Vol.2017-HPC-158 No.27
2017/3/10

大規模システムにおける耐故障マルチSMPD プログラミング開発実行環境の応用と評価

辻 美和子[1,a)]　佐藤 三久[1]

概要：多数のマルチコアプロセッサやアクセラレータから構成される大規模で階層的なアーキテクチャにおいて，そのような階層性を考慮したプログラミングモデルであるマルチ SMPD プログラミングモデルを提案した．このモデルでは，複数の分散並列プログラムや分散並列とスレッド並列のハイブリッドなプログラムを，ワークフローのタスクとして実行する．また，ノード数の増加にともなうシステム全体の平均故障間隔の短縮に対応するために，マルチ SMPD プログラミング開発実行環境を耐障害性をサポートするように拡張した．本稿では，耐障害マルチ SMPD プログラミング開発実行環境の京コンピュータ向け拡張および評価を行った．とくに，タスクへのデータ入出力時に故障検知のためのハートビートが適切に送られないことがあり，対策を検討した．

1. はじめに

近年では，計算機システムの大規模化や構成要素数の増加により，システム全体の平均故障間隔 (Mean Time Between Failure, MTBF) が短くなっている．このため，障害が発生した場合でもその少し前の時点から実行を再開できるように一定時間ごとにデータを確保しておくなど，耐障害性を意識したアプリケーション開発が必要とされる．しかしながら，既存の多くのアプリケーションを，耐障害性を考慮しつつ書き換えることは大きな労力を要する．著者らは，アプリケーションソースコードを変更することなく耐障害性を実現するために，マルチ SPMD プログラミング開発実行環境を拡張することを提案した [3].

本稿では，拡張したマルチ SPMD プログラミング開発実行環境の京コンピュータへの応用およびその評価を行った．

2. 背景

2.1 マルチ SPMD プログラム開発実行環境

本節では，大規模かつ階層的なシステムを効率的に利用するために提案されたマルチ SPMD プログラミングモデル，およびその開発実行環境について述べる．

将来の大規模システムは，プロセッサ内で NUMA 構造を持つようなメニーコアプロセッサやアクセラレータから構成される巨大かつ階層的なアーキテクチャになると考えられる．OpenMP+MPI などの既存のプログラミングモデルのみでは，このようなシステムを常に効率的に利用することは難しい．我々は，大規模かつ階層的なシステムを効率的に利用するために，マルチ SPMD (Single Program Multiple Data) プログラミングモデルを提案し，これを実現する開発実行環境を実装してきた [8][4].

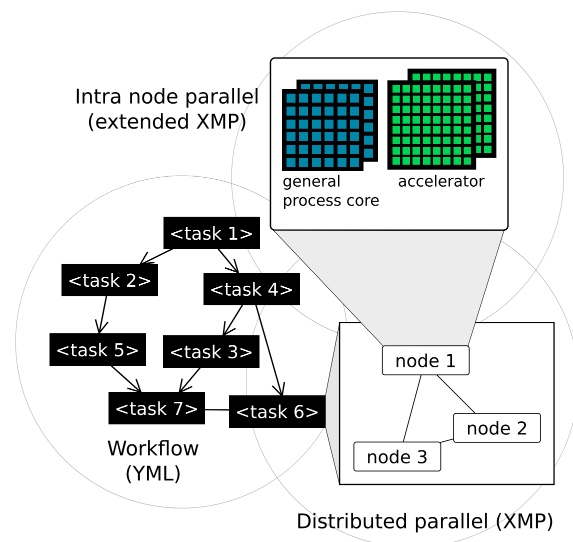


図 1　XMP/YML によるマルチ SPMD プログラム開発実行環境の概要

図 1 に示すように，マルチ SPMD プログラミングモデルにおいては，複数の分散並列プログラムがワークフローのタスクとして同時に実行される．分散並列プログラムは，アーキテクチャに応じて OpenMP や GPGPU により，さらに並列化される．

1　理化学研究所計算科学研究機構
RIKEN Advanced Institute for Computational Science
a)　miwako.tsuji@riken.jp

ⓒ 2017 Information Processing Society of Japan

表 1 階層的アーキテクチャと想定するプログラミングモデル，およびプログラミングモデルを実現するソフトエア，コンパイラやライブラリ

アーキテクチャ	プログラミングモデル	ソフトウエア
クラスタ クラスタのクラスタ	ワークフロー	YML
密結合なノードグループ	分散並列	XMP, MPI など
メニーコア CPU メニーコア CPU 内のコアグループ アクセラレータ	共有メモリのスレッド並列, GPGPU など	OpenMP, OpenACC など

このモデルをに基づいてアプリケーションを平易に記述するために，本研究では**表 1** に示すように

- ワークフローの記述および実行には YML [1]
- 分散並列プログラムの記述およびコンパイルには並列プログラミング言語 XcalableMP (XMP) [5][9]

を採用し，これらを拡張してマルチ SPMD プログラム開発実行環境を実装した．また，ワークフロースケジューラが分散並列タスクを起動し，管理するためのミドルウエア・ライブラリ `OmniRPC-MPI` を作成した．

