計算グリッド向けフォールトトレラントシステム Eagle の提案と初期評価

服部晃和[†] 薬師寺健太[†] 横田隆史[†]
大津金光[†] 古川文人[††] 馬場敬信[†]

近年，ネットワークで結ばれた異なるドメインに属する計算資源を結合し，仮想的な並列計算機を動的に構築するためのインフラストラクチャである計算グリッドが高い関心を集めている．一般に，大規模な計算資源を利用し長時間にわたって計算を行う場合，システム内に障害が発生し，計算を正常に完了できなくなる可能性が高くなる．このため，計算グリッドを用いて構築されるシステムの実用化のためには，構築されるシステムを高信頼化する技術が必要である．本稿では，ドメイン単位のプロセス譲渡機能を備えた計算グリッド向けフォールトトレラントシステム Eagle を提案する．Eagle では，複数プロセスの同時障害からのリカバリに加えて，ドメイン単位のプロセスの譲渡が可能である．我々は，MPI アプリケーション向けの Eagle の実装である MPICH-EG を開発している．グリッド環境との親和性を高めるため，MPICH-EG は主要なグリッドミドルウェアである Globus 上で実装を進めている．本稿では，マイクロベンチマークを用いて MPICH-EG の基本通信性能を評価し，NAS Parallel Benchmarks（NPB）の実行オーバヘッドを評価する．また，いくつかのチェックポインティング手法をチェックポインタ ckpt をベースとして実装し，NPB の実行オーバヘッドを評価する．これらの評価結果より，MPICH-EG の基本特性を明らかにするとともに，MPICH-EG に有効なチェックポインティング手法を検討する．

A Proposal and Preliminary Evaluation of a Novel Fault-tolerant System Named Eagle for Computational Grids

Akikazu Hattori,[†] Kenta Yakusiji,[†] Takashi Yokota,[†]
Kanemitsu Ootsu,[†] Fumihito Furukawa[†] and Takanobu Baba[†]

Computational grid technologies are greatly expected as an infrastructure to dynamically build virtual parallel computers by collecting computational resources across multiple domains. Generally, a long-running application on a huge parallel computer has a certain risk due to the increase of failure rate of the system. Therefore, fault-tolerant technologies are required to build a reliable computational grid system in practice. In this paper, we propose a novel fault-tolerant system Eagle for computational grids. It enables all processes in a domain to migrate to another domain. Furthermore, Eagle can tolerate simultaneous process failures. We are developing a fault-tolerant MPI named MPICH-EG as an implementation of Eagle. Implementation of MPICH-EG is in progress on Globus, a major grid middleware, to increase affinity with grid environment. We evaluate both a basic communication performance and practical overheads by using microbenchmarks and NAS Parallel Benchmarks (NPB). We also discuss checkpointing methods and evaluate their overheads by using a checkpointer called ckpt.

1. はじめに

近年，大規模化・複雑化の一途をたどる計算問題に対応するためのインフラストラクチャとして計算グリッド[1)]が注目されている．計算グリッドを利用することで，異なるドメインに属する計算資源を結合し，仮想的な並列計算システムを動的に形成することが可能となる．これにより，単一ドメイン内の計算資源だけでは取り扱えなかった大規模な計算問題を取り扱うことが可能となる．しかし，大規模な計算資源を利用し長時間にわたって計算を行う場合，一般に，システム内に発生した障害のために計算を正常に完了できなくなる可能性が高くなる．そのため，大規模な並列計算を

† 宇都宮大学工学部情報工学科
Department of Information Science, Faculty of Engineering, Utsunomiya University
†† 宇都宮大学ベンチャービジネスラボラトリ
Venture Business Laboratory, Utsunomiya University

実現するためには，動的に形成された並列計算システムを高信頼化する技術が必要である．

本稿では，複数の異なるドメインに分散した計算資源を利用し，並列処理を行う計算グリッドを対象としたフォールトトレラントシステム Eagle を提案する．Eagle では，プロセス間で送受信されるメッセージを中継ノードを介して記録・配送する．また，計算プロセスの全実行情報を含むチェックポイントデータを定期的に作成し障害に備える．システム内に障害が発生した場合は，チェックポイントデータとメッセージの記録を用いて自律的にリカバリを行う．加えて，計算プロセスを他の計算資源に譲渡するプロセス譲渡とメッセージ通信の制御を組み合わせ，ドメイン単位のプロセス譲渡を実現する．ドメイン単位のプロセス譲渡を実現するシステムはいまだ存在せず，Eagle は新規なものである．

本研究では，MPI アプリケーションを対象とした Eagle の実装である MPICH-EG を開発している．MPICH-EG は，グリッド向けの通信ライブラリである MPICH-G2[2] をベースとして開発されている．また，MPICH-EG の構成要素を Globus[3] の API を用いて実装することで，Globus が提供する資源管理，データ管理等の各機能を活用し，ヘテロな環境下で MPICH-EG を動作させることを目指す．これにより，MPICH-EG とグリッド環境の親和性を高め，MPICH-EG がグリッド環境での実用に耐えるものとなる．本稿では，MPICH-EG の基本通信性能をマイクロベンチマークにより実機評価するとともに，実アプリケーションに近い NPB 2.3（NAS Parallel Benchmarks）[4] の実行オーバヘッドを実機評価する．また，いくつかのチェックポインティング手法をチェックポインタ ckpt[5] 上に実装し，NPB 2.3 の実行オーバヘッドを実機評価する．これにより，MPICH-EG の基本特性を明らかにするとともに，MPICH-EG に有効なチェックポインティング手法を検討する．

2. フォールトトレランス手法

現在，メッセージ通信を行うシステムを対象としたフォールトトレラントシステムに関する多数の研究が理論，実装の両面からなされている．それらの多くはシステム内に障害が発生した場合，システムを障害が発生する以前の状態に巻き戻すロールバックリカバリ[6] を対象としている．ロールバックリカバリにはチェックポイントベースの手法と，ログベースの手法がある．チェックポイントベースの手法は，システムを構成する各プロセスの実行イメージであるチェックポイントデータのみを用いる．ログベースの手法は，チェックポイントデータと通信されたメッセージの記録の両者を用いる．ログベースの手法はさらに Pessimistic logging，Optimistic logging，Causal logging に分類される．

チェックポイントベースの手法は，ログベースの手法と異なり，プロセス間で通信されるメッセージの保存が不要であり，そのためのオーバヘッドが発生しない．そのため各プロセスが生成するチェックポイントデータのサイズが信頼できる保存先である Stable storage に極端な入出力負荷をかけない程度であれば，実行時間のオーバヘッドをログベースの手法よりも低く抑えることができる．しかし，プロセスに障害が発生しリカバリを行う場合，全プロセスがロールバックの対象となってしまう．これに対し，ログベースの手法ではロールバックの対象となるプロセスを限定できるので，リカバリ時間をチェックポイントベースの手法より短くすることができる．

2.1 チェックポイントベースの手法

MPI を対象とした並列チェックポインタの実装に CoCheck[7]，Starfish[8] 等がある．Cocheck ではチェックポインティングを行う際，各プロセスが，送信中のメッセージの送信完了に続いて RM（Ready Message）と呼ばれる特別なメッセージを全プロセスに送信する．チェックポインティングは全プロセスから RM を受信した後に行われる．これにより，通信路にメッセージが流れている状態のままチェックポインティングを行わないことを保証している．しかし，この方法では RM の送受信によるプロセス間の同期のコストが大きくなりすぎてしまうため，チェックポインティングのオーバヘッドが大きく実用には至っていない．Starfish ではユーザにチェックポインティングのための API を提供している．チェックポインティングの間隔や，並列計算環境の変化に応じたシステムの挙動をユーザが定義できるため，プログラムに適したフォールトトレラントシステムを形成できる．しかし，並列プログラムのコードを変更しなければならないため，ユーザにとっては大きな負担になる．

