自律的な通信回復を行う Fault Tolerant MPI の実装と評価

實本英之 † 高宮安仁 † 松岡聡†,††

クラスタシステムでは、ノード数の増大によりシステム全体の障害発生の潜在的可能性が高い。そのため、長時間にわたる計算を安定して行うには、耐故障性を持ったミドルウェアが必要になる。本研究では、耐故障性を持った MPI の実装と評価を行った。この MPI は逐次プロセスのチェックポインタと耐故障性通信路により MPI プロセスのチェックポインティング／リスタートを行う。ベースとして MPICH を用い、通信路の耐故障性は Rocks ライブラリ、チェックポインタは ckpt ライブラリを用いた。32 プロセスを用いた NPB-CG の結果、本実装では、オーバーヘッドがオリジナル MPICH の高々8%程度に抑えられることを確認した。

Implementation and Evaluation of a Fault Tolerant MPI with Reliable TCP/IP Sockets

Hideyuki Jitsumoto,† Yasuhito Takamiya† and Satoshi Matsuoka†,††

On cluster systems, failure rates tend to be high due to a large number of constituents. Therefore, to perform stable long-time computation on cluster systems, middleware support for fault-tolerancy is inevitably required. We implemented a fault-tolerant MPI prototype system and measured the overhead of the system. Our MPI system implements coordinated checkpointing and recovery protocol on MPICH using a single process checkpointer called *ckpt* and a reliable network called *Rocks*. Preliminary evaluation using NPB-CG with 32 processes showed the overhead posed by Rocks stayed within just 8% .

1. はじめに

クラスタシステムは今日、さまざまな分野で広く用いられる。しかしながら、コモディティパーツの利用や、台数増加による障害発生の増加などから、信頼性に大きな問題を抱えている。またクラスタリングシステムの代表的なアプリケーションである科学技術計算は、そのほとんどが MPI により実装されている。加えて、その実行時間は、数時間から数日に及ぶことが常である。システムのどこかに障害がおき、計算が続けられなくなると、それまでの長時間に及ぶ計算結果が一切無駄になる。したがって MPI プロセスの実行に対して耐故障性を実現することは、クラスタシステムを用いるに当たり急務であるといえる。こうしたニーズから並列プロセスの耐故障性についてさまざまな研究が行われてきている[4),7)]。しかしながら、このようなシステムを実環境において実証した例は少なく、評価データが少ないのが実状である。

† 東京工業大学
Tokyo Institute of Technology
†† 国立情報学研究所
National Institute of Informatics

本稿では、耐故障性を実現する MPI の設計、実装について述べる。このシステムは耐故障性通信路とチェックポインタ、およびスケジュールを管理するデーモンにより実現され、ユーザー透過☆に MPI プロセスのチェックポインティング／リスタートを行うことを可能とする。さらに、この耐故障性 MPI を用いた性能評価結果も示す。

以下、2 節では本研究に用いた要素技術について述べ、3 節において、既存の並列チェックポインティングについて、および本研究のアプローチについて述べる。また 4,5 節ではプロトタイプ実装について述べ、6 節ではその評価を行う。最後に 7,8 節において関連研究とまとめを述べる。

2. 要素技術

2.1 Rocks

Rocks[11)] はネットワークの障害をユーザー透過に対処するソケットを実現するライブラリである。

物理的な障害もサポートしており、同時に 1 プロセスまでであれば、通信ケーブルの切断や IP アドレス

☆ 本研究ではユーザー透過を、ユーザーコードの改変の必要のないものと定義する

の変更が起こっても通信を維持することが可能である。

Rocks は TCP/IP が用意している send/recieve バッファのほかに、現在 in-flight であるメッセージを保存するバッファを用意する。これに加え、各ノードで送受信したパケット数を比較することによってメッセージが失われるのを防いでいる。具体的には Rocks は通信路上に障害が発生すると、以下のように通信路を修復する。

- 双方がリスニングソケットを開く。
- 双方がかつて通信していた IP アドレスに接続要求を出す。
  - IP アドレスに変更のなかった側は相手へ通信を確立できない。
  - IP アドレスを変更した側は相手へ通信を確立することができる。
- 通信が確立できていないソケットを閉じる

以上によって通信路が再接続されると、双方の送受信したパケット数を比較し、不足分を in-flight バッファから再送する。

2.2 ckpt

ckpt[10] はプロセスのチェックポインティング／リスタートを行うライブラリである。

このライブラリはプロセスのメモリ空間、シグナルハンドラとその状態をチェックポイントする。また、動的リンクライブラリのチェックポイントも可能である。対してファイルオープンの状態、共有メモリ空間の保存は行わない。後者に関しては同一ノード上で二つ以上のプロセスを動かす場合に必要であるため、本研究において今後改良していく予定である。チェックポインティングは、setjmp, longjmp を用いてメモリ空間のコピーを作成しそれをファイルに書き込むことで実現される。