YML は，ベルサイユ大学で開発されたワークフロー開発実行環境であり，タスクどうしの依存関係を記述するワークフロー記述言語 YvetteML をサポートしている．この依存関係は有向グラフに変換され，YML のワークフロースケジューラは，グラフにしたがってタスクを実行する．

XMP は，XMP ワーキンググループによって仕様が定義されている並列プログラミング言語およびそのコンパイラである．XMP は逐次のプログラムに指示分を挿入することで，既存の逐次プログラムを平易に並列プログラムに書き換えることができる．XMP コンパイラは，ソース・トゥ・ソース・コンパイラであり，C/Fortran と XMP 指示分からなるソースを，XMP ランタイムライブラリ呼び出しを行う C/Fortran のソースコードに変換する．XMP ランタイムライブラリは，実行時に内部で MPI 関数などを呼び出し，通信を行う．XMP においては，データおよび処理の分散は，テンプレートと呼ばれる仮想配列を定義することで指定される．本開発実行環境では，XMP によって既述されたタスクに対して，行列などのデータ分散が必要な入力もしくは出力がある場合，テンプレートの既述に基づき，自動的にデータの分散を行うことができる．

2.2 想定される実行環境

本稿では，とくにバッチシステムを持つ大規模なシステムを想定する．ユーザは，実行したいジョブやその要求ノード数，時間などをジョブスクリプトに既述し，ジョブをキューに投入する．バッチシステムは，システムの運用

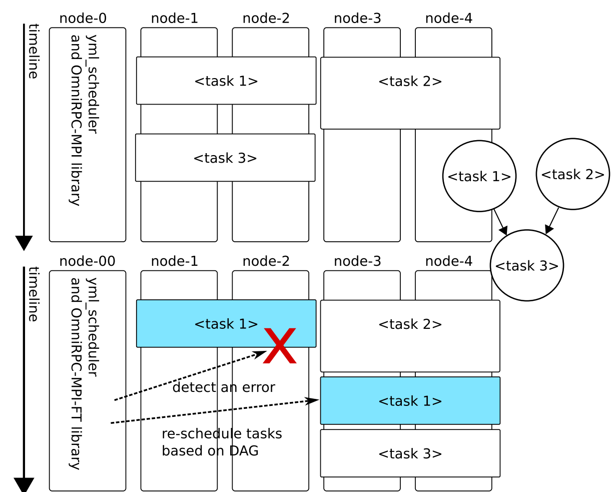


図 2 耐故障 XMP/YML の概要．上は障害が発生しなかった場合，下は障害がノード 2 に発生し，task1 が完遂しなかった場合．task1 は DAG にしたがってリスケジュールされ再実行される

ポリシーにしたがって，ジョブに計算資源を割り当てる．しばしば，大規模システムにおけるバッチジョブのスケジューラは，ジョブの用いるノードやネットワークになんらかの障害が生じた場合，その障害が一部であっても，ジョブが実行しているすべてのプロセスを停止させる．村井ら [6] は，一部に障害が生じた場合でもジョブ全体は停止せずに残りのノードでジョブを継続するために，新たなバッチジョブのスケジューラを提案した．これは，マスタープログラムが `MPI_Comm_spawn` によってワーカープログラムを起動するマスターワーカープログラムモデルを想定しており，バッチジョブのスケジューラは以下のように振るまう：

- ワーカープログラムが使用しているノードに障害が起こった場合も，`mpiexec` は継続する
- あるワーカープログラムが使用しているノードに障害が起こったとき，そのワーカープログラムの他のプロセスは停止し，関連するノードはジョブが終了するまで使用不可になる．しかし，それ以外のワーカープログラムおよびマスタープログラムは，継続する．

3. 耐故障マルチ SMPD プログラミング開発実行環境と大規模システムへの応用

3.1 耐故障マルチ SMPD プログラミング開発実行環境

本研究の目的は，アプリケーションのソースコードを変更することなく，耐障害性を実現するマルチ SMPD プログラミング開発実行環境を実現することである．そのために，なんらかの障害によってタスクの実行が失敗しても，失敗したタスクを障害の起こっていない別のノードで再度実行することで障害からの回復が可能なように，マルチ SMPD プログラミング開発実行環境を拡張した [3]．

図 2 に示すように，ワークフロー実行中に障害が生じてタスクが失敗したとしても，ワークフロースケジュー

ⓒ 2017 Information Processing Society of Japan

ラがこれを検知し，失敗したタスクを再度スケジューリングし実行することで，自動的に障害からの回復が可能になる．このために YML ワークフロースケジューラおよび OmniRPC-MPI ミドルウエアに以下のような拡張を行った：

障害の検知と OmniRPC-MPI ライブラリ

OmniRPC-MPI は，YML のワークフロースケジューラおよびワーカープログラムによって使用されるライブラリであり，OmniRPC[2] の拡張である．従来のリモートプロシージャコール (Remote Procedure Call, RPC) ライブラリである OmniRPC が逐次のワーカープログラムをサポートしたのに対し，OmniRPC-MPI は，内部で MPI による通信を行うような並列型のワーカープログラムを実行することができる．