チェックポイントベースのロールバックリカバリ機能を備えた MPI の実装に MPICH-GF[9] がある．MPICH-GF はグリッドミドルウェアの Globus 上に実装されている．MPICH-GF では Central Manager が Local Manager にチェックポインティングを指示し，Local Manager がチェックポインティングを計算プロセスに指示する．チェックポインティングの指示を受けた計算プロセスは CoCheck のメカニズムをバリ

ア同期により実現し，チェックポインティングを行う．

2.2 Pessimistic logging

Pessimistic logging はプロセスどうしが通信を行う際，取り交わされるメッセージを計算に反映させる前に Stable storage に格納することを保証するプロトコルである．Pessimistic logging では，各計算プロセスが他のプロセスから独立してチェックポインティングを行うことができる．また，複数のプロセスに同時に障害が発生しても，障害が発生したプロセスは他のプロセスと独立にロールバックリカバリを行うことができる．Pessimistic logging の実装として MPICH-V [10] がある．MPICH-V では，メッセージを CM（Channel Memory）と呼ばれる信頼できる中継ノードを介して配送することで Pessimistic logging を実装している．

正常実行時のメッセージ保存によるオーバヘッドを削減するため，メッセージの記録を送信側で保存し，受信側では送信プロセス，受信のタイミング等メッセージの因果情報のみを保存しておく Sender based message logging [6] と呼ばれるプロトコルが考案されている．Sender based message logging では正常実行時のメッセージ保存によるオーバヘッドは削減できるが，リカバリ時には全プロセスにメッセージの記録を再送するよう要求する必要があり，リカバリに要する時間が Pessimistic logging に比べて増加する．また，同時に 1 つのプロセス障害しか許容できない．Sender based message logging の実装として MPICH-V2 [11] がある．MPICH-V2 では信頼できるプロセスがメッセージの因果情報を管理することにより，複数の同時プロセス障害を許容することができる．

2.3 Optimistic logging

Optimistic logging は，プロセス間で取り交わされるメッセージの記録が Stable storage に格納されていなくても，メッセージを計算に反映してしまうログベースのプロトコルである．メッセージの保存による正常実行時のオーバヘッドを削減することができるが，1 つのプロセス障害が他の正常なプロセスのロールバックを招くことがある．

2.4 Causal logging

Causal logging では，メッセージを送信する際，そのメッセージが生成されるまでの送受信系列である追跡情報をメッセージに付加して送るログベースのプロトコルである．メッセージ本体と，追跡情報を f 個の異なるプロセスのメモリ空間に複製する．これにより同時に $f-1$ 個の同時プロセス障害をリカバリすることが可能となる．しかし，メッセージ本体に追跡情報を付加して送るうえ，宛先のプロセス以外にもメッセージを送信するため，ネットワークにかかる負荷は増大する．また，Causal logging はリカバリと，不要なメッセージの記録やチェックポイントデータを削除するガーベイジコレクション処理を複雑にする．Causal logging の実装としては Manetho [12] があげられる．

3. Eagle

3.1 Eagle の設計理念

グリッド上で複数のドメインに分散した計算資源を利用し，並列処理システムを構築する場合，構築されるシステムの規模が従来より大きくなることが予想される．一般に，システムが大規模化すると，システム内の複数の計算資源に同時に障害が発生する可能性が高くなる．また，メンテナンス作業等により，あるドメインがサービスを一時中断せざるをえない状況が起こりうる．そのため，グリッド上で構築される並列処理システムは，以下の 2 点の特性を備える必要があると考える．

(1) グリッド環境の制約条件の中で複数の同時プロセス障害をリカバリ可能であること．
(2) 計算資源を提供しているドメインが，実行中のプロセスを他のドメインに譲渡し，計算から脱退することが可能であること．

大規模なシステムをフォールトトレランスの対象とする場合，1 つのプロセス障害が他の正常なプロセスに与える影響を最小化するため，チェックポイントベースの手法よりもログベースの手法の方が採用するリカバリプロトコルとして適していると考える．また，計算資源どうしが WAN で結ばれていることがありうるため，チェックポイントデータやメッセージの記録の送信を通信レイテンシの大きい WAN を介さずに行えるリカバリプロトコルを採用することが，リカバリ性能を向上させるうえで重要と考える．

以上の条件に鑑み，プロセス障害が他のプロセスのロールバックを招かず，チェックポイントデータとメッセージの記録の送信元がドメイン内の Stable storage に限定される Pessimistic logging をリカバリプロトコルとして採用する．

Eagle では，メッセージの中継ノードを用いた通信機構を用いて Pessimistic logging を実装する．計算プロセスが実行されている実際のノードの位置を中継ノードが管理することにより，リカバリやドメイン単位のプロセス譲渡により変化する計算プロセスの位置を他の計算プロセスから隠蔽することが可能となる．Eagle ではドメイン単位のプロセス譲渡を可能とする

ため，ドメイン間でのメッセージ通信を制御する中継ノードを Pessimistic logging の実装用とは別に設置する．

グリッド上では，動的なシステム構築が求められている．また，プロセス障害のリカバリ，プロセス譲渡はシステムの動的な変化といえる．こうしたシステムの動的な変化を実現，許容するためにはシステムがグリッド環境のヘテロ性を許容することが必要である．ヘテロ性を許容することができれば，リカバリやプロセス譲渡を計算資源のアーキテクチャに依存せずに行うことが可能となるため，システムに柔軟性を持たせることが可能となる．グリッド環境のヘテロ性の許容のため，Eagle を主要なグリッドミドルウェアである Globus 上に実装する．

3.2 Eagle の概要

図 1 に Eagle のアーキテクチャを示す．Eagle を構成するためのプロセスは図中において楕円で示されており，その中にプロセスが担当する機能を示す名称が書かれている．ユーザのプログラムは，計算プロセス CP（Computation Process）として実行される．図中の他のプロセスは，Eagle でフォールトトレランスを実現するために用いられる．以下，システム内部の動作をメッセージ通信系，チェックポインティング系，プロセス監視系の各機能に分けて説明する．

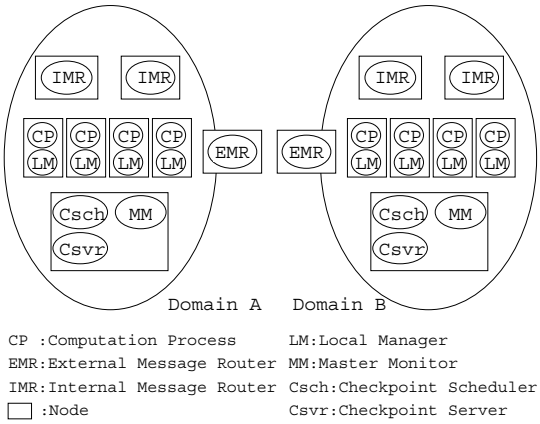

図 1 Eagle のアーキテクチャ
Fig. 1 The architecture of Eagle.

