 128 に固定し, ppn (processes per node) を 1,8,32 と変えて測定した.

図 10 では, 最大値に着目するとおよそ, ppn32 で 2.2[s], ppn8 で 0.52[s], ppn1 で 0.069[s] であり, ppn1 を基準にすると, ppn8 で 0.552(=0.069×8), ppn32 で 2.208(=0.069×32) となり, この測定でもまた, 1 つのロック変数に RDMA 操作でアクセスした場合, 同時プロセス数に比例することがわかる. 一方で, 1000 回のロック・アンロックの総経過時間は, プロセスごとに様々であり, ppn32 の場合で 0.1~2.2[s] までの幅がある. 図 10 の右下のグラフは, ppn32 測定時の対象ロックまでの Tofu ネットワークのホップ数を示している. ホップ数の短いところから, 1000 回のロック・アンロックが終了してゆくことが推測できる. 従って, 節 4.2 の reader の処理より, ロックとアンロックで発行される 2 回の fetch_and_add の処理待ちが支配的であることが強く示唆される.

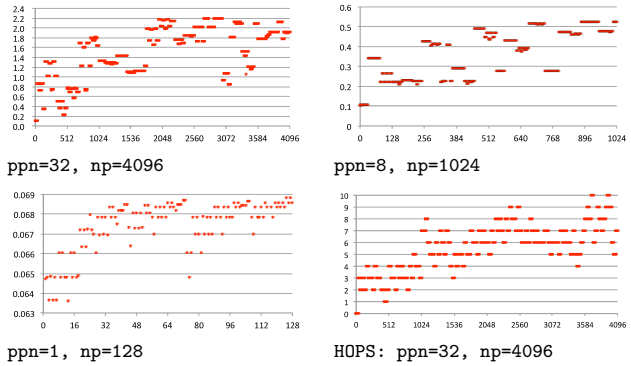

図 10 FX100 MCS-RW-RP (ランクごと)
図 8 の測定と同じだが, ランクごとの測定値. X 軸がランク番号, Y 軸が (各ランク) の lock/unlock(shared) の 1000 回の経過時間 [秒]

そこで, RDMA 操作のスループットを向上するため, Tofu インターコネクトではノードあたり 4 基の RDMA エンジンを有することを活用し, 同時に複数の RDMA アトミック操作を並行処理させる処理を追加した. これまでの MCS-RW-RP 実装では, 対象ロックに対し, RDMA エンジンを 1 基使用していたが, 処理の集中する対象ロック側の受信 RDMA エンジン 4 基使用するように実装を変更した. 但し, 実験の容易さから, ノード内ループバックで RDMA エンジンを介して評価した (つまり, 測定値自体は伝送路遅延を含まないことに注意).

図 11 は, 修正前の RDMA エンジン 1 基使用時の, fetch_and_add を単発で発行した場合の経過時間を示す.

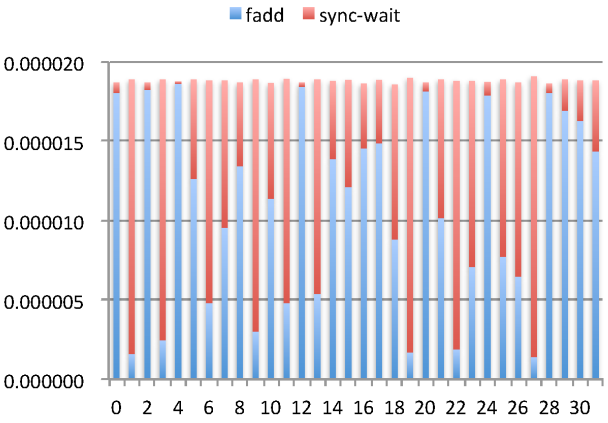

図 11 FX100 MCS-RW-RP 性能 (1 基)
X 軸がランク番号, Y 軸が (各ランク) の fetch_and_add (fadd) の処理時間 [秒]. sync-wait は, CPU による処理完了同期待ち合わせ時間.

