Vol.2014-ARC-213 No.21
Vol.2014-HPC-147 No.21
2014/12/10

予備ノードを利用した故障後の実行継続手法の検討と評価

吉永 一美[1,a)]　亀山 豊久[1]　堀 敦史[1]　石川 裕[1]

概要：ハードウェア規模の増大によりシステムの MTBF が短縮されるエクサスケール環境では，耐故障性の確保が重要な課題となっている．耐故障性の実現に向け，アプリケーションと連携した故障対策を行う Fault Resilience が注目され，多くの研究が進められている．我々は Fault Resilience なエクサスケール環境において実現される耐故障機構を用いて，どのように実行を継続することが効果的であるか，アプリケーションの実行継続手法についての研究を進めている．本論文では，故障後も効率的な実行継続を実現する手法を確立するために，実行継続手法の評価と検討を行う．そして予備ノードを利用した手法について，3 種類の予備ノード利用方式を提案しその性能差を検討するとともに，実アプリケーションへの適用に向けた議論を進める．

An Evaluation of Fault Mitigation Method Using Spare Nodes

Kazumi Yoshinaga[1,a)]　Toyohisa Kameyama[1]　Atsushi Hori[1]　Yutaka Ishikawa[1]

Abstract: In the upcoming Exa-scale era, faults could happen more frequently than ever, and thus, fault tolerance (FT) is getting more important. Although many FT mechanisms to survive failures has been proposed so far, there is no discussion how a job should survive from failures. In this paper, we explore and discuss three fault mitigation methods how to survive from a failure using spare nodes without loosing execution efficiency. Finally, it is discussed to apply those proposed method to real applications.

1. はじめに

今後の実現が予想されるエクサスケール環境においては，構成するハードウェアが膨大なものとなるためにシステムの故障発生率が増加し，システム全体の MTBF(平均故障間隔/Mean Time Before Failure) が短くなることが懸念されている．そのため，エクサスケール環境の実現に向け，耐故障性の確保は重要な課題となっており，多くの研究が進められている．

現在の HPC 環境において広く用いられている故障対策手法として，システムレベルでのチェックポイント・リスタート手法を用いたものがある．この手法では，システムにより定期的にアプリケーションのスナップショット (チェックポイント) をストレージへと保存し，故障発生後はそのスナップショットから状態を復元してアプリケーションを再度実行することで，耐故障性を実現する．しかしシステムが大規模化するエクサスケール環境では，保存されるスナップショットのデータ量が大幅に増加する．そのため，チェックポイント作成時のストレージへの書き込み，及びリスタート時のデータ読み込み時間が長時間となる．一方，システムの MTBF は前述の通り短くなるために，故障に対応するためにはチェックポイント間隔を短くする必要が生じる．その結果，故障が発生しない正常な状態の大半をチェックポイント・リスタートの処理に費やさなければならず，チェックポイント作成時間が MTBF を上回る最悪の場合では，アプリケーションの実行が完全に停止することも考えられる．このため，エクサスケール環境においては単純なシステムレベルのチェックポイント・リスタート手法は破綻すると言われている [1].

以上のような背景から，故障への対応をシステムに一任せず，アプリケーションと連携した効率的な故障対策を実現する，Fault Resilience という考え方が注目されている．例えば，チェックポイント・リスタート処理において

[1] 理化学研究所 計算科学研究機構
RIKEN AICS
[a)] kazumi.yoshinaga@riken.jp

は，アプリケーションと連携し実行再開に必要なデータのみを保存することで，チェックポイント時間の短縮が実現できる．また，故障の発生をアプリケーションに通知することで，これまでと異なり故障発生時に実行を中断するのではなく，故障発生後も実行を継続する方法が提案されている．この方法は，実行を中断しチェックポイントの状態から再実行する手法が rollback と呼ばれているのに対し，rollforward による故障対策と呼ばれている [2].

ULFM(User Level Failure Mitigation)[3], [4] は，この rollforward による Fault Resilience 環境を実現するためのミドルウェアの一つである．ULFM は OpenMPI をベースに故障対策機構を追加した MPI 実装であり，MPI Forum の Fault Tolerance Working Group[5] により設計され，テネシー大学を中心として開発が進められている．ULFM が対象とする故障は Process Failure であり，実行中にノード故障などでプロセスが停止した場合に，故障のプログラムへの通知，停止プロセスを除いた新たなコミュニケータの生成など，ユーザレベルでの故障対策を実現するための機構を実装し，利用するための API を提供している．

以上のように，エクサスケール環境での耐故障性を実現するための研究は多くなされている．その一方で，研究成果により実現された手法を用いて，どのように実行を継続することが効果的であるかについては，まだ研究がなされていないのが現状である．

そこで我々は，ULFM のようなユーザレベルでの rollforward な故障対策手法と，保存されたチェックポイントを用いた，効率的なアプリケーションの実行継続手法についての研究を進めている [6], [7]. 研究対象としてステンシル計算アプリケーションを想定し，これまでに予備ノードを利用した実行継続手法について，その性能の調査を進めてきた．故障したノードの代わりに予備ノードを利用することで，プログラムを大きく変更すること無くロードバランスを維持した実行継続を実現可能である．しかし，ネットワーク的に離れたノード間での通信が生じ，実行継続後には通信衝突が発生することで，性能が低下することを確認している．