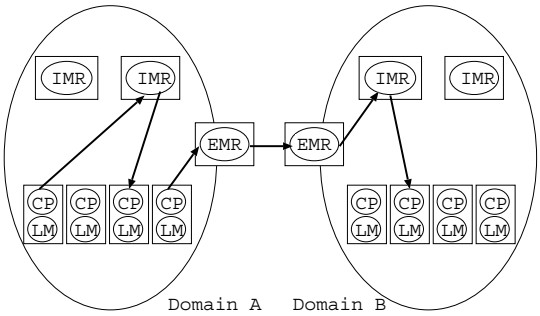

図 2 Eagle のメッセージ送受信の概念図
Fig. 2 Message communication in Eagle.

メッセージ通信系

図 2 に Eagle のメッセージ送受信の概念図を示す．計算プロセス CP 間の通信は，メッセージの記録と配送を行う中継プロセス IMR（Internal Message Router）を介して行う．これにより Pessimistic logging を実現するとともに，CP の物理的な位置を相互に隠蔽することが可能となる．IMR は，CP が実行されるノードとは別のノード上に設置する．

また，ドメイン単位のプロセス譲渡を実現するため，ドメイン間で通信されるメッセージを制御するためのプロセス EMR（External Message Router）を設置する．EMR は，IMR が実行されているノードとは別のノード上に設置する．

チェックポインティング系

CP の障害に備えて定期的にチェックポイントデータを作成し，保存しておく必要がある．このため，チェックポイントデータを受信し管理するためのプロセス Csvr（Checkpoint Server）を CP とは別のノード上に設置する．

チェックポインティングの間隔はリカバリ性能を大きく左右するので，アプリケーションに適した間隔を保つ必要がある．適切なチェックポインティングの間隔を保つためには，CP とは別のプロセスによるチェックポインティングのタイミング管理が必要である．そのため，チェックポインティングをスケジューリングするプロセス Csch（Checkpoint Scheduler）を，Csvr が実行されているノードと同一のノード上に設置する．

プロセス監視系

計算プロセス CP の生存を監視し，また，チェックポイントスケジューラ Csch の指示に従いチェックポインティング指示を CP に伝えるプロセスが必要であるため，CP および Csch の仲介をするプロセス LM（Local Manager）を CP が実行されているノード上に設置する．

また，LM を監視することで CP および CP が実行されているノードに発生した障害を検出するためのプロセスが必要であるため，Csch，Csvr が実行されているノード上に障害検出を行うプロセス MM（Master Monitor）を設置する．

以下に，Eagle のコンポーネントとその役割をまと

める．

- CP（Computation Process）：計算プロセス．ユーザコードを実行するフォールトトレランスの対象となるプロセスであり，チェックポインティング機能を有する．
- IMR（Internal Message Router）：内部中継プロセス．ドメイン内のメッセージの保存と配送を行うプロセスである．
- EMR（External Message Router）：外部中継プロセス．ドメイン間のメッセージ転送を制御するプロセスである．
- Csvr（Checkpoint Server）：チェックポイントサーバ．チェックポイントデータを CP から受け取り，管理するプロセスである．
- LM（Local Manager）：ローカルマネージャ．CP が実行される各ノードで実行され，CP の監視とチェックポインティングの指示を CP に対して行うプロセスである．
- Csch（Checkpoint Scheduler）：チェックポイントスケジューラ．Csvr が実行されるノードで実行され，チェックポインティングの指示を LM に対して行うプロセスである．
- MM（Master Monitor）：マスターモニタ．Csch，Csvr が実行されるノードで実行され，CP および CP が実行されているノードの障害を検出するためのプロセスである．

計算資源に発生したプロセス障害のリカバリを当該ドメイン内で実現するとともに，ドメイン単位のプロセス譲渡を実現することが Eagle の目的である．Eagle では，ネットワークの断絶や計算ノードのメモリが不正な値を返すことによるプロセスの不正動作を許容することは対象としていない．また，Eagle では CP ならびに CP を実行しているノード以外には障害が発生しないものとする．

3.3 メッセージの記録

Eagle では計算プロセス CP どうしが直接通信をすることはなく，必ず内部中継プロセス IMR を経由して通信を行う．各 CP は，ドメイン内の IMR のいずれかと対応関係を持つ．

送信先の CP が同一ドメイン内に属している場合，宛先の CP に対応する IMR に対してメッセージを送信する．メッセージを受け取った IMR は，受け取ったメッセージを保存した後，宛先の CP に受け取ったメッセージを送信する．

宛先の CP が他ドメインに属している場合，同一ドメイン内の外部中継プロセス EMR，宛先の CP が属するドメイン内の EMR，宛先の CP に対応する IMR の順番でメッセージを送信する．IMR がメッセージを受け取った後の動きはドメイン内転送の場合と同様，メッセージを保存した後，宛先の CP に受け取ったメッセージを送信する．

IMR は，対応する CP がチェックポイントデータを作成し，チェックポイントサーバ Csvr へチェックポイントデータを転送した後に，チェックポイントスケジューラ Csch の指示によりチェックポインティング以前に受信したメッセージの記録を破棄する．

3.4 チェックポインティング

チェックポインティングを行う際，ネットワーク，OS のバッファに存在するメッセージには特別な注意を払う必要がある．ネットワーク，OS のバッファにメッセージが存在する状態で作成されたチェックポイントデータを用い，チェックポイントデータを作成したホストと異なるホスト上でプロセスをリカバリすると，ネットワーク，OS のバッファに存在するメッセージが失われてしまうからである．この問題を解決する最も簡単な方法はネットワーク，OS のバッファにメッセージが存在する状態でチェックポインティングを行わないことである．

チェックポインティング時の Eagle の動作を**図 3** に示す．実線で表した矢印は CP 間で通信されるメッセージであり，破線で表した矢印はフォールトトレランス用のメッセージである．IMR，CP 等の名称から伸びている横線は時間軸を表し，fork() と記載されている点は CP が UNIX システムコールの fork() を用いて子プロセスを生成したことを表す．図 3 に示した番号に合わせ，チェックポインティングの手順を以下に示す．以下ではチェックポインティングを行う CP を単に CP と記述する．このアルゴリズムが動作するためには，TCP 等によりメッセージ到着順を保障す

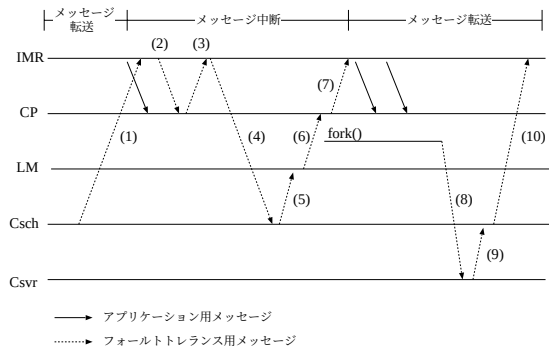

図 3 チェックポインティング時の動作
Fig. 3 Timing flow at checkpointing.