これに対し, 図 12 は, 修正後の RDMA エンジン 4 基使用時の測定結果である. 1 基に比べ, 約 40%の改善が見られるが, 1/4 にはならなかった. これは, RDMA アトミック操作に対するバスロックによるシリアライズや RDMA エンジン間のキャッシュの invalidate のため, 台数的な効果は得られなかったものと推測する.

よって, 大規模実行時の MPI_Win_lock_all の使用の是非については, 現時点でまだ一般に利用を推奨できるレベルではなく, もう一段の改善が必要であると考える.

6. 結論

本稿では, RDMA アトミック操作をサポートした PRIMEHPC FX100 上での RMA ロック同期の最適化実装, 並びに予備的な性能評価を論じた. 本実装では, 分散

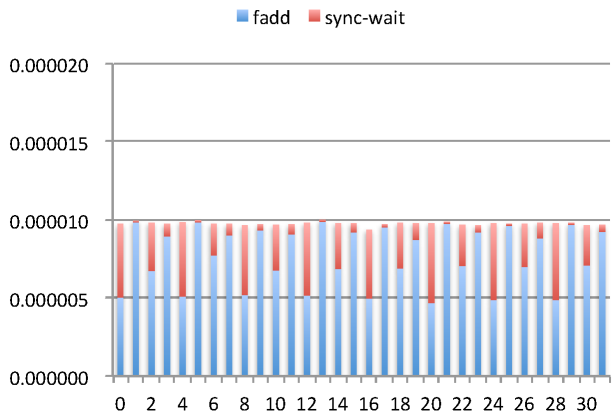

図 12 FX100 MCS-RW-RP 性能（4 基）

キュー型のロックアルゴリズムに分類される，MCS ベースの Readers/Writer ロック（Reader 優先）を，FX100 の Tofu2 インターコネクトの直上に実装した．性能評価を通し，京コンピュータ規模でのロック同期オーバヘッドについて，予想性能を提示した．また，性能測定で得られたボトルネック推定に基づき，処理の集中する対象ロック側の受信 RDMA エンジンを複数使用するよう実装を修正すると，約 40%の改善が見込まれるデータが得られた．

今後，同時ロック要求のプロセス数に比例して RMA ロック同期のオーバヘッドが増大する問題に対し，階層型のロックアルゴリズム等により，対象ロックの存在するノードへの RDMA 操作の集中を軽減する方式を検討してゆく予定である．

参考文献

[1] Gerstenberger, R., Besta, M. and Hoefler, T.: Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided, *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, ACM, pp. 53:1–53:12 (online), DOI: 10.1145/2503210.2503286 (2013).

[2] Li, M., Potluri, S., Hamidouche, K., Jose, J. and Panda, D. K.: Efficient and Truly Passive MPI-3 RMA Using InfiniBand Atomics, *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, New York, NY, USA, ACM, pp. 91–96 (online), DOI: 10.1145/2488551.2488573 (2013).

[3] Kumar, S. and Blocksome, M.: Scalable MPI-3.0 RMA on the Blue Gene/Q Supercomputer, *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, New York, NY, USA, ACM, pp. 7:7–7:12 (online), DOI: 10.1145/2642769.2642778 (2014).

[4] Ajima, Y., Inoue, T., Hiramoto, S., Uno, S., Sumimoto, S., Miura, K., Shida, N., Kawashima, T., Okamoto, T., Moriyama, O., Ikeda, Y., Tabata, T., Yoshikawa, T., Seki, K. and Shimizu, T.: Tofu Interconnect 2: System-on-Chip Integration of High-Performance Interconnect, *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*, ISC 2014, New York, NY, USA, Springer-Verlag New York, Inc., pp. 498–507 (online), DOI: 10.1007/978-3-319-07518-1_35 (2014).

[5] Mellor-Crummey, J. M. and Scott, M. L.: Scalable Reader-writer Synchronization for Shared-memory Multiprocessors, *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, New York, NY, USA, ACM, pp. 106–113 (online), DOI: 10.1145/109625.109637 (1991).

[6] Gerstenberger, R., Besta, M. and Hoefler, T.: foMPI: A Fast One-Sided MPI-3.0 Implementation, `http://spcl.inf.ethz.ch/Research/Parallel_Programming/foMPI/` (2013).