スケジューラ側では，OmniRPC-MPI は，pthread により 2 つのスレッドを起動する．1 つめのスレッドは，リモートプログラムを起動し，YML スケジューラの要求に応じて，リモートプログラムにタスクの実行のリクエストを送信する．もう 1 つのスレッドは，クライアントプログラムとリモートプログラムの間の通信を監視し，リモートプログラムからタスクの終了を知らせる信号が送られてきたらこれを処理する．

耐障害性を実現するために，OmniRPC-MPI を拡張し，障害の検出を可能にした [7]．以下ではこれを，OmniRPC-MPI-FT と呼ぶ．障害の検出のために，ワーカー側から一定間隔でハートビートを送信し，マスターはこれが一定時間以上途切れた場合に，障害があったと判断する．スケジューラからワーカーの利用可能性を問い合わせるための API も実装した．

障害からの回復とワークフロースケジューラ

YML ワークフロースケジューラは，API を通して，OmniRPC-MPI からタスクの終了を通知される．OmniRPC-MPI-FT は，前述の方法で障害を検知した場合，同じ API を通してタスクの終了の代わりに障害を通知し，タスクのステータスを変更する．ワークフロースケジューラは，これにより，そのタスクを，再び実行すべきタスクのキューに投入する．現在のマルチ SMPD プログラミング開発実行においては，データはネットワークファイルシステム上に存在し，各リモートプログラムが MPI-IO などによりデータを読み書きすることで，タスク間のデータ入出力がなされる．各タスクの終了時がチェックポイントのような役割を果たし，その時点でのデータの保管が行われるため，失敗したタスクに対する入力データはファイルシステムに保持されたままであり，再実行時に使用することができる．

3.2 大規模システムへの応用

表 2 に本研究でターゲットとした京コンピュータの仕様を示す．また，京コンピュータのファイルシステムは，ジョブの入出力などのユーザデータを格納するグローバルファイルシステムと，計算中に使用されるローカルファイルシステムからなる．現状のマルチ SMPD プログラミング開発実行環境では，タスク間のデータの入出力は，ローカルファイルシステムへのデータの読み書きを通して行われる．前述のように，タスク内の各プロセスへのデータの分散は，XMP のテンプレートの定義に基づいて自動的に行われるため，アプリケーション開発者がとくに記述する必要はない．

表 2 京コンピュータ

CPU	Fujitsu SPARC64VIIIfx, 8core, 2.0 GHz
メモリ	16GB, 64GB/s
コンパイラ	Fujitsu Compiler 1.2.0-21
ネットワーク	Tofu Interconnect 5GiB/s x 2

ワーカープログラムは， pthread のサブスレッドから，マスターに向けて一定間隔でハートビートを送信する．しかし，ワーカーが主に行っている処理の内容によっては，サブスレッドにプロセッサが渡らなかったり，通信資源が主スレッド側の処理に占有されるなどの事情で，一定間隔でハートビートを届けることが困難な場合があると予想される．とくに，現時点の実装では，タスクに対するデータの入出力に MPI-IO を用いており，この処理において，ハートビートの送信が困難になる恐れがある．

そこで，予備実験として，マスタープログラムが n プロセスにより実行されるワーカープログラムを 1 つのみ起動する単純なアプリケーションに対して，

- ワーカーが MPI-IO により行列の書き込みを行う，
- ワーカーが単純な計算のみを行う，

と同時にサブスレッドから以下のように一定間隔でマスターに向けてパケットを送信した場合の，マスター側の受信間隔を調査した：

```
// worker 側の MPI rank 0 のサブスレッドの処理
for(itr=0; itr<nitr; itr++){
  write(fd, &ack, 1); // send
  usleep(100000); // sleep 0.1 sec
}
```

ワーカーのプロセス数は 4096 (512 ノード)，パケット送信の間隔は 0.1 秒，MPI-IO により書き込みされた行列は double 型で 32768x32768 (1 プロセスに 512x512 を分散して保持) とした．書き込みは 10 回繰り返して行った．

図 3 に，ワーカーのメインスレッドが IO もしくは計算を行い，サブスレッドがハートビートを送信した場合の，マスター側のハートビートの受信間隔を示す．図から，ワーカーが計算を行っている裏で送信されたハートビートは，

ⓒ 2017 Information Processing Society of Japan

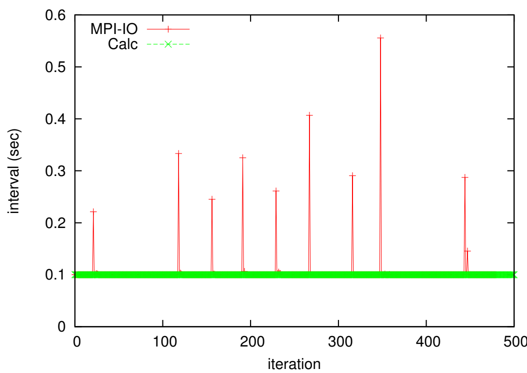


図 3 予備実験．ハートビートの受信間隔とメインの処理

一定間隔でマスターに受信されているのに対して，ワーカーが MPI-IO を行っている裏で送信されたハートビートは，しばしば受信間隔が不規則になっていることがわかる．この場合，マスター側はハートビートが遅れているのか，送られていない (すなわちワーカーに障害が発生している) のかを正しく判断することができない．