る必要がある．

(1) Csch は IMR に対し，CP へのメッセージ送信の一時中断を指示する．
(2) IMR は CP 宛に送信中のメッセージを送信し終えた後，メッセージ中断完了を表す特別なメッセージを CP へ送信する．
(3) 中断完了メッセージを受け取った CP は IMR に応答メッセージを送信する．
(4) CP からの応答メッセージを受け取った IMR は Csch に応答メッセージを送信する．
(5) Csch は LM に対しチェックポインティングを指示する．
(6) LM は CP に対しチェックポインティングを指示する．
(7) fork システムコールを用いプロセスを複製しチェックポインティングを開始する．親プロセスは IMR にメッセージの送信再開を依頼し，子プロセスはチェックポイントデータの作成を開始する．
(8) 子プロセスは作成したチェックポイントデータを Csvr に送信した後終了する．
(9) Csvr はチェックポイントデータの受信が完了した後 Csch にチェックポインティング完了を通知する．
(10) Csch は IMR に対し，メッセージの中断完了を表すメッセージを送信する以前に送信したメッセージの記録を破棄するよう指示する．

3.5 リカバリ

計算プロセス CP，あるいは CP が実行されているノードに障害が発生した場合，マスターモニタ MM が障害を検出し，ドメイン内のスケジューラに報告する．ドメイン内のスケジューラは代替ノードを用意し，CP，LM をその代替ノード上に生成する（代替 CP，LM と呼ぶ）．その後，障害が発生した CP に対応する Csvr，IMR がそれぞれ保持しているチェックポイントデータとメッセージの記録を代替 CP に送信する．代替 CP は，受信したチェックポイントデータによりチェックポインティング時の状態を復旧する．チェックポインティング時以降，障害が発生までの間に受信したメッセージは，IMR から受信したメッセージの記録により再現される．リカバリにともない，代替 CP との通信に必要な IP アドレス，ポート番号等の情報が変わるが，この変更は障害が発生したプロセスに対応する IMR にのみ反映すればよい．この機構により，CP の障害が影響する範囲を最小化できる．加えて，ある CP の障害は他の CP とは完全に透過となる．

3.6 譲　　渡

プロセスの譲渡には，ノードが単独で脱退する場合に行われるものと，ドメイン全体が脱退する場合に行われるものとがある．ある CP を実行しているノードが計算から脱退する場合，ドメイン内のスケジューラが代替ノードを用意し，リカバリと同一の手法を用いて譲渡を完了する．

あるドメインが計算から脱退する場合，脱退の対象となるドメインに対応する外部中継プロセス EMR が他のドメインの EMR に対し，一時的なメッセージの停止要求であるブロックリクエストを送信する．ブロックリクエストを受け取った EMR は対象のドメイン宛のメッセージを一時ブロックする．ブロックの完了後，ブロックリクエストの送信者に対してブロック完了メッセージを送信する．脱退の対象となるドメインでは，ブロック完了メッセージを待っている間，各 CP がチェックポイントデータを作成しチェックポイントサーバ Csvr に送信しておく．ブロックリクエストを送信した EMR はドメイン内の Csvr にチェックポイントデータを譲渡先に送信するよう指示する．チェックポイントデータの送信完了により，対象ドメインの計算からの脱退処理が完了する．

譲渡は以下の手順で行う．以下の処理は TCP 等によりメッセージ順序を保障する必要がある．

Phase 1. メッセージ送信の停止依頼

プロセスの譲渡を行うドメイン（脱退ドメイン）に属する EMR は，他のドメイン（脱退対象外ドメイン）に属する全 EMR に対してメッセージ送信の停止を依頼する．

Phase 2. メッセージ送信の完遂と停止

メッセージの停止依頼を受け取った EMR は，他ドメイン宛に転送途中のメッセージをすべて送出した後に，最初の停止依頼を送った脱退ドメインの EMR に対して応答メッセージ（ACK）を返す．その後，脱退対象外ドメインの EMR は，ドメイン間の通信が再開するまで脱退ドメイン宛のメッセージの転送を行わず，自ノードに保存しておく．

脱退ドメインの EMR が，脱退対象外ドメインのすべての EMR からの ACK を受け取れば，それ以降，脱退ドメインへのメッセージが送られてこないことが保証される．

脱退ドメインの EMR は，通常の動作どおりに，他ドメインから送られてきたメッセージを適切な IMR に転送する．IMR はメッセージを記録してから目的の CP に配送する通常の動作を行う．

Phase 3. チェックポインティング
次に，脱退ドメインに属する Csch が，自ドメイン内の全 CP に対してチェックポインティングの開始を指示する．ドメイン内の各構成要素は，通常のチェックポインティングと同様に，3.4 節で述べた方法でチェックポインティングを行う．これにより，ドメイン内で実行されている全 CP の状態が安全に保存される．チェックポインティング終了後，CP は終了する．

Phase 4. プロセス状態の移動
全 CP でのチェックポインティングが終了したら，Csvr に保存されているチェックポイントデータと IMR に保存されているメッセージ記録を，譲渡先ドメインに転送する．転送は，脱退ドメインおよび譲渡先ドメインの EMR の間で行われる．
譲渡先ドメインの EMR は，受け取ったチェックポイントデータならびにメッセージ記録を適切な配送先（Csvr，IMR）に転送する．これにより脱退ドメインのプロセス状態がすべて譲渡先ドメインに転送される．

Phase 5. プロセスの再開
前ステップの処理が完了したら，譲渡先ドメインの各プロセスの動作を再開させる．CP の再開は，通常の障害回復と同じ方法で行えばよい．こうしてドメイン内の全プロセスが正しく再開できたら，EMR によるドメイン間通信を再開し，ドメイン間でのプロセス譲渡が完了する．

コストの見積り

上述のドメイン間プロセス譲渡に要する時間を見積もる．Phase 1，2 はドメイン間のメッセージ転送を停止するための手順であり，これにかかる時間は，最初に脱退ドメインの EMR からメッセージ送信の停止要求が出されたあと，脱退対象外ドメインの EMR で配送途中にあったメッセージをフラッシュし，応答メッセージ（ACK）を脱退ドメインの EMR に返すまでとなる．

Phase 3 では脱退ドメイン内でいっせいにチェックポインティングが行われる．チェックポインティングは各 Csvr を単位にドメイン内で並行して行われる．このため，Phase 3 にかかる時間は，各 Csvr が担当する CP 数によってほぼ決まる．

Phase 4 では脱退ドメイン内のすべてのチェックポイントデータとメッセージ記録（譲渡データ）を譲渡先ドメインに転送する必要がある．この転送は，上記の譲渡データの総量と，脱退ドメイン・譲渡先ドメインの間のバンド幅によって決まる．

表 1 PC クラスタ Swallow の仕様
Table 1 Specification of PC cluster Swallow.

CPU	Intel Celeron 2.0 [GHz] × 1
Memory	512 [MB]
Network	100Base-T switching hub
OS	Debian Linux kernel 2.4.18
Grid middleware	Globus 2.4
MPI Framework	MPICH 1.2.5

以上から，ドメイン間のプロセス譲渡にかかる時間は Phase 4 が支配的となる．譲渡データの転送時間を $T_{migration}$，譲渡データのサイズを S_{ckpt}，脱退ドメインのプロセス数（CP 数）を N_{CP}，両ドメイン間の WAN のバンド幅を B_{WAN} とすると，

$$T_{migration} = \frac{S_{ckpt} \times N_{CP}}{B_{WAN}} \quad (1)$$

となる．たとえば，4.4 節で用いたアプリケーション（BT）のチェックポイントデータサイズは約 90 M バイトであった．WAN のバンド幅を 100 Mbps，メイン内のプロセス数を 100 とすると，$T_{migration} = 720$ 秒となる．

4. 実装と評価

本研究では，MPI アプリケーション用の Eagle の実装である MPICH-EG を開発している．MPI は並列アプリケーションを記述するための枠組みとして広く普及している．そのため，MPI アプリケーションを対象としたフォールトトレラントシステムを考察することは，計算グリッドのフォールトトレランスを考えるうえで有意義である．