本論文では，故障後も効率的な実行継続を実現する手法を確立するために，実行継続手法の詳細な評価と検討を行う．まず，第 2 章において，故障後の実行継続を実現する手法についてその設計指針を述べ，3 種類の手法を提案しそれぞれの性能について議論を行う．次に第 3 章において，予備ノードを利用した実行継続手法について，予備ノードの利用方式による性能の差を述べ，実アプリケーションへの適用に向けた有効性について論ずる．そして第 4 章にて関連研究について述べ，第 5 章でまとめを行う．

2. 故障後の実行継続手法

2.1 設計指針

故障後の実行継続手法を提案するに当たり，我々は以下の点を考慮して手法を設計し評価する．

プログラムの変更が少ないこと

故障発生後の実行継続を実現するためには，アプリケーションプログラマによるコードの変更が不可欠である．実現性を考慮し，コードの変更はチェックポイント/リスタート処理などの故障対策に必須な部分のみに抑えることが望ましい．

故障対策による実行時間の増加が少ないこと

アプリケーション利用者にとって重要なのは，ジョブのターンアラウンドタイムが短いことである．実行継続に必要な故障対策処理などの影響により，ジョブ実行時間は非故障時と比較して増加は免れないが，可能な限りこの増加を抑えた手法を実現する．

2.2 対象として想定する環境

現在の一般的な環境では，アプリケーションが利用しているノードに故障が発生した場合，アプリケーションの実行が停止してしまう．本論文では ULFM などを利用した，ノード故障発生後もアプリケーションが停止せず，実行の継続が可能なシステムを対象としている．また，チェックポイントは全ノードからアクセス可能な共有ストレージ上に保存され，保存されたチェックポイントからのリスタートが可能な環境を想定している．

実行継続手法を適用するアプリケーションは，HPC アプリケーションで広く用いられているステンシル計算を想定し，2 次元 5 点ステンシル計算を用いて議論を行う．また，システムのネットワークトポロジは XY ルーティングを持つ 2 次元メッシュネットワークとして議論を進める．

2.3 実行継続手法を用いたジョブ実行時間

実行継続手法を適用したジョブの実行時間について議論するために，故障が発生しない場合の実行時間が下の式で表される 2 次元 5 点ステンシル計算ジョブを考える．

$$T_{Elapse} = T_{Done} + T_{Togo} \quad (1)$$

ここで，T_{Done} までの処理が完了した後に故障が発生したと仮定し，実行継続手法を適用した場合の実行時間について議論する．実行継続手法として，ジョブの再投入・故障ノードの除去・予備ノードの利用という 3 手法を提案し，それぞれの性能を述べる．

2.3.1 ジョブの再投入による実行継続手法

ジョブの再投入による実行継続は，ジョブが故障に遭遇した場合一旦実行を終了し，再度ジョブをシステムに投入

することで，チェックポイントから復帰して残りの処理を実行するものである．この手法は，ノード故障発生時にアプリケーションが停止する現在の一般的な環境において，チェックポイント/リスタートを用いた故障対策手法として広く用いられているものである．本手法を利用した実行時間は式 2 にて表すことができる．

$$T_{Elapse} = T_{Done} + T_{Gap} + T_{Restart\_resub} + T_{Togo} \quad (2)$$

ここで，$T_{Restart\_resub}$ はチェックポイントからの状態の復帰に必要な時間，T_{Gap} はジョブを再度投入し実行が開始されるまでの時間である．大規模な並列計算機は一般的に専有利用ではなく多数のユーザにより共有されているため，ジョブを再投入してもすぐにスケジュールされ実行開始するとは限らず，待ち時間が生じる．また，京のようなファイルステージングを採用したシステムでは，ファイルの転送時間も必要である．これらを全て含めたものが T_{Gap} となる．

2.3.2 故障ノードの除去による実行継続手法

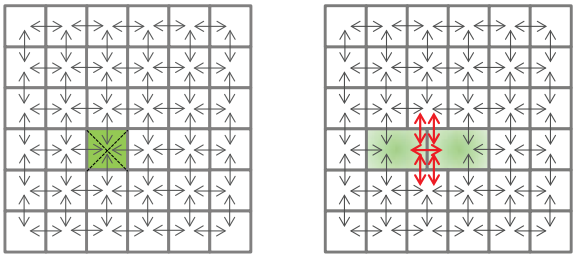
本手法は，故障が発生したノードを取り除き，残りの正常なノードのみを用いて実行を継続するものである．N 台を用いて実行している 2 次元 5 点ステンシルアプリケーションに対して，1 ノード故障後に故障ノードの除去による実行継続手法を適用した場合の実行時間は，式 3 のようになる．

$$T_{Elapse} = T_{Done} + T_{Restart\_rm} + \alpha \times T_{Togo} \times \frac{N}{N-1} \quad (3)$$

ここで，T_{Resart_rm} はチェックポイントを読み込み状態を復帰するのに必要な時間を示す．故障した 1 ノードを除去するため，故障後の計算ノード数は $N-1$ ノードとなり，故障後の実行時間は理想的には $T_{Togo} \times \frac{N}{N-1}$ である．しかし，ノードを除去した影響によって更なる性能の低下が発生する．この性能低下率を $\alpha(\geq 1)$ として示している．