このような現象はタスクへの入出力以外でも発生する可能性があるが，まずはタスクへの入出力時にハートビート間隔が不規則になった場合でも，マスターが正しく状況を把握することができるようにする必要がある．ワーカーは MPI-IO の開始および終了をマスターに通知し，マスターはこの間はハートビートの間隔が長く開いた場合もワーカーを正常稼働中であるとみなすようにした (**図 4**)．ハートビートの種類は以下のように拡張された：

- 0: 通常，
- 1: IO 開始，もしくは IO 途中，
- 2: IO 終了．

通常，ハートビートは一定間隔で送られるが，IO の開始前にはこの間隔を無視して，IO 開始信号を含んだハートビートを強制的に送信する．これは，IO がはじまってしまうと，信号を送ることができなくなることがあるためである．通常のハートビートは，ワーカープログラムの MPI ランク番号 0 のプロセスからフォークされたサブスレッドによって送られるが，この信号は MPI-IO がはじまる前にすぐに送らなければならないため，スレッド切り替えのオーバーヘッドを防ぐために，メインスレッドから送られる．IO 終了後は，終了信号を送るが，開始信号と比較して緊急性を持たないので，メインスレッドは信号の値のみを変更し，サブスレッドが通常の間隔で送信時間になったのならばこのハートビートを送信する．

3.3 MPI-IO

また，大規模システムにおけるワークフローの IO オーバーヘッドの評価の一助とするために，MPI-IO の性能を評価した．

図 5 に 8192x8192, 16384x16384, 32768x32768 の double

```
 1 int task1(int argc, char **argv)
 2 {
 3   int __myrank;
 4   MPI_Comm_rank(MPI_COMM_WORLD, &__myrank);
 5
 6   {
 7     extern void _start_input_sig();
 8     extern void _end_input_sig();
 9     if(__myrank==0) _start_input_sig();
11     type_import(.., Matrix_improt, ..);
12     if(__myrank==0) _end_input_sig();
13   }
14
15   {
16     // =====================================
17     //  task source code
18     // =====================================
19   }
20
21   {
22     extern void _start_output_sig();
23     extern void _end_output_sig();
24     if(__myrank==0) _start_output_sig();
25     type_export(..., Matrix_export, ...);
26     type_export(..., Matrix_export, ...);
27     if(__myrank==0) _end_output_sig();
28   }
29
30   return 0;
31 }
32
33 void _start_output_sig()
34 {
35   omrpc_io_sig = 1;  // 信号を IO 開始として次のハー
36   omrpc_send_mpi_io_ack(); // トビートをすぐ送る
37 }
38
39 void _end_output_sig()
40 {
41   extern pthread_mutex_t   omrpc_ft_mutex;
42   extern pthread_cond_t    omrpc_ft_cond ;
43   extern char              omrpc_io_sig;
44
45    // 次のハートビート信号を IO 終了としておく
46    // 次の信号の時間になったら送られる
47   pthread_mutex_lock(&omrpc_ft_mutex);
48   omrpc_io_sig = 2;
49   pthread_mutex_unlock(&omrpc_ft_mutex);
50 }
```

図 4 タスクの例．リモートプログラムは，マスタープログラムからの要求に応じたタスク関数への呼び出しを行う

型の行列を，16x16, 32x32, 64x64 プロセスに二次元に分散した場合の，MPI-IO 書き込みの時間を示す．実験は，フラット MPI で行われた．各プロセス数・行列サイズ組について，5 回の独立な測定を行い，それぞれの時間および平均を示した．なお，この実験における複数の試行は，同時には行われない．図から，MPI-IO 書き込みの時間は通常で数秒程度だが，しばしばその数倍以上の時間を要する場合があり，ばらつきが大きいことがわかる．