本章では，CP–IMR 間のメッセージ通信部の実装を説明し評価するとともに，チェックポインティング手法を検討する．通信部の評価では 1 対 1，全対全の通信性能の評価にマイクロベンチマークを用いた．また，一般によく知られている，実アプリケーションに近い NPB 2.3 のアプリケーションを用い，実際の通信性能を評価した．また，NPB 2.3 のアプリケーションを用い，本研究で提案するチェックポインティングのスケジューリングによる性能の変化を評価し，MPICH-EG に適したチェックポインティング手法を検討した．評価環境として，独自に構築した PC クラスタ Swallow を用いた．Swallow の仕様を**表 1** に示す．

4.1 メッセージ通信部の実装

各 CP は，他の CP への接続情報として CP に対応する IMR への接続情報を持つ．これにより，自分以外の CP 宛のメッセージは IMR に送信される．IMR は，メッセージを受け取るとメッセージを保存した後，

宛先の CP に受け取ったメッセージを送信する．

MPICH-EG は Globus の API を用いて実装されている．Globus はアプリケーションのプラットフォーム独立を実現するため，ソケット通信，スレッド制御のためのプラットフォーム非依存な共通の API を提供する．本実装では，広く普及しているという理由から Globus を使用した．

4.2 メッセージ通信部の評価

MPICH-EG を用いた 1 対 1 通信の性能を測定するため，MPI_Send()，MPI_Recv() を用い，相手にメッセージを送った後，同一サイズのメッセージを受信するピンポンプログラムを作成し，メッセージが 1 往復するのに要した平均時間を測定した．本実験では CP 数を 2 とし，各 CP に別の IMR を割り当てた．結果を**図 4** に示す．横軸が送信メッセージのバイト数であり，縦軸がメッセージが 1 往復するのに要した時間である．図 4 の結果によると，メッセージサイズにかかわらず MPICH-EG は MPICH-G2 の約 2 倍の通信時間がかかっていることが分かる．これはメッセージを一度 IMR に送り，IMR が宛先の CP にメッセージを送信するため，直接通信した場合は 1 回で済むメッセージ送信が 2 回行われたことが原因である．この通信時間の増加は，中継ノードを介してメッセージ通信を行う Eagle のアーキテクチャ上やむをえない．MPICH-EG の 1 対 1 の通信時間が MPICH-G2 の通信時間の約 2 倍の増加に抑えられていることから，IMR でのメッセージの記録処理は MPICH-EG の 1 対 1 通信の性能にほとんど影響を与えないことが分かった．

次に，MPI_Alltoall() の実行時間を計測するマイクロベンチマークを実行することで，全対全通信の性能を評価する．結果を**図 5** に示す．横軸が送信メッセージのバイト数であり，縦軸が MPI_Alltoall() を 1 回実行するのに要した時間の平均値である．また，図 5 の左上に示した MPICH-EG: nCP/IMR は，1 つの IMR に対応するプロセス（CP）が n 個であることを表している．図 5 の結果によるとメッセージサイズにかかわらず MPICH-EG における MPI_Alltoall() の実行時間は，MPICH-G2 での実行時間の約 $n+1$ 倍になっている．MPI_Alltoall() は全プロセスが同時に通信を開始するため，メッセージが IMR にプロセス数分集中することになる．そのため，1 メッセージが利用できるバンド幅が IMR に割り当てられたプロセス数で分割された値になってしまったことが原因であると考えられる．この結果は，同時刻にメッセージが IMR に対応する CP 数分だけ IMR に集中する最悪の場合の MPICH-EG の通信性能を表しているといえる．この結果も，1 対 1 通信の場合と同様，Eagle のアーキテクチャ上やむをえない．MPICH-EG の全対全通信のオーバヘッドをバンド幅の分割に起因するオーバヘッドのみに抑えられていることから，IMR でのメッセージの記録処理は MPICH-EG の全対全通信の性能にほとんど影響を与えないことが分かった．

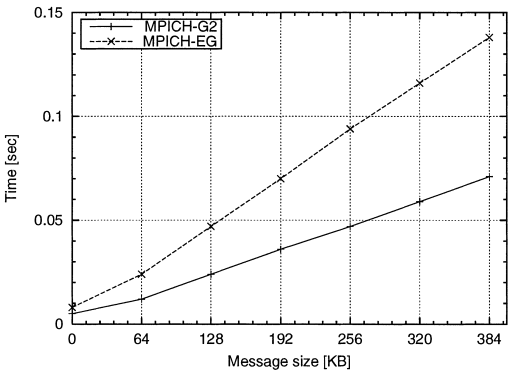

図 4 ピンポンプログラムによる 1 対 1 通信性能の比較
Fig. 4 Comparison of a point-to-point communication performance by Pingpong program.

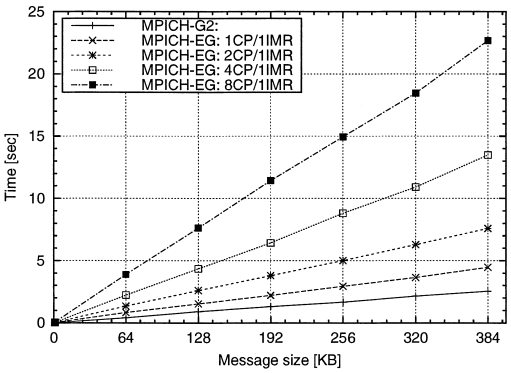

図 5 MPI_Alltoall() による全対全通信性能の比較
Fig. 5 Comparison of a all-to-all communication performance by executing MPI_Alltoall().

次に，実際にアプリケーションを実行した場合に発生する実行時間のオーバヘッドを測定するため，実アプリケーションに近い NPB 2.3 の IS，BT，SP を MPICH-EG を用いて実行し，MPICH-G2 を用いた場合の実行時間と比較した．本実験では 1 つの CP に対し 1 つの IMR を割り当てた．各ベンチマークの問題のクラスは A とし，IS は 8 個の CP，BT，SP は 9 個の CP で実行した．各ベンチマークプログラムでの実行オーバヘッドの測定結果を**表 2** に示す．表 2 の結果によると IS の実行時間のオーバヘッドは約 63%と，BT，SP と比べて 2 倍以上になることが分かる．こ

表 2 実行オーバヘッド
Table 2 Execution overhead.