α の大きな要因となるのは，実行継続後のノード間のロードインバランスの発生と，アプリケーション内での通信パターンの変化である．**図 1(a)** に示されるステンシル計算ジョブの実行中に，×印の付いた濃い緑のノードに故障が発生した場合の，故障ノードの除去による実行継続例を**図 1(b)** に示す．この例では，故障したノード上で計算されていた領域を両横のプロセスに分担することで，故障ノードの除去による実行継続を実現している．そのため，領域を負担することになった 2 プロセスの計算量がその他のプロセスの 1.5 倍となってしまい，ロードインバランスが発生している．また，2 次元 5 点ステンシル通信は隣接する 4 プロセスとのみ通信を行うものであるが，故障ノードの除去による実行継続では図 1(b) のように通信パターンが変化し，通信性能が低下する．以上の理由から，理想的な実行時間から更なる性能の低下 α が発生する．

本手法の効率を向上させるためには，α を小さくすることが重要であるが，そのためにはロードバランスと通信パターンを考慮した実行継続処理の実装を，アプリケーションプログラマが行わなければならない．この改変作業では，プログラム内のコアとなる計算や通信部分にも手を入れる必要があり，プログラマへの負担は非常に大きなものとなる．故障台数が増加すれば更に複雑な処理が必要となり，変更するコード量が大幅に増大することから，本手法を採用した効率的な実行継続手法は現実的ではないと考える．

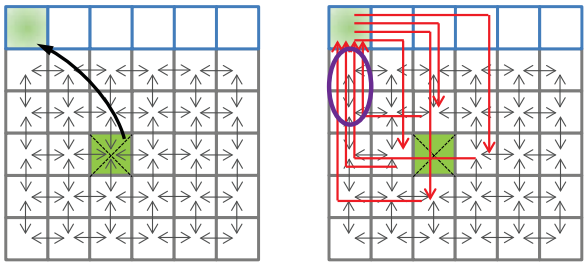


(a) 故障前の実行状態　(b) ノード故障後の実行継続例

図 1 故障ノードの除去による実行継続

2.3.3 予備ノード利用手法

本手法は，実際に計算に使うノードの他に予備ノードを確保してジョブを実行しておき，故障が発生した際にはこの予備ノードを用いることにより実行を継続するものである．**図 2(a)** は，予備ノード利用手法による実行継続例である．最上段に位置する水色の線で囲まれたノードが予備ノードであり，×印の付いた濃い緑のノードが故障ノードである．この例ではノード故障発生後，そのノード上が担当していた計算領域を予備ノードに担当させて継続実行することで，故障後のジョブの実行継続を実現している．



(a) 予備ノードを用いた実行継続例　(b) 実行継続後の通信パターン

図 2 予備ノードを利用した実行継続

N 台のノードを用いてジョブを実行し，そのうち N_{spare} 台を予備ノードとして確保した場合のジョブの実行時間は，以下の式 4 で表される．

$$T_{Elapse} = (T_{Done} + \beta \times T_{Togo}) \times \frac{N}{N - N_{spare}} + T_{spawn} + T_{Restart\_spare} \quad (4)$$

ジョブが確保した N ノードのうち，実際に計算に利用するノードは予備ノードを除いた $N - N_{spare}$ ノードである

ため，故障発生の前後に関わらず実行時間は $\frac{N}{N-N_{spare}}$ 倍になる．また，故障発生後には予備ノードの利用により後述する性能低下が発生する．この性能低下率を $\beta(\geq 1)$ として示している．$T_{Restart\_spare}$ は，他手法と同様にチェックポイントからのジョブの状態復帰処理に要する時間である．そして，T_{spawn} は予備ノード上に新たにプロセスを起動するために必要な時間である．この時間は，実行時に予備ノードにも予めプロセスを起動しておき，故障発生までは何も処理を行わずに，その予備ノードを利用することになった場合にのみ処理を行うように切り替える実装を行うことにより，削減することが可能である．

本手法では，故障発生の前後で計算に利用するノード数は変化しないため，故障ノードの除去による実行継続手法とは異なり，ロードバランスの維持が容易である．また，アプリケーション内での論理的な通信パターンも故障前後で維持することができる．そのため，アプリケーションの変更が少なく，実装が容易である．しかしその一方で，プロセスの配置されるノードが移動することにより，物理的な通信パターンを考えると故障前後で大きく異なるものとなり，通信性能の低下が発生する．**図 2(b)** は，図 2(a) のような実行継続を行った場合の，ノード間での通信を示したものである．図中の紫で囲んだ部分において 5 通信が同一経路上に存在しており，通信衝突が発生し通信性能が低下する可能性が生じる．これまでの研究 [6], [7] により，通信時間は経路を共有する最大通信数に比例して増大することを確認しており，これが β の大きな要因となる．

2.4 再投入と予備ノード利用の比較

予備ノード利用手法を利用した実行時間が，ジョブ再投入による実行時間よりも短くなるためには，式 2 及び 4 より，下式を満たす必要がある．

$$\begin{aligned} T_{Gap} + T_{Restert-resub} > T_{Done} \times \left(\frac{N}{N - N_{Spare}} - 1 \right) \\ + T_{Togo} \times \left(\frac{\beta \times N}{N - N_{Spare}} - 1 \right) \\ + T_{spawn} + T_{Restart\_spare} \end{aligned} \tag{5}$$