ⓒ 2017 Information Processing Society of Japan

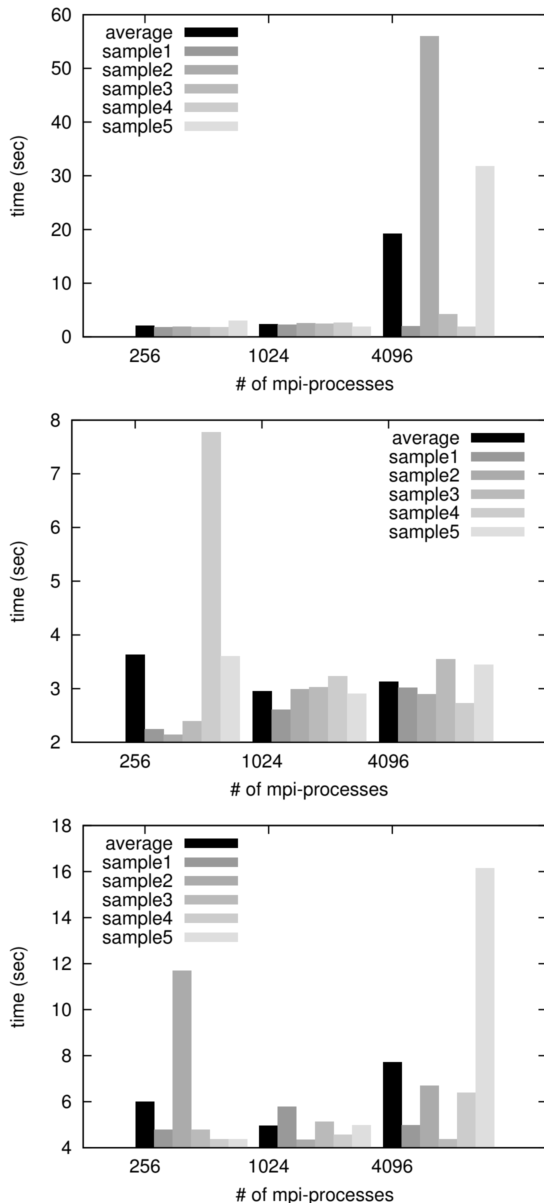


図 5　行列サイズが上から 8192x8192, 16384x16384, 32768x32768 を二次元分割したときの MPI-IO (write) の時間．グレーは各サンプルでの時間，黒は 5 つのサンプルの平均を示す．

亮へのデータの入出力については，MPI-IO 以外の代替手法を検討する必要があるが，これは本稿では行わず，今後の課題である．

4. 実験

4.1 エラーシナリオ

京コンピュータにおける障害の発生率は非常に低い．そこで，大規模システムにおいてエラーが発生した場合のマルチ SPMD プログラミングモデルにおける知見を得るために，適当なプロセス毎 MTBF を想定し，そこから 100 ミリ秒ごとの故障確率を算出して，乱数を用いてプロセスを停止させる実験を行った．より多くのプロセスを使用している状態を調査するために，タスクはフラット MPI で実行されるものとした．

表 3 に仮定したプロセス毎の MTBF と，その場合の 100 ミリ秒ごとの故障確率を示す．

表 3　エラーシナリオ

プロセス毎 MTBF(sec)	1.8×10^6	3.6×10^6	7.2×10^6
100msec 毎故障確率	5.6×10^{-8}	2.8×10^{-8}	1.4×10^{-8}

表 4　ブロックガウスジョルダンにおける行列の分割方法

ブロック数	1^2	2^2	4^2	8^2
ブロックのサイズ	32768^2	16384^2	8192^2	4096^2
タスク数	3	18	108	696
最大並列タスク数	1	4	16	64

表 5　各タスクに割り当てられるプロセス数

プロセス数	512	1024	2048	4096	8192	16384
最大同時タスク数	32	16	8	4	2	1

4.2 テスト問題

テスト問題として正方行列の逆行列を求めるアルゴリズムであるブロックガウスジョルダン法を用いた．YML ワークフローのソースコードを**図 6** に示す．行列は `p x p` 個のブロックに分割される．対角上に位置する各ブロックの逆行列がタスク `inversion` で計算される．他のブロックは，計算された逆ブロックを用いてタスク `prodMat`，`mProdMat`，`prodDiff` などによって，更新される．図で，`//`で区切られた範囲のソースは，互いに並列に実行可能である．また，`par; do` で既述された繰り返しの内部は，互いに並列に実行可能である．ただし，ソース内の `wait` は，対応する `notify` を待つ必要がある．

前述のように，各タスクは XMP で既述され，分散並列プログラミングモデルにより並列に実行される．本実験では，ワークフロー全体が使用するプロセスの総数は 16384+1 プロセス (2048+1 ノード) に固定する．ただし，1 プロセスは，ワークフロースケジューラによって使用される．各タスクが使用するプロセス数や，ブロックのサイズ — 言い換えれば行列の分割数 — は変化させる．たとえば，1 タスクごとに 1024 プロセスを使用する場合，最大で 16 個のタスクが同時に実行可能である．

表 4 に，それぞれの試行における行列のブロック分割数と，そのときの各ブロックのサイズ，およびタスク総数を示す．タスク総数には図 6 では省略されている行列の初期化のタスクなども含まれる．**表 5** に，それぞれの試行におけるタスク毎プロセス数，およびそのときの同時に実行可能なタスク数の最大値を示す．

4.3 実験結果

4.3.1 パラメータ

ハートビートの間隔は，通常は 0.08 秒とし，タスク入出力時は 10 秒とした．

ⓒ 2017 Information Processing Society of Japan

4.3.2 ハートビートのオーバーヘッド

まずは障害検知のためのハートビートのオーバヘッドを評価するために 4.1 節で示したエラーシナリオを使用せずに，ハートビートを用いる場合と用いない場合でワークフローを実行した．

図 7 に，ハートビートを使用するときとしないときの実行時間を示す．各点は 3 回の試行の平均である．X 軸は各タスクに割り当てられたプロセス数を示す．Y 軸はワークフローの実行時間を示す．異なる実線および点線は，ブロック分割数を示す．