ベンチマーク	オーバヘッド [%]
IS	63.4
BT	19.9
SP	29.9

れは，IS は実行時間が短く集団通信が多いため，実行時間に対する通信時間の割合が高いことが原因となっていると考えられる．対して BT は約 20%，SP 約 30%の実行オーバヘッドとなっている．BT，SP は IS と比べて集団通信によるメッセージのレイテンシによる性能低下が小さかったことが原因と考えられる．MPICH-EG で並列アプリケーションを実行した場合，集団通信の少ないもので 20%から 30%，集団通信の占める通信が多いアプリケーションでも 60%程度の実行オーバヘッドに抑えられることが分かった．

本実験はドメイン内に属するプロセスを用いて行った．複数ドメインに属するプロセスを利用して MPICH-EG を動作させた場合，CP，EMR，WAN，EMR，IMR，CP の順にメッセージが転送される．MPICH-EG の通信性能は WAN の通信遅延の大小によって大きく左右されると考えられる．WAN の通信遅延が大きければ大きいほど EMR，IMR を介することによる通信遅延は隠蔽され，オーバヘッドは減少傾向になる．

4.3 チェックポインティング部の検討

Eagle において，ドメイン単位での Pessimistic logging によるリカバリを実現するためには，上記の IMR によるメッセージ記録のほか，CP の実行状態を記録・保存するチェックポインティングが必須である．一般にチェックポイントデータは通信メッセージと比べてサイズが大きいため，本稿ではグリッド環境に適用可能な現実的なチェックポインティング手法について実験的な方法で検討する．

チェックポインティングの処理が実際のアプリケーションプログラムに及ぼす影響は，チェックポイントデータが Csvr に集中することによる負荷に起因するものと，CP がチェックポイントデータを作成し Csvr に転送する負荷に起因するものとに分けられる．前者は各 CP でのチェックポインティングの開始タイミングにより制御できる．後者は，チェックポイントデータの送出を同一プロセスで行うか，別プロセスにするかに分けて評価することにした．我々は，これら 2 つの要因ごとに考えられる手法をあげ，要因ごとの組合せにより各チェックポインティング手法が実際のアプリケーションプログラムの実行時間に及ぼす影響を測定した．

各 CP でのチェックポインティングの開始タイミングを決める方法として，

- Sequential Request（SR）：あらかじめ設定された一定時間ごとのインターバルに従い，Csch が LM を介し各 CP に対して**逐次に**チェックポインティング要求を送信する，
- Batch Request（BR）：あらかじめ設定された一定時間ごとのインターバルに従い，Csch が LM を介し各 CP に対して**いっせいに**チェックポインティング要求を送信する，
- Self Timer（ST）：あらかじめ各 CP に一定時間のインターバルを設定し，そのインターバルごとに**自律的に**チェックポインティングを開始する，

の 3 つを考えた[13]．Sequential Request では，チェックポイントデータがいっせいに送られることがないため，Csvr の負荷は平均化される．逆に Batch Request では，チェックポイントデータがいっせいに Csvr に送られる．Self Timer では，チェックポインティングの開始時刻が各 CP のタイマに任されるため，全体的にチェックポインティングの時期が分散することが期待できる．

CP でのチェックポイントデータの生成・送信方法は，UNIX の fork() システムコールの使用の有無により，以下の 2 方式について検討した．

- Synchronous Checkpointing（SC）：fork() せずに現在のプロセスイメージをすべて Csvr に転送してからアプリケーションの実行を続ける．
- Asynchronous Checkpointing（AC）：fork() システムコールにより，子プロセスを生成し，現在のプロセスイメージを子プロセスにコピーする．プロセスイメージは親・子のプロセスに複製されるので，親プロセスではそのままアプリケーションプログラムの実行を継続する．子プロセスは，プロセスイメージからチェックポイントデータを作成し，それを Csvr に転送する処理を，親プロセスと並行して行う．転送終了後，子プロセスは終了する．

同期式の SC では送信にかかる時間がほぼそのままアプリケーションの実行時間に加算されオーバヘッドとして顕在化することが予想される．非同期式の AC では，同期式の SC とは異なりチェックポイントデータの送信による待ちは生じないが，その代わりにプロセスイメージを複製することにより，OS レベルでの処理およびメモリ使用量のオーバヘッドが生じる．また，子プロセスでのチェックポイントデータの送信の

負荷が親プロセスに与える影響も考えられる．チェックポイントデータを Csvr に送信する処理は基本的に I/O バウンドであるから，非同期式の AC の場合親プロセスへの影響は軽微であると予想される．しかし本稿では同期式の SC，非同期式の AC 両者のオーバヘッドについて実験により定量的に評価を行うこととした．

本稿では，提案手法の実装が容易であることからチェックポイントのベースとして ckpt[5] を用いた．ckpt は，チェックポインティング/リスタートを行うためのライブラリである．CP はチェックポインティング機能を得るため，計算開始時に ckpt をダイナミックリンクする．CP は，特定のシグナルを受け取ることで ckpt が提供する関数に制御を移し，チェックポイントデータを作成する．チェックポインティングは Csch が LM 宛にメッセージを送ることで指示し，LM が CP にシグナルを送ることで開始される．

ckpt はチェックポイントデータの作成時，復旧時にそれぞれ setjmp()，longjmp() を使用するため，プラットフォームに依存する．本稿では，チェックポインティング手法の検討とオーバヘッド量の見積りのため，ckpt を用いた．MPICH-EG の実装にあたっては，ckpt とは別のプラットフォーム非依存なチェックポインタを用いる．プラットフォーム非依存なチェックポインタの例として CosMic[14] がある．

4.4 チェックポインティング部の評価

チェックポインティングによる実行時間のオーバヘッドを測定するため，NPB 2.3 の BT を用い，実行時間のオーバヘッドを測定した．アプリケーションプログラム中で，並列実行を開始する部分，終了する部分に時刻を計測する関数 MPI_Wtime() を挿入することにより，並列処理部分の正味の実行時間を測定している．アプリケーションプログラムはプロセス数（CP 数）4 で実行し，各 CP および Csvr はそれぞれ別のプロセッサで実行させた．本評価中にメッセージ記録は行っていない．そしてチェックポインティングを行った場合/行わない場合の実行時間を上記方法により測定し，チェックポインティングによるオーバヘッドを求めた．

チェックポインティング手法は前節で説明したように，{SC, AC}×{SR, BR, ST} による 6 通りの組合せ，すなわち SC&ST，SC&SR，SC&BR，AC&ST，AC&SR，AC&BR となる．これら各チェックポインティング手法でのオーバヘッドの測定結果を**図 6** に示す．横軸はベンチマーク実行時に行ったチェックポインティングの回数を表し，縦軸は実行オーバヘッドを表す．

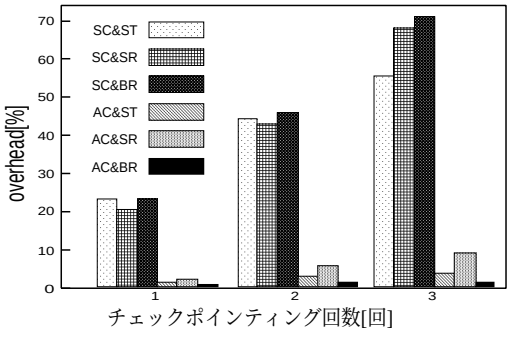

図 6 チェックポインティングのオーバヘッド
Fig. 6 Checkpointing overhead.