ここで，2.3.3 小節にて述べた通り，T_{spawn} は実装により削減することが可能である．また，チェックポイントからの復帰に必要な時間である $T_{Restart\_resub}$ と $T_{Restart\_spare}$ については，全てのプロセスがチェックポイントを読み込むアプリケーションであれば，同一になると考えられる．しかし，アプリケーション内の各プロセスが，故障前の状態に復帰するためのデータをメモリ上にも保持する実装では，この時間に差異が生じる．ジョブの再投入による手法では，一旦実行中のジョブを終了するためにメモリ上のデータはクリアされ，全プロセスにおいてストレージからの読み込みが必要である．一方，予備ノード利用手法では，故障前から変わらず同じノード上で実行されているプロセスにおいて，メモリからの復帰が可能となる．その結果ストレージからのデータ読み込み量が削減され，$T_{Restart\_spare}$ は $T_{Restart\_resub}$ よりも短時間にすることが可能になる．

以上により，予備ノード利用手法による実行継続がジョブの再投入手法よりも短時間での実行を実現するためには，(1) 性能低下率 β の軽減，(2) チェックポイントからの復帰時間 $T_{Restart\_spare}$ の短縮，の 2 点が重要となる．2.3.3 小節にて述べたように，性能低下の大きな要因は通信の衝突であり，同一経路上に存在し経路を共有する通信数により決定する．また，$T_{Restart\_spare}$ は前述の通りアプリケーションの実装によるが，チェックポイントをストレージから読み込むプロセス数を減少させることで短縮が期待でき，これは故障発生後に実行ノードが移動したプロセスの数により決定する．

そこで次章では，予備ノード利用手法について，予備ノードをどのように利用すれば高効率な手法が実現できるか，経路を共有する通信数と故障後に移動するプロセス数の 2 点に基づいた検討を行う．

3. 予備ノード利用手法の性能

本章では，予備ノードの利用方式として置換方式，1 次元スライド方式，2 次元スライド方式の 3 種を提案し，その性能について議論する．本章においても第 2 章と同様に，2 次元 5 点ステンシル通信と，XY ルーティングアルゴリズムを持つ 2 次元メッシュネットワークトポロジを想定して議論を進める．

3.1 置換方式

置換方式は，故障したノード上で実行されていたプロセスを予備ノードの一つへと移動し，実行を継続する方式である．2.3.3 小節において例示した**図 2** がこの方式を用いた実行継続である．本方式では，故障前と別ノードで動作するプロセスは，予備ノードへと移動したプロセスのみであり，その数は故障ノード数と同一になる．

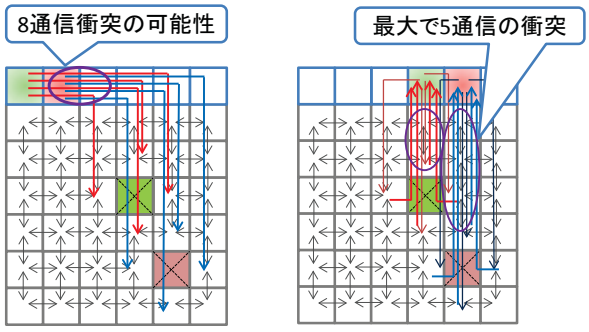
故障ノードが 1 ノードのみであれば，どの予備ノードを利用した場合も経路を共有する通信数は 5 となり，ノード配置に寄る性能差はない (通信先の少ない端領域に当たるプロセスを除く)．しかし，複数台の故障が発生した場合は，置換先予備ノードの選択によって性能が変化するため，本方式では置換先としてどの予備ノードを選択するかが重要である．

3.1.1 置換先ノードの選択による性能への影響

図 3(a) および**図 3(b)** は，どちらも同じ 2 ノードが故障した場合の置換方式を用いた実行継続である．まず図 3(a) は，左から順に予備ノードを選択する方式を採用したものである．この場合，2 台の予備ノードからの通信が最上行

で全て衝突しており，紫で囲んだ部分において 8 通信が同一経路上に存在し衝突する可能性がある．そのため通信時間は最大で通常時の 8 倍となる．この問題を解決するために，故障ノードと同一列にある予備ノードを利用するような配置方式を採用したものが図 3(b) である．同一列に存在する故障ノードが 1 台であると仮定した場合，このような配置による通信衝突数は下記の理由により最大で 5 に抑えることができる．

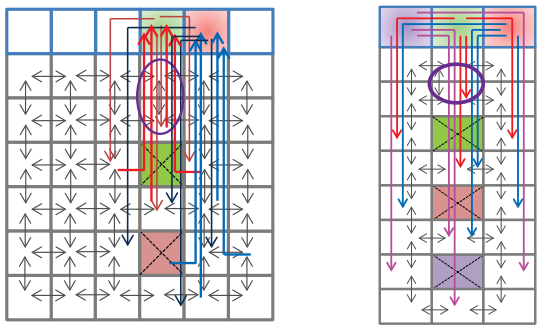
- 予備ノードから発生する通信のうち，X 方向での通信は，自ノードから左右隣接ノードへ向かう通信のみであり，互いに干渉しない．
- 予備ノードへの通信のうち，X 方向での通信は，故障前と変わらない左右隣接ノードへ向けた通信のみであり，衝突は発生しない．
- 衝突の可能性が発生するのは Y 方向の通信のみであるが，ここでプロセスが配置された予備ノードの列に着目すれば，予備ノードからの通信・予備ノードへの通信それぞれについて，以下のように最大 5 通信の衝突に抑えることができる．
  - 予備ノードへの通信は必ず 4 通信が存在し，正規のステンシル通信との衝突を含めて最大の衝突数は 5 である．
  - 予備ノードからの通信 (下方向への通信) は，自ノードと同列に 2 通信と，左右の列に 1 通信ずつが発生する．そのため，両隣列の予備ノードが利用されていたとしても，自ノードからの 2 通信と両隣からそれぞれ発生する 1 通信の，合わせて 4 本が同列に発生するのみであり，経路を共有する最大の通信数は正規のステンシル通信を含めて 5 通信となる．