3. チェックポインティングアルゴリズム

3.1 MPI プロセスのチェックポインティング

耐故障性を実現するに当たり、頻繁に用いられる技術はチェックポインティング／リスタートである。並列プロセスにおいて、この手法の実装例は少ない。これは、並列プロセスのチェックポインティングは通信の一貫性の実現が難しく、煩雑なためである。しかしながら逐次プロセスにおける実装例は豊富であり、また並列プロセスにおいても既存研究において様々なシステムモデルが提案されている。

通信の一貫性を確保したチェックポインティングを行うには、『チェックポイント前に受信した全てのメッセージの送信者が、メッセージを送信したことを確認できる状態』[3] でチェックポインティングする必要がある。良く知られた一貫性モデルとして、以下のようなものがある。

Coordinated Checkpointing

チェックポインティング時に各プロセスが同期を取り通信の一貫性を保障する。これにより、必ず一貫性のあるチェックポイントをとることができる。

同期の種類としては全ての通信が終了するまで待ち、通信路を一度空にするもの、またチェックポインティング終了を待ち、それまでの一切の送信を行わないようにするものなどがある。

このような方法では同期のオーバーヘッドが大きい。しかしながら、全てのチェックポイントが一貫性を保持しているため無駄なチェックポイントを作ることが無く、古いチェックポイントは破棄することができる。

Uncoordinated Checkpointing

各プロセスが任意にチェックポインティングを行い、復旧時に通信の一貫性が保たれるチェックポイントを選ぶ。

この方法では同期を行わないため、同期によるオーバーヘッドがない。しかしながら最後の一貫性の取れるチェックポインティング以降にとられたチェックポイントは全て無駄になってしまう（ドミノ効果）。

Communication-induced Checkpointing

各プロセスがある特定の条件の下にチェックポインティングを行う。これにより同期を行わず、さらにドミノ効果の発生を防ぐこともできる。

このような条件は複数知られている。MRS model[8] と呼ばれる条件は、あらゆる受信があらゆる送信よりも先に起こればドミノ効果が起こらないというもので、全ての受信の直前にチェックポインティングを行なう。

3.2 本研究のアプローチ

本研究では実装の容易さを考慮して、前述したアルゴリズムの内の Coordinated Checkpointing を用いる。

チェックポインティングが始まると、MPI 構成プロセス☆はそれ以降通信を行わなくなる。また、通信中であった場合も強制的に以降のメッセージが送受信できなくなる。チェックポインティングのタイミングは全てデーモンにより管理されており、MPI 構成プロセスすべてのチェックポインティングが終了すると、デーモンの指示により MPI 構成プロセスは通信を再開する。再開後、強制中断により失われたメッセージを再送する。

リスタート時は、各 MPI 構成プロセスのチェックポイントをそれぞれのノードで再開する。このとき、通信網はすべて断絶されているため、まず再接続を行う。再接続後、MPI 構成プロセスが通信を再開し、チェックポインティング時と同様に失われたメッセージを再送する。

☆ クラスタ上で実行されている MPI のプロセス全体を「MPI プロセス」とし、各ノードで実行されている MPI プロセスを構成するプロセスを「MPI 構成プロセス」とする