ハートビートを使用しないほうが高速な場合と，そうでない場合があるが，これはハートビートの有無よりも MPI-IO の速度のばらつきが原因であると考えられ，ハートビートのオーバーヘッドは MPI-IO 速度のばらつきの範囲に収まっている．

図 8 に，ハートビートを使用するときとしないときの，各ワークフローにおけるタスクの実行時間の平均を示す．ただし，タスクの実行時間にタスクへのデータ入出力は含まない．タスク数がおおいワークフロー (たとえば，行列を 8x8 ブロックに分割した場合) は，各タスクの平均実行時間は短くなることに注意されたい．図から，タスクの処理の入出力以外の処理に対して，ハートビートのオーバーヘッドはごくわずかである．

4.3.3 エラーを起こした場合の評価

続いて，4.1 節で示したエラーシナリオを使用した場合について述べる．4.2 節で述べた各ブロック数，タスク毎プロセス数の組合わせに関して，4.1 節で示した各想定 MTBF で 10 回の試行を行った．

図 9 にそれぞれ 10 回の試行を行った場合の，ワークフローの成功回数 (完遂回数) を示す．X 軸はブロック数を示す．棒線の色分けはタスク毎プロセス数を示す．2.2 節で述べたように，想定するシステムは，あるワーカープログラム (タスク) が使用しているノードに障害が起こったとき，そのワーカープログラムの他のプロセスは停止し，これらのノードはジョブが終了するまで使用不可になる．すなわち，1 つのタスクに割り当てたプロセス数が多ければ，障害が起こったのちに，タスクを再スケジュールし続きを実行するための残りの使用可能なプロセス数が少なくなる．

極端な場合，すなわち 1 つのタスクにすべてのプロセス (16384 プロセス) を割り当てた場合，一度でも障害が起こった場合，すべてのプロセスが使用不可能になり，その後の処理を続行することはできない．図から，この場合の完遂率は，ゼロもしくは 3 割ていどである．完遂しているのは，ワークフローがすべて終了するまでに，たまたま一度もエラーが起こらなかった場合である．

しかし，ワークフローが適当に分割され，各タスクに全体の数分の一程度のプロセスを割り当てている実験では，比較的多くのエラーが起こるプロセス毎 MTBF1.8×10^6 秒の場合でも，ワークフローを完遂することができた．また，3.2 節で述べた対策により，タスクへのデータ入出力時にハートビートの間隔が長くなり，エラーを誤検出する，という現象も見られなかった．

図 10 に，それぞれのエラーシナリオを用いたときに，成功率が 50%を越えた場合の，実行時間の平均を示す．ブロック数が 1x1 などのタスクの数が少なく，個々のタスクが重い処理を行う場合，エラーから回復できたとしても，実行時間の増加は大きくなる．2x2 ブロックあるいは 4x4 ブロックなどのタスクの数が中程度のとき，エラーを考慮しない場合と比較してタスクあたりのノード数が小さいほうが，効率的にワークフローが実行される．8x8 ブロックのときは，エラーを考慮しない場合でもタスクあたりのノード数が小さいほうが高速だった，エラーを考慮しても同様の傾向であった．これは，そもそも，このとき各ブロックサイズは 4096x4096 であり，比較的小さいノード数が適切であるからであると考えられる．

5. おわりに

耐故障マルチ SMPD プログラミング開発実行環境を京コンピュータにおいて評価した．タスクへのデータ入出力時に，故障検知のためのハートビートが適切に受送信できないことがあり，対策を検討した．提案手法では，ワーカーからマスターにハートビートを送信し，ワーカーの状態を信号情報に含めることで，エラーの誤検出を防いでいる．各タスクのデータの出力は耐故障のためのチェックポイントとしての役割も果たし，現在の実装では MPI-IO によって行われているが，MPI-IO はオーバーヘッドが大きく，また所要時間も不規則であることから，今後の課題として別の方法を検討する必要がある．

謝辞

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ある．

参考文献

[1] O. Delannoy, N. Emad, and S. Petiton. Workflow global computing with yml. In *The 7th IEEE/ACM International Conference on Grid Computing*, pp. 25–32, 2006.

[2] M. Sato, M. Hirano, Y. Tanaka, and S. Sekiguchi. Omnirpc: A grid rpc facility for cluster and global computing in openmp. In *International Workshop on OpenMP Applications and Tools, 2001*, pp. 130–136, 2001.

[3] M. Tsuji, S. Petiton, and M. Sato. Fault tolerance features of a new multi-SPMD programming/execution environment. In *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware, SC15*, pp. 20–27. ACM, 2015.