(a) 左から順に置換先を選択　(b) 故障ノードと同一列に置換先を選択

図 3 複数ノード故障時のノード選択による通信衝突

本配置方式では，同一列に 2 ノード以上の故障が発生した場合の対応は不可能である．そのような場合は，X 方向の通信の衝突を防ぐために，可能な限り故障ノードの列に近い位置へと予備ノードを配置する．つまり，同列，右隣列，左隣列，2 列右，2 列左…とまだ利用されていない予備ノードを探していき，利用する予備ノードを決定する．この配置方式を採用した例が**図 4** である．この方式では，同一列に 2 ノードの故障が発生した**図 4(a)** の場合，最大の経路共有通信数は紫で囲んだ部分の 5 通信となり，1 ノード故障時と同性能を保つことができる．しかし同一列で 3 ノードが故障した**図 4(b)** の場合，最大経路共有通信数は紫で囲まれた部分の 7 通信となり，性能が低下している．同一列の故障が増えていけば更なる性能の低下が考えられる．



(a) 2 ノード故障後　(b) 3 ノード故障後

図 4 同一列での複数ノード故障発生時の通信

3.1.2 予備ノードを 2 辺に確保した置換方式

同一列が 3 台以上故障した場合の性能低下を改善するために，2 辺に予備ノードを確保する方式を提案する．これまでは最上行にあるノードを予備ノードとして利用していたが，本方式では最も右にある縦 1 列も予備ノードとして利用する．そして故障発生時には，基本的に故障ノードと同列の上にある予備ノードを利用するが，そのノードが既に利用されていた場合は同行の右にある予備ノードを利用する．その後は右隣列の上，左隣列の上，直上行の右，直下行の右…と利用する予備ノードを検索していく．

図 5 は，同一列に 3 ノードの故障が発生した場合における，予備ノードを 2 辺に確保した置換方式による実行継続例である．図 4(b) と故障ノードの位置関係は同じであるが，新たに右列に用意した予備ノードを利用することにより，最大経路共有通信数を 5 に低減している．

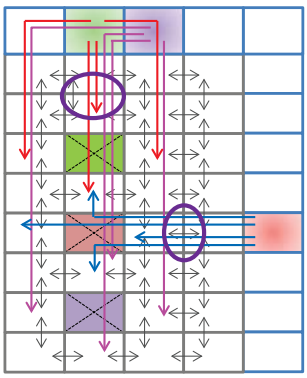
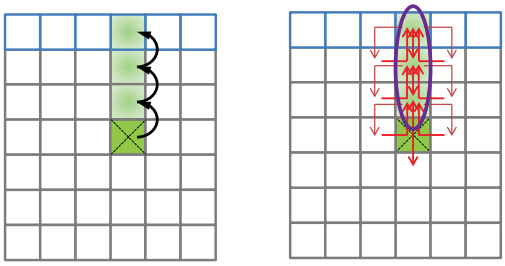


図 5 予備ノードを 2 辺に確保した場合の置換方式 (3 ノード故障後)

Vol.2014-ARC-213 No.21
Vol.2014-HPC-147 No.21
2014/12/10

3.2 1 次元スライド方式

1 次元スライド方式は，故障ノードが発生した際に，故障ノードと予備ノードまでの間にあるプロセス全てを予備ノード側へ 1 ノードずつスライドさせる方式である．**図 6(a)** に 1 次元スライド方式を採用した，予備ノード利用手法による実行継続を示す．×印の付いた濃い緑のノードが故障ノードであり，故障ノードより上のノードに配置されたプロセス全てをスライドさせることで，実行継続を実現している．そのため置換方式とは異なり移動プロセス数は故障ノード数よりも多くなる．一方で 1 ノードに故障が発生した場合の実行継続後の最大経路共有通信数は，**図 6(b)** に示すように 3 となり，置換方式と比較して通信時間の短縮が見込まれる．



(a) 1 ノード故障時の実行継続例　(b) 1 ノード故障後の通信衝突

図 6 1 次元スライド方式による実行継続

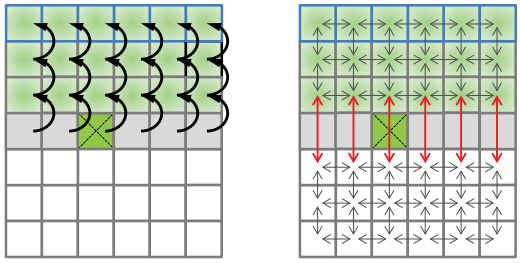
b

予備ノードが 1 辺のみにしか存在しない場合，1 次元スライド方式では同一列の故障に対応することができない．そのため，最悪の場合には 2 ノードの故障で実行継続が不可能になる．置換方式と同様に 2 辺に予備ノードを配置することで，少なくとも 3 ノードの故障に耐えることができるため，これを採用する．