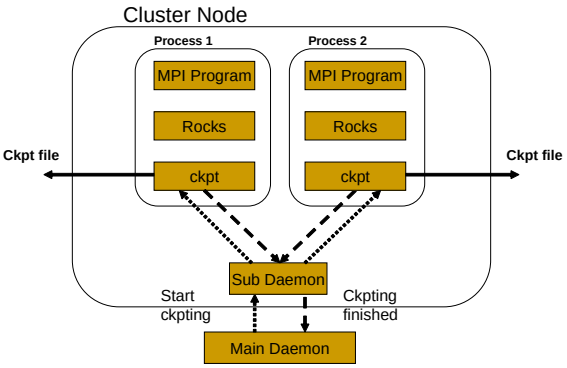

図 1 ノード内プロセス構成図

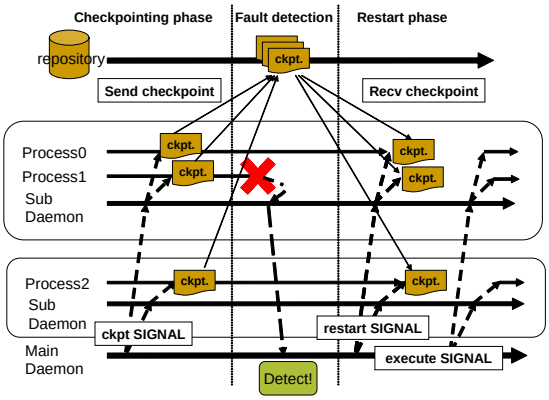

図 2 システム全体図

以上のうち、メッセージの再送、通信の再接続は耐故障性通信路、チェックポインティングは既存のチェックポインタに任せることにより、実装のコストを削減する。

4. システムデザイン

本研究で提案する MPI プロセスは、図 1 中で示されるような逐次プロセスの集合で構成されている。プロセスは MPI プログラム、耐故障性通信路、チェックポインタからなり、サブデーモンを通じてメインデーモンから命令を受けると、ネットワーク上の指定されたレポジトリにチェックポイントを作成する。

システムの全体図を、図 2 に示す。メインデーモンがチェックポイントを一元的に管理し、チェックポインティングの開始と終了、リスタートのタイミングをノードごとのサブデーモンに伝える。サブデーモンはノードごとの MPI プロセスの PID を知っており、メインデーモンの命令を伝える。また、メインデーモンは耐故障性通信路、およびサブデーモンにより通信路の故障を検出すると、全てのプロセスを中断させ、チェックポイントファイルを各ノードに転送する。さらに、ノードのハードウェア的な不良を検知し、リスタートとマイグレーションのどちらにより処理を復旧するかが決定される。

5. 実　　装

本研究では、耐故障性通信路、チェックポインタに Rocks, ckpt ライブラリを用いた。ベースとなる MPI 実装としては MPICH v1.2.5 を用いた。また、MPICH に対する変更のほとんどは MPICH の通信ライブラリである p4 に対して行ったものであり、p4 ライブラリを用いた他のアプリケーションに対しても容易に適用可能である。

MPI プロセスの実行中は以下の 5 つの動作を各デーモンにより管理している。このうち、チェックポインティングは MPI プロセスの起動時に周期を設定することによって定期的に行われる。また、リスタート、マイグレーションは MPI プロセスに障害が起きたときに自動的に実行される。なおチェックポイントの転送については現状は NFS を利用している。このため、チェックポイントリポジトリとなる NFS サーバーに負荷が集中する。リポジトリの設計については今後の課題とする。

初期化

MPI プロセスは mpirun によって起動される。このとき mpirun は、メインデーモンに対し、チェックポインティングのタイミングなどの必要な情報を送信すると共に、ユニーク ID を要求する。mpirun は得られたユニーク ID を環境変数 PARAKEET_UID を通じて、各ノードで起動される MPI 構成プロセスに渡す。この MPI 構成プロセスは各ノードのサブデーモンに自分のプロセス ID とユニーク ID を伝える。

チェックポインティング

メインデーモンは、同じユニーク ID を持つすべてのプロセスをチェックポインティングするようサブデーモンに指示する。サブデーモンは、指示のあったユニーク ID を持つプロセスにチェックポインティング開始シグナルを送る。各 MPI 構成プロセスでチェックポインティングが開始すると、チェックポインティングルーチンにプログラムフローが移るため、通信が中断する。チェックポイントが作成されると、各プロセスはサブデーモンを通じてメインデーモンにチェックポインティング終了の通知を行う。メインデーモンはすべてのプロセスがチェックポインティングを終了したことを確認すると、サブデーモンを通じて実行再開のシグナルを出す。このシグナルによりチェックポインティングルーチンが終了し、実行が再開される。

リスタート／マイグレーション

各サブデーモンは自分のノードでどの MPI 構成

プロセスが動作しているか覚えている。そして、メインデーモンからの通知で覚えておいたプロセスのチェックポイントを再開させる。マイグレーションを行うときは、サブデーモンがノード内の負荷を測定しその結果からマイグレーション先のノードを決定する。メインデーモンはマイグレーションノードのサブデーモンにマイグレーションさせるプロセスのチェックポイントを指示し、リスタートと同じ手順で復旧させる。

フォルトディテクション

MPI プロセスのどこかに障害がおきると通信に障害が起こる。これを Rocks が検出し、通信不良ノードと共にサブデーモンを通じてメインデーモンに知らせる。メインデーモンは通信不良があったノードのサブデーモンが接続されているかを確認し、MPI 構成プロセスが異常終了したのか、ノード自体および通信ケーブルに異常があるのかを判断する。