[4] M. Tsuji, M. Sato, M. Hugues, and S. Petiton. Multiple-SPMD programming environment based on PGAS and

ⓒ 2017 Information Processing Society of Japan

workflow toward post-petascale computing. In *Proceedings of the 2013 International Conference on Parallel Processing (ICPP-2013)*, pp. 480–485. IEEE, 2013.
[5] XcalableMP. http://www.xcalablemp.org/.
[6] 村井均, 南一生, 横川三津夫, 梅田宏明, 佐藤三久, 辻美和子, 稲富雄一, 青柳睦, 中島真. スーパーコンピュータ「京」におけるマスタ・ワーカ型プログラミングモデルの検討. 情報処理学会研究報告, No. 2013-HPC-138, pp. –. 情報処理学会, 2013.
[7] 辻美和子, 佐藤三久. マルチ SPMD 環境における耐故障性実現に向けた OmniRPC-MPI の拡張. 情報処理学会研究報告, No. 2014-HPC-146, pp. –. 情報処理学会, 2014.
[8] 辻美和子, 佐藤三久, M. Hugues, S. Petiton. 並列コンポーネントを統合する階層型並列プログラミングモデル. 情報処理学会研究報告, No. 2012-HPC-135, pp. 1–10. 情報処理学会, 2012.
[9] 李珍泌, 朴泰祐, 佐藤三久. 分散メモリ向け並列言語 xcalablemp コンパイラの実装と性能評価. 情報処理学会論文誌コンピューティングシステム (ACS), 3(3):153–165, 2010.

```
par(k:=0;p-1); do
  par
    if (k neq 0) then
      wait(prodDiffA[k][k][k-1]);
    endif
    compute inversion(A[k][k],B[k][k]);
    notify(bInversed[k][k]);
//
    if (k neq p-1) then
      par (i:=k+1; p-1); do
        wait(bInversed[k][k]);
        compute prodMat(B[k][k],A[k][i]);
        notify(prodA[k][i]);
      enddo
    endif
//
    wait(bInversed[k][k]);
    par(i:=0;p-1);  do
      if(i neq k) then
        compute mProdMat(A[i][k],B[k][k],
                                B[i][k]);
        notify(mProdB[k][i][k]);
      endif
      if(k gt i) then
        compute prodMat(B[k][k],B[k][i]);
        notify(prodB[k][i]);
      endif
    enddo
//
    par(i:=0;p-1);  do
      if (i neq k) then
        if (k neq p-1) then
          par (j:=k+1;p-1);  do
            wait(prodA[k][j]);
            compute prodDiff(A[i][k],A[k][j]
                                      ,A[i][j]);
            notify(prodDiffA[i][j][k]);
          enddo
        endif
        if (k neq 0) then
          par(j:=0;k-1); do
            wait(prodB[k][j]);
            compute prodDiff(A[i][k],B[k][j]
                              ,B[i][j]);
          enddo;  endif;  endif;  enddo
  endpar
enddo
# inversion(A,B) : B=A^-1
# prodMat(A,B)   : B=BxA
# mProMat(B,A,C): C=-(BxA)
# prodDiff(B,A,C): C=C-(BxA)
```

図 6 ブロックガウスジョルダンのワークフロー．p はブロック数

ⓒ 2017 Information Processing Society of Japan

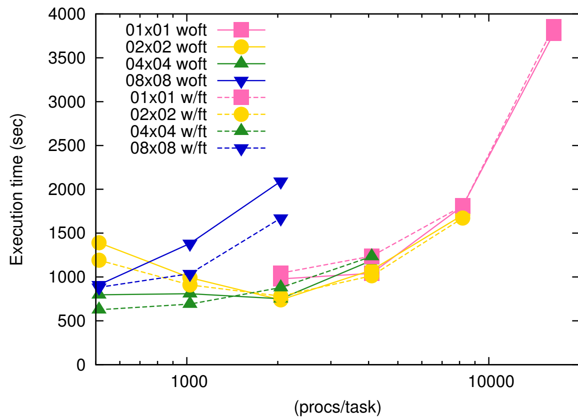


図 7 ハートビートを使用するとき (点線) としないとき (実線) の総実行時間

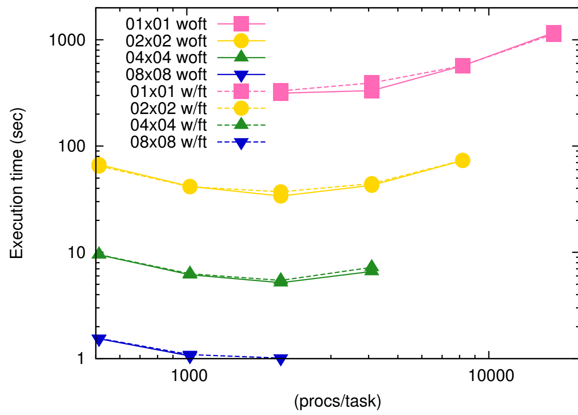
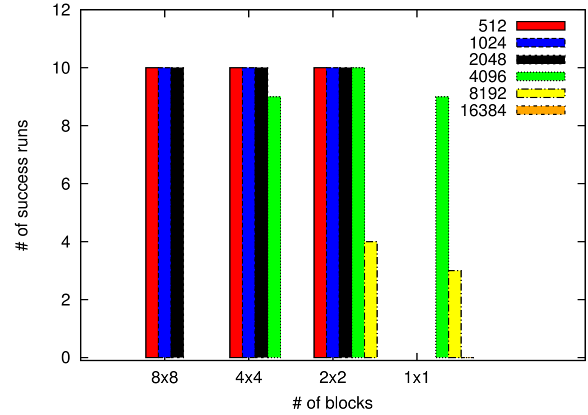
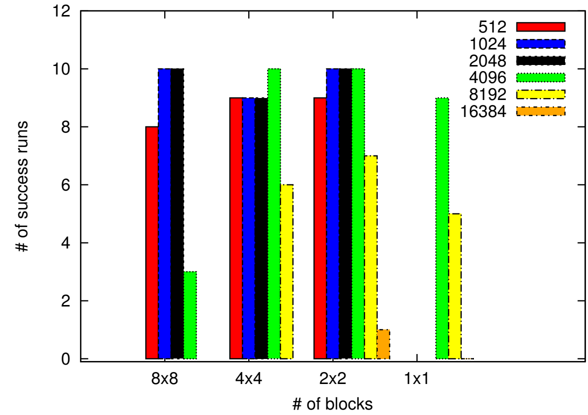