3.3 2 次元スライド方式

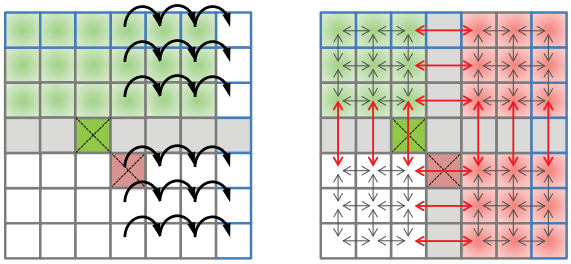
2 次元スライド方式は，故障が発生したノードと同行にあるノードが全て故障したとみなし，全列について 1 次元スライド方式の要領で予備ノード側へスライドさせる方式である．**図 7(a)** に 2 次元スライド方式による実行継続例を示す．×印の付いた濃い緑のノードが故障した場合，すべての列に対してプロセスをスライドする形で予備ノードを利用している．本方式の特徴は，**図 7(b)** に示されるように，故障発生後も通信に衝突が発生しないことである．そのため通信時間は故障前と比較しても変わらない．一方で，移動するプロセス数は 1 次元スライド方式よりも増加する．

予備ノードが 1 辺にしか存在しない場合，本方式では同一列に故障が発生した場合を除き，1 ノードの故障にしか対応できない．本方式でも置換・1 次元スライドの両方式と同様に 2 辺に予備ノードを確保することで，**図 8** のように 2 ノードまでの故障に対応することが可能となる．



(a) 1 ノード故障時の実行継続例　(b) 1 ノード故障後の通信

図 7 2 次元スライド方式による実行継続



(a) 2 ノード故障時の実行継続例　(b) 2 ノード故障後の通信

図 8 予備ノードを 2 辺に確保した場合の 2 次元スライド方式

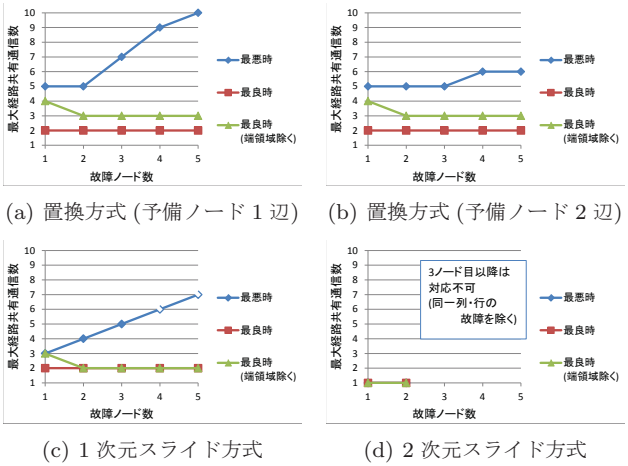
3.4 各方式の比較

故障発生後に実行を継続した際の性能を確認するために，各方式による実行継続の模擬プログラムを作成しシミュレーションを行った．このシミュレーションプログラムでは，ジョブ実行ノードとして 12x12 ノードを確保し，予備ノード 1 辺の場合には最上行を，2 辺の場合には最上行と最右列を予備ノード群として確保し，2 次元 5 点ステンシル通信を実行する．そして計算に用いているノードが故障した場合，提案した各方式により実行継続を行った際の最大経路共有通信数と，実行ノードを移動するプロセス数がどのように変化するか，2 次元スライド方式では対応可能な 2 ノード故障まで，それ以外では 5 ノード故障までの全ての故障パターンについて調査した．

3.4.1 経路を共有する通信数

予備ノードを 1 辺に確保した置換方式と，予備ノードを 2 辺に確保した置換方式・1 次元スライド方式・2 次元スライド方式の 4 種類について，故障台数と最大経路共有通信数の関係を**図 9** に示す．故障ノードの位置により性能が変化するため，最悪・最良それぞれの場合について示している．なお，最良の場合については，通信数の少ない端領域を担当するノードの故障を除いたものも合わせて示す．また，最悪性能の凡例のうち白抜きで示された部分は，故障ノードの位置により対応不可能な組み合わせが存在した場合を表している．

Vol.2014-ARC-213 No.21
Vol.2014-HPC-147 No.21
2014/12/10



(a) 置換方式 (予備ノード 1 辺)　(b) 置換方式 (予備ノード 2 辺)

(c) 1 次元スライド方式　(d) 2 次元スライド方式

図 9　各方式による経路共有通信数

最も通信性能の良い方式は 2 次元スライド方式である．2 次元スライド方式では 2 辺に予備ノードを確保した場合でも 2 ノードの故障までしか対応できないが，対応可能な台数であれば通信衝突の可能性が発生することはなく，性能の維持が可能である．置換方式と 1 次元スライド方式を比較すると，最良時に関しては 1 次元スライド方式が常に高性能を示している．しかし最悪時を見ると，故障ノード数が 3 以下であれば 1 次元スライド方式が有利であるが，4 ノード故障時で両方式が同性能となり，5 ノード故障時では逆転している．また，1 次元スライド方式では 4 ノード以上の故障ノードが発生した場合，ノードの位置によっては対応できない場合も存在する．

3.4.2　移動プロセス数

各方式による実行継続で移動するプロセス数を表 1 に示す．表において N_w 及び N_h は，それぞれ計算に用いているノードの幅と高さを示し，n_{fault} は故障ノードの台数を示している．また，1 次元スライド方式及び 2 次元スライド方式において *or* となっている部分は，スライドする方向によって決定する．