非同期式の AC と同期式の SC を比較すると，チェックポインティング手法，チェックポインティング回数にかかわらず非同期式の AC の方が圧倒的にオーバヘッドが少ない．前節での検討のとおり，同期式の SC ではチェックポイントデータの送出が完了するまで CP は計算を進めることができないため大きなオーバヘッドとして現れている．これに対し，非同期式の AC ではチェックポイントデータの送出を行っている時間のほとんどをアプリケーションの実行に利用することが可能であるため同期式の SC に比べ劇的にチェックポインティングのオーバヘッドを削減できていることが確認できる．

図 6 から，同期式の SC でのチェックポインティングのオーバヘッドは，ネットワークのバンド幅や Csvr の負荷状況により変動することが分かる．チェックポイントデータが Csvr に向かっていっせいに送られる BR の方式では，ネットワークのバンド幅や Csvr の処理負荷により転送時間が長くなる．この転送中はアプリケーションの処理が行われずオーバヘッドになることから，各 CP での転送時間が最も長くなる BR の方式で最も大きいオーバヘッドとなる．これに対してチェックポイントデータを順次転送する SR では BR よりもオーバヘッドが軽減される．

また図 6 によると，チェックポインティング回数が多いほど，各 CP が自律的にチェックポインティングを行う ST のオーバヘッドが相対的に低くなっている．これは，チェックポインティングの開始時期がチェックポインティングの実行によって分散されたことで，セルフタイマにより Csch との通信オーバヘッドが存在しないことが有利に働いた結果であると考えられる．

同期式の SC の場合は，チェックポイントデータが Csvr に向かっていっせいに送られる BR が最もオーバヘッドが大きいのに対して，非同期式の AC では逆

に BR が最も低いオーバヘッドとなっている．これは以下のように説明されるものと考えている．

非同期式の AC 方式により子プロセスがチェックポイントデータを Csvr に送信する場合であっても，送信のための負荷は（軽微であっても）ゼロではないため，計算を続行している親プロセスの実行にも影響する．評価に用いたプログラムは MPI によりプロセス間（CP 間）相互に通信しているため，送信・受信のどちらか一方でも処理に遅れが生じると，他方のプロセスに待ちを生じることになり，これが結局全体の処理の遅延となって現れる．BR 方式の場合は，チェックポイントデータの転送処理が全 CP について同時に行われるため，全 CP が同時に遅延することになる．このため，MPI メッセージの送受信時の待ちは，チェックポイントデータが Csvr に向かっていっせいに送られている間に限られる．

これに対して，SR ではチェックポイントデータの送信が各 CP について順番に行われる．このため，系内でたかだか 1 個の CP の（親プロセスの）処理が遅延し，これと通信関係にある他のプロセスを待たせることになる．こうした MPI データ送受信の待ちは，CP についてチェックポイントデータの転送が終わるまで続くものと考えられる．このために，全体の処理のオーバヘッドが BR に比べ大きくなる．各 CP が自律的にチェックポインティングを行う ST についても同様であり，チェックポインティングの開始タイミングが分散することにより比較的大きなオーバヘッドになっているものと考えられる．

以上から，非同期式の AC により同期式の SC に比べ十分に低いオーバヘッドでチェックポインティングが可能であることが分かる．チェックポインティングの開始タイミングについてはどうか．図 6 の結果によれば，AC&BR が最良であるが，各 CP からのチェックポイントデータが Csvr に集中することによる現実的な問題も考慮しなければならない．Csvr では，各 CP から送られてくる巨大なチェックポイントデータを，安定したストレージに保存し終わるまでメモリ上に保存しておく必要がある．アプリケーションによっては，上述の SR のような方法により，チェックポイントデータの転送を制御する必要があろう．すなわち，BR ないし SR の方法をアプリケーションの特性に応じて切り替えるのが現実的といえる．BR，SR いずれの場合であっても，Csch によるチェックポインティングの開始タイミングの制御が必要である．

5. 関連研究との比較

本研究で比較・検討の対象としたフォールトトレランス手法については，2 章で概略を説明し MPI の実装例をあげている．ここでは，MPI にフォールトトレランス機能を持たせたシステムである MPICH-V，MPICH-V2，MPICH-GF のそれぞれについて，以下の観点で本稿によるシステム MPICH-EG との比較を行う．

- MPICH のデバイス
- ベースとするフォールトトレランス手法
- 通信性能，リカバリ性能

なお，MPICH のデバイスとは，ユーザに提供される MPI_Send()，MPI_Recv() 等の API の実装となる部分を指す．

MPICH-V は，LAN 内のワークステーションを結合して構築されたクラスタを対象とした MPICH のデバイス ch_p4 をベースとして実装されている．ch_p4 は計算に参加するノードおよびネットワークが均質である計算環境を対象としたデバイスなので，LAN 内に閉じた環境で使用する場合には有効に働くものと考えられる．しかし，計算に参加するノードおよびネットワークが不均質であることが考えられる大規模な計算グリッドには適していない．提案手法の MPICH-EG では，計算に参加するノードおよびネットワークが不均質な計算グリッドへの適用を前提として Globus 上に作られた MPICH のデバイス globus2 を使用している点で大きく異なる．

フォールトトレランス手法は，MPICH-V においても MPICH-EG と同様の Pessimistic logging を用いている．MPICH-V では通信メッセージの中継ノードを Channel Memory（CM）と呼ぶ等若干の差異が認められるが，基本的な動作は MPICH-EG のドメイン内でのそれとほぼ同じである．

このため，基本的な通信性能は MPICH-V，MPICH-EG で同等である．ただし，ドメイン間の通信に関しては，MPICH-V がドメイン内外の区別なく通信をするのに対して，MPICH-EG ではドメイン間通信の際に外部中継プロセス EMR を経由する必要があるため，通信レイテンシがやや増える．

一方で，障害からのリカバリに関しては，MPICH-V ではドメイン境界に関係なく WAN を経由して回復のための処理が行われるのに対し，MPICH-EG ではドメイン内，すなわちローカルな処理で済む．このため MPICH-EG のほうがリカバリ時間が短い．

MPICH-V2 も，MPICH-V と同じ MPICH のデバ

イス ch_p4 をベースに構築されている．このため大規模計算グリッドへの適用性は MPICH-V と同様である．フォールトトレランス手法は，Sender based message logging である．MPICH-V2 では，このフォールトトレランス手法をベースとすることによって，計算プロセス間で直接通信をすることが可能になっている．またチェックポインティングも，各計算プロセスで独立に行うことができる．

計算プロセス間で直接通信を行うため，通信レイテンシは中継プロセスを用いる MPICH-V，MPICH-EG の半分程度である．しかし，MPICH-V2 では送信プロセスが通信メッセージを自身で保存する方式であるため，障害からのリカバリ時には回復プロセスがシステム内の全プロセスから保存メッセージを回収する必要がある．数多くのプロセスが WAN を介して接続される大規模計算グリッドでは，リカバリに極端に時間がかかり現実的ではないと考えられる．一方，MPICH-EG では，リカバリに必要な情報はすべてドメイン内（LAN 内）から得られるため，大規模グリッドでも実用的なリカバリ性能が得られると考えられる．

MPICH-GF は，上記 2 者とは異なり，Globus 上に作られた MPICH のデバイス globus2 を実装のベースとしている．この点は，本稿での提案である MPICH-EG と同様である．

MPICH-GF は，フォールトトレランス手法としてチェックポイントベースの手法（2.1 節）を用いている点で，MPICH-EG と異なる．システム内の全ノードで同期をとり，いっせいにチェックポインティング動作を行う．システム内に障害が発生した場合は，全プロセスでロールバックリカバリを行う．このために，MPICH-GF では通信メッセージの記録・保存の必要がなく，計算ノード間での直接通信ができる．通信レイテンシは，中継プロセスを用いる MPICH-V，MPICH-EG の半分程度である．ただし，チェックポインティングを行う前に全プロセスで同期をとる必要があるため，大規模計算グリッドではそのためのオーバヘッドが無視できない大きさになる．このオーバヘッドを抑えるためにチェックポインティングの間隔を長くすると，リカバリのペナルティが大きくなる．全プロセスが同時に過去の状態（チェックポインティング時）に戻るためである．

これに対して MPICH-EG ではフォールトトレランス手法として Pessimistic logging を用いているため，計算プロセスは他のプロセスと同期をとる必要なく独立にチェックポインティングすることが可能である．また，障害からの回復の際にロールバックを行うのは，障害が起きた当該プロセスのみであり，ロールバックにともなう実行時間のペナルティを最小限に抑えることができる．