図 8 ハートビートを使用するとき (点線) としないとき (実線) の各タスクの実行時間の平均 (タスクへのデータ入出力は除く)





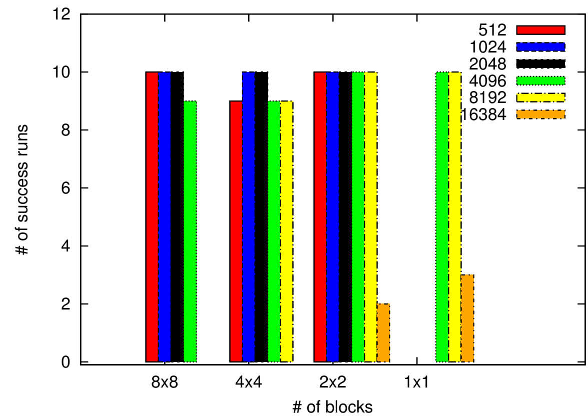


図 9 想定 MTBF が上から $1.8, 3.6, 7.2 \times 10^6$ のときのワークフローの成功数 (最大が 10)．X 軸はブロック数

ⓒ 2017 Information Processing Society of Japan

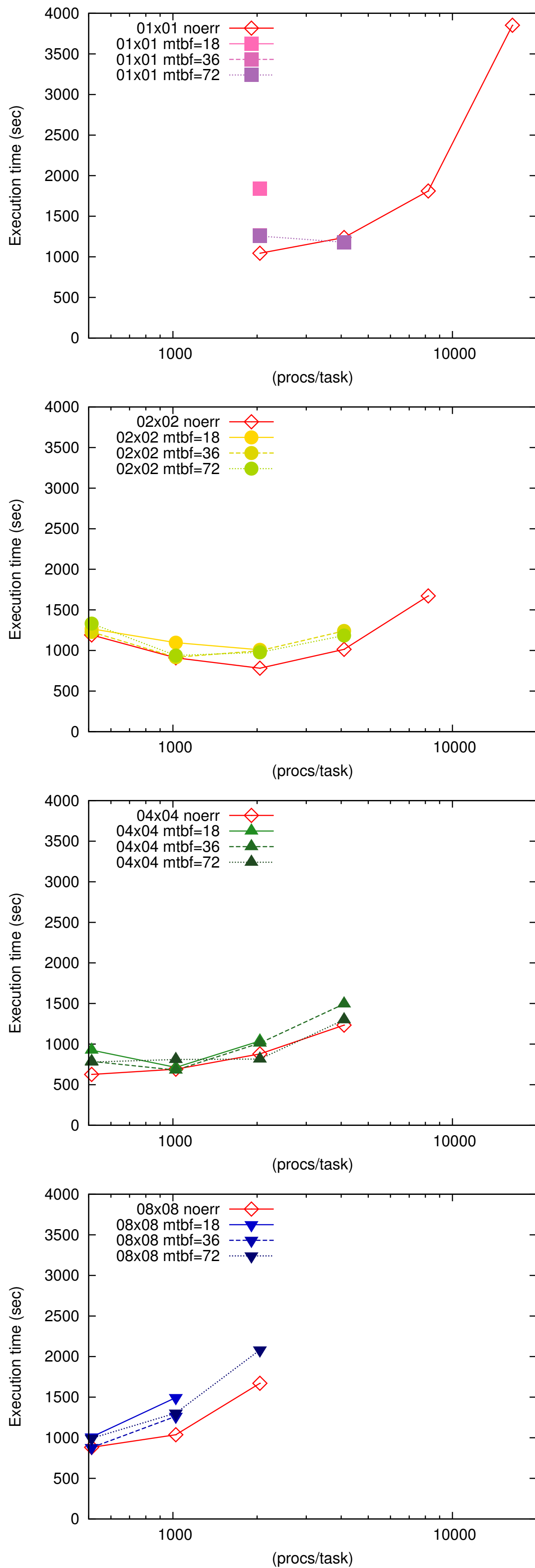


図 10 エラーが起こったときの実行時間．最終的にワークフローを完遂した指向のみの平均値．ただし，完遂率が 50%以下のものは除いている．上からブロック数が 1^2,2^2,4^2,8^2 のとき．X 軸はタスク毎のプロセス数．

ⓒ 2017 Information Processing Society of Japan