表 1　各方式により移動するプロセス数

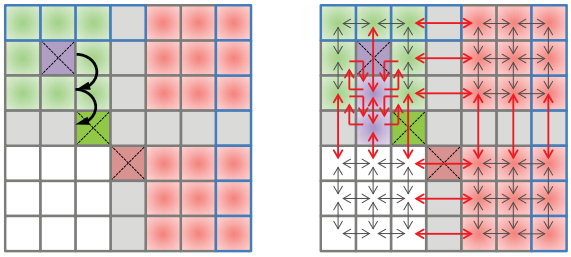
	置換方式	1 次元スライド方式	2 次元スライド方式
最悪時	n_{fault}	$(N_w or N_h) \times n_{fault}$	$N_w \times N_h \times n_{fault}$
最良時	n_{fault}	n_{fault}	$(N_w or N_h) \times n_{fault}$

置換方式による実行継続では，移動するプロセスの数は故障ノード数と同一であり常に一定である．一方 1 次元スライド方式と 2 次元スライド方式では，故障ノードの位置によりスライドするプロセス数が変化し，その数は置換方式と比較して非常に多くなる．このため，プロセス内のメモリに復帰に必要なデータを保持しているようなアプリケーションでは，置換方式が $T_{Restart\_spare}$ を最も短縮可能となる．

3.5　ハイブリッド方式

通信性能の観点では最高性能の 2 次元スライド方式は，通信衝突の可能性が生じない反面，2 ノードの故障までしか対応不可能である．そこで，2 ノードの故障までは通信性能を低下させず，かつ 3 ノード以上の故障に耐えるための手法として，これまでの方式を組み合わせたハイブリッド方式が考えられる．

2 次元スライド方式では，故障したノードと同列 (または同行) の故障していないノードも故障したとみなしてスライドする．そのため，2 ノード故障後もまだ正常なノードに空きが存在する．この空きノードを予備ノードとして扱い，3 ノード目の故障以降は，1 次元スライド方式を用いた対応を行う．**図 10(a)** は，図 8 で示した 2 次元スライド方式による実行継続の後に，更にもう 1 ノードに故障が発生した場合のハイブリッド方式による対応例である．経路を共有する最大通信数は，3 ノード故障時では**図 10(b)** のように 1 次元スライド方式の 1 ノード故障時と同じ 3 となり，以降 n ノード故障時には 1 次元スライド方式の $(n-2)$ ノード故障時と同等となる．



(a) 3 ノード目の故障発生時の対応　(b) 実行継続後の通信

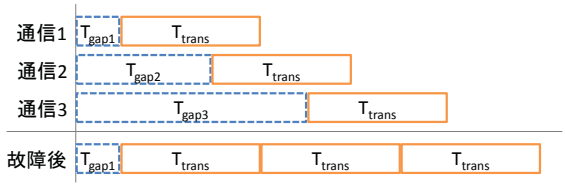
図 10　2 次元スライド方式と 1 次元スライド方式を組み合わせたハイブリッド方式

また，1 次元スライド方式も確実な対応可能台数が 3 ノードまでに限られるため，6 ノード故障時からは置換方式を用いることで，予備ノードと同数の故障ノード数までの実行継続を保証可能である．

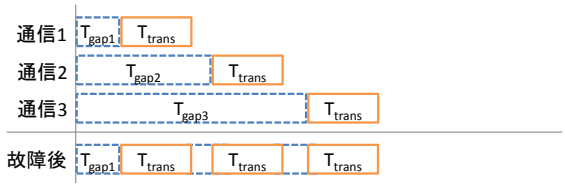
3.6　通信タイミングのずれと通信データサイズによる影響

ここまで，通信時間は経路を共有する最大の通信数に比例して増大するという想定の基に議論を進めてきた．この想定は通信時間と実際のデータ転送時間が同一であり，それぞれの通信が同時刻に到達して完全に衝突するという前提によるものである．しかし実際には，通信発行時のソフトウェア・ハードウェア処理に要する時間や，ネットワークレイテンシ等の影響があるために，この前提のような理想的な環境は存在しない．

図 11 は，あるタイミングで同時に発行された 3 通信の通信時間の例を示したものであり，T_{trans} は実際にデータを転送するために必要な転送時間を，T_{gap} は通信のずれを示している．**図 11(a)** と**図 11(b)** の違いはデータサイズであり，データサイズの大きい図 11(a) の方が T_{trans} が



(a) データサイズ大



(b) データサイズ小

図 11 通信データサイズの差による経路共有時の通信時間

長くなっている．それぞれの例において，故障発生前は 3 通信が全て別の経路を通り，故障発生後はこれらの 3 通信が同一の経路を通過するものとする．故障発生前においては，全通信が完了するまでの時間は最も長い通信時間である $T_{gap3} + T_{trans}$ となる．しかし，故障発生後の通信時間は双方で大きく異なってくる．図 11(a) では，通信の衝突が発生するために通信時間が増加し，通信完了までには $T_{gap1} + 3 \times T_{trans}$ を要する．一方で図 11(b) では，転送時間が通信のずれの中に吸収されており，同一経路を通過した場合でも通信に衝突が発生せず，通信完了までに要する時間は故障前と変わらず $T_{gap3} + T_{trans}$ である．