またさらに MPICH-EG では，計算資源を提供しているドメインが，実行中のプロセスを他のドメインに移動させ，自身は計算から脱退することを可能にする，ドメイン間でのプロセスの「譲渡」の概念を導入し，フォールトトレランス機能とあわせて実現することを図っている．この概念および実現手法は，上記の MPICH-V，MPICH-V2，MPICH-GF にはなく，Eagle およびその MPI 実装である MPICH-EG に独自のものである．

6. まとめと今後の課題

計算グリッド向けフォールトトレラントシステム Eagle を提案し，MPI アプリケーションを実行するための計算グリッドを対象とした Eagle の実装である MPICH-EG の通信性能を評価し，チェックポインティング手法を検討した．

通信性能の評価では，ピンポンプログラムを用いて MPICH-EG の 1 対 1 通信の性能を評価した．また，MPI_Alltoall() の実行により MPICH-EG の全対全通信の性能を評価した．加えて，実アプリケーションに近い NPB 2.3 のアプリケーションを実行することで，MPICH-EG の実通信性能を評価した．その結果，MPICH-EG の通信レイテンシの増加は 1 対 1 通信，全対全通信とも IMR を介することに起因するもので，IMR でのメッセージの記録処理は通信レイテンシにほとんど影響を与えないことが分かった．また，全対全通信の割合の少ない NPB 2.3 の BT，SP でそれぞれ約 20%から 30%の実行時間のオーバヘッドとなり，全対全通信の割合の多い NPB 2.3 の IS でも実行時間のオーバヘッドを約 60%に抑えられることが分かった．

チェックポインティング手法の検討では，本研究で提案しているチェックポインティングの各手法を NPB 2.3 の BT を用いて評価した．その結果，チェックポインティングを fork() システムコールを利用して子プロセスに行わせることにより，アプリケーションの実行オーバヘッドを大幅に削減できることが分かった．また，チェックポインティングを行うプロセスとは別のプロセスがチェックポインティングのタイミングを管理することにより，アプリケーションに適したチェックポインティングを行えることが分かった．

今後は，メッセージの送信，保存の手法をより効率化し，MPICH-EG の性能向上を図る．そのうえで，リカバリ性能を含めて詳細に MPICH-EG の性能を評

価する．また，プロセス譲渡機能を実装し，評価する．

謝辞 本研究は，一部日本学術振興会科学研究費補助金（基盤研究（B）14380135，同（C）16500023，若手研究 14780186）の援助による．

参 考 文 献

1) Foster, I., et al.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations, *International Journal of Supercomputer Applications*, Vol.15, No.3, pp.200–222 (2001.1).
2) Karonis, N., et al.: MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface, *Journal of Parallel and Distributed Computing*, Vol.63-5, pp.551–563 (2003.5).
3) The Globus Alliance. http://www.globus.org/
4) Bailey, D., et al.: The NAS Parallel Benchmarks 2.0, Report NAS-95-020, Numerical Aerodynamic Simulation Facility NASA Ames Research Center, Mail Stop T 27A-1 Moffett Field, CA 94035-1000, USA (1995.12).
5) Zandy, V.: ckpt: A process checkpoint library. http://www.cs.wisc.edu/~zandy/ckpt
6) Elnozahy, E., et al.: A survey of rollback-recovery protocols in message-passing systems, Technical Report CMU-CS-99-148, Carnegie Mellon University (1999).
7) Stellner, G.: CoCheck: Checkpointing and Process Migration for MPI, *The 10th International Parallel Processing Symposium*, pp.526–531 (1996.4).
8) Agbaria, A., et al.: Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations, *The 8th IEEE International Symposium on High Performance Distributed Computing* (1999).
9) Woo, N., et al.: MPICH-GF: Providing Fault Tolerance on Grid Environments, *The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid the poster and research demo session* (2003.5).
10) Bosilca, G., et al.: Toward a Scalable Fault Tolerant MPI for Volatile Nodes, *IEEE/ACM Super Computer 2002* (2002.11).
11) Bouteille, A., et al.: MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on the Pessimistic Sender Based Message Logging, *IEEE/ACM Super Computer 2003* (2003.11).
12) Elnozahy, E., et al.: Replicated Distributed Processes in Manetho, *22nd International Symposium on Fault-Tolerant Computing*, pp.18–27, (1992.7).
13) 薬師寺健太，服部晃和，横田隆史，大津金光，古川文人，馬場敬信：グリッド環境におけるチェックポインティング手法の検討と初期評価，情報処理学会第 66 回全国大会講演論文集，pp.1-135–1-136 (2004.3).
14) Chung, P., et al.: Checkpointing in CosMiC: a User-level Process Migration Environment, *Pacific Rim International Symposium on Fault-Tolerant Systems*, pp.187–193 (1997.12).

(平成 16 年 1 月 31 日受付)
(平成 16 年 5 月 9 日採録)

服部 晃和（学生会員）
2003 年宇都宮大学工学部情報工学科卒業．現在，宇都宮大学大学院工学研究科情報工学専攻在籍．分散システムを対象としたフォールトトレランス手法に興味を持つ．

薬師寺健太
2004 年宇都宮大学工学部情報工学科卒業．現在，宇都宮大学大学院工学研究科情報工学専攻在籍．並列計算を対象としたフォールトトレランス手法に興味を持つ．

横田 隆史（正会員）
1983 年慶應義塾大学工学部電気工学科卒業．1985 年同大学大学院電気工学専攻修士課程修了．同年三菱電機（株）に入社．1993 年 12 月から 1997 年 3 月まで新情報処理開発機構（RWCP）に出向．2001 年 4 月より宇都宮大学工学部助教授．計算機アーキテクチャ，設計方法論等の研究に従事．工学博士．ICCD Outstanding Paper Award（1995），FPGA/PLD Design Conference 審査委員特別賞（2002）各受賞．電子情報通信学会，IEEE 各会員．

大津 金光（正会員）

1993 年東京大学理学部情報科学科卒業．1995 年同大学大学院修士課程修了．1997 年東京大学大学院博士課程退学，同年より宇都宮大学工学部助手となり現在に至る．計算機システムの高性能化に関すること，特にマルチスレッドアーキテクチャ，バイナリ変換処理，実行時最適化等に興味を持つ．

古川 文人（正会員）

1998 年宇都宮大学工学部情報工学科卒業．2000 年同大学大学院博士前期課程修了．2003 年同大学大学院博士後期課程修了．同年より宇都宮大学ベンチャー・ビジネス・ラボラトリ非常勤研究員．博士（工学）．高性能計算機システムに興味を持つ．

馬場 敬信（正会員）

1970 年京都大学工学部数理工学科卒業．1975 年同大学大学院博士課程単位取得退学．同年より電気通信大学助手，講師を経て，現在宇都宮大学工学部教授．工学博士．1982 年より 1 年間メリーランド大学客員教授．計算機アーキテクチャ，並列処理等の研究に従事．1992 年情報処理学会 Best Author 賞，2002 年 FPGA/PLD Design Conference 審査委員特別賞，PDCS2002 国際会議 Best Paper Award 各受賞．著書 “Microprogrammable Parallel Computer”（MIT Press），『コンピュータアーキテクチャ（改訂 2 版）』（オーム社）等．電子情報通信学会，IEEE 各会員．