つまり，通信データサイズが大きく通信のずれによる影響が小さい場合，これまでの前提とほぼ同様に通信時間は経路を共有する最大通信数に比例するが，通信データサイズが小さく，ずれの影響が大きくなれば，経路を共有する最大通信数の性能への影響は減少する．

3.7 実アプリケーションへの適用

実アプリケーションへと予備ノード利用手法を適用する際には，どの方式が故障後の実行時間を最も短縮可能であるか，式 4 に基づいて考慮する必要がある．

アプリケーションが故障前へ復帰するための情報をメモリに保持していない場合は，チェックポイントからの復帰時間 $T_{Restart\_spare}$ はどの方式を用いても同一であるため，通信性能の良いハイブリッド方式を採用することが有効だと考えられる．一方，メモリからのロールバックが可能な場合は，置換方式を利用することで $T_{Restart\_spare}$ の短縮が期待できる．また，実アプリケーションでは，計算処理のばらつきによる通信発行のずれが加わり，経路共有による通信時間への影響は低下すると考えられ，置換方式が有効な場合も多いと想定される．しかし，通信データサイズが大きい場合は，経路共有による通信時間の増加が大きくなるため，通信時間と復帰時間のバランスを考慮して方式を採用する必要がある．

4. 関連研究

Laguna らによる [8] では，実アプリケーションに対して ULFM を利用した故障対策の実装を行い，評価を行っている．この論文では，実アプリケーションとして分子動力学計算を行うアプリケーション ddcMD を用いている．ddcMD には，プロセス数の変更に伴う計算領域の変更への対応や，各プロセス間でのロードバランス処理といった特徴があるため，故障ノードを除去する手法での実行継続を採用している．その一方で，多くの HPC アプリケーションはこのような特徴を持っておらず，その実装は難しいために，故障ノードの除去による対応は困難であると述べられている．そこで，我々が本論文で述べた予備ノード利用手法のような，ノード数を維持した実行継続手法の必要性が書かれているが，提言のみであり復帰後の性能などの詳細な検討はなされていない．

5. まとめ

本論文では，エクサスケールでの耐故障性の実現に向け，故障発生後の実行継続手法についての検討を進めた．現在広く利用されているジョブの再投入と比較して，予備ノード利用手法を用いた実行継続による実行時間が短くなるためには，予備ノード利用によって発生する性能低下の抑制と，状態復元に要する時間の削減の 2 点が重要である．

そこで，どのように予備ノードを利用することが効果的であるか，3 種類の利用方式について検討した．メモリ内に状態復元のためのデータを保持しないアプリケーションでは，故障後も通信性能の低下がない 2 次元スライド方式や，それを拡張したハイブリッド方式が有効である．一方，復元に必要なデータをメモリに保持している場合，データを保持したプロセスが多く生存する置換方式の優位性が上昇する．

今後，複数の実アプリケーションへと各方式を適応し，その有効性を確認する．また，集団通信などのステンシル通信パターン以外のアプリケーションに対する効果的な実行継続手法についての検討も進めていく予定である．

謝辞 本研究は，科学技術振興機構 (JST) 戦略的創造研究推進事業 (CREST) における研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」研究課題「メニーコア混在型並列計算機用基盤ソフトウェア」によるものである．

参考文献

[1] Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B. and Snir, M.: Toward Exascale Resilience, *Int. J. High Perform. Comput. Appl.*, Vol. 23, No. 4, pp. 374–388 (2009).

[2] Cappello, F., Geist, A., Gropp, B., Kale, S., Kramer, B. and Snir, M.: Toward Exascale Resilience:2014

Update, *Supercomputing frontiers and innovations*, Vol. 1, No. 1, pp. 1–28 (online), available from ⟨http://superfri.org/superfri/article/view/14/7⟩ (2014).

[3] Fault Tolerance Research in the Innovative Computing Lab at the University of Tennessee: User Level Failure Mitigation — ICL Fault Tolerance, The University of Tennessee (online), available from ⟨http://fault-tolerance.org/ulfm/⟩ (accessed 2014-11-7).

[4] Bland, W., Bouteiller, A., Herault, T., Hursey, J., Bosilca, G. and Dongarra, J.: An evaluation of User-Level Failure Mitigation support in MPI, *Computing*, Vol. 95, No. 12, pp. 1171–1184 (online), DOI: 10.1007/s00607-013-0331-3 (2013).

[5] MPI Forum: MPI Forum's Fault Tolerance Working Group, MPI Forum (online), available from ⟨https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/FaultToleranceWikiPage⟩ (accessed 2014-11-7).

[6] 吉永一美，亀山豊久， 堀敦史， 石川裕：エクサスケールでの耐故障性実現に向けた代替ノード配置による通信性能の評価，情報処理学会研究報告. [ハイパフォーマンスコンピューティング]， Vol. 2014, No. 16, pp. 1–6 (2014).

[7] 吉永一美，亀山豊久，畑中正行， 堀敦史， 石川裕：代替ノード利用手法による耐故障性実現に向けた通信性能の評価と検討，情報処理学会研究報告. [ハイパフォーマンスコンピューティング]， Vol. 2014, No. 6, pp. 1–8 (2014).

[8] Laguna, I., Richards, D. F., Gamblin, T., Schulz, M. and de Supinski, B. R.: Evaluating User-Level Fault Tolerance for MPI Applications, *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, ACM, pp. 57–62 (2014).