5.1 実装上の制限

本システムでは Rocks,ckpt の制限により以下の制限がある。

- 同時に 2 箇所の障害が発生した場合、マイグレーション不能
- 1 ノードに MPI 構成プロセスが 2 プロセス存在する場合、ノード、通信ケーブルの故障に未対応
- ファイル I/O 中のチェックポインティングは不可能

5.2 現　　状

現状では、デーモンの実装は未完成である。このため、手動によるチェックポインティング／リスタートを行うことは可能だが、デーモンを用いた自動チェックポインティング／リスタート、およびフォルトディテクションは行うことが出来ない。また、チェックポイントの転送に NFS を用いているため、リポジトリである NFS サーバーが故障してしまうと耐故障性を実現できない。

6. 実験と考察

6.1 NPB-CG,MG による評価

Rocks を用いることによる性能変化を、プロセス数を増加させながらオリジナルの MPICH と比較した。実験には NAS Parallel Benchmark[1] の CG,MG を用い、実験環境は、東京工業大学松岡研究室の PrestoIII クラスタを用いた。各ノードは表 1 の構成である。

図 3 は、NPB-CG のプロセス数と性能の関係を示したものである。これによると、Rocks を用いた場合大きな性能低下が起きている。Rocks はソケットごとに、送信メッセージをコピーするバッファを持つ。また、通信路の障害検知の為に、シグナルハンドラによ

表 1 実験環境

CPU	Athlon 1900+×2
Memory	768MB
Network	100base-T
OS	Linux 2.4.18
MPICH	mpich-1.2.5

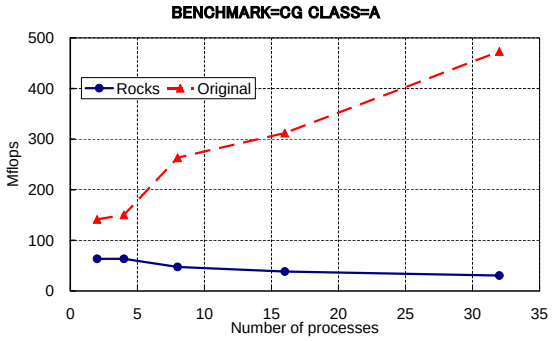

図 3 NPB-CG によるプロセス数とオリジナル MPICH および Rocks+MPICH の性能

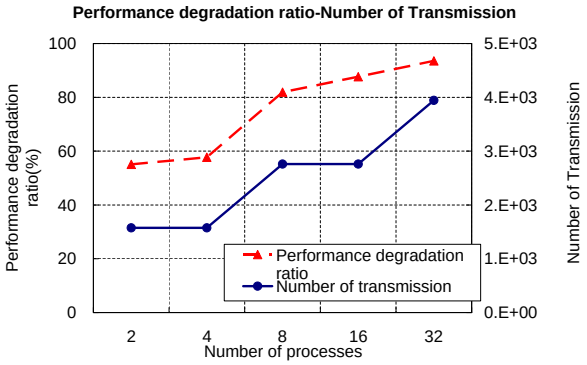

図 4 NPB-CG による Rocks 利用時の性能低下と通信回数の関係

り毎秒 1byte の送受信を行っている (Heart Beat)。これらがオーバーヘッドになると考え、プロセス数の変化による通信回数と、性能低下率の関係について調べたものが図 4 である。これにより、通信回数の増加が性能低下を引き起こしていることがわかる。一方 NPB-MG は、通信回数がプロセス数によって変わらない。しかしながら MG のプロセス数と性能の関係を示した図 5 より、プロセス数増加に伴い性能が低下していることがわかる。つまりプロセス数自体も性能低下を引き起こすことが分かる。この原因を調べるために、Heart Beat 、および send/recv バッファのコピーを無効にしたが、有効時の性能に対しほとんど改善が見られなかった。

6.2 NPB-CG におけるオーバーヘッドの解析

次に、NPB-CG において MPI 関数の呼び出しタイミングを調べた。これには、MPICH に用意されている-mpilog パラメータを用い、同様に MPICH が用意

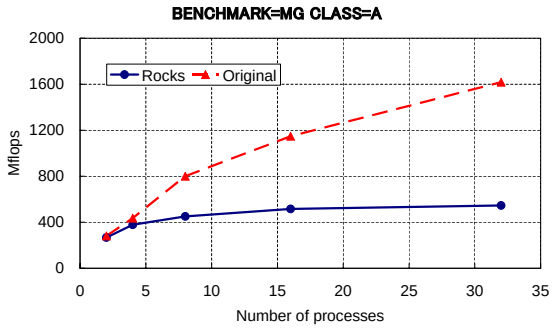

図 5 NPB-MG によるプロセス数とオリジナル MPICH および Rocks+MPICH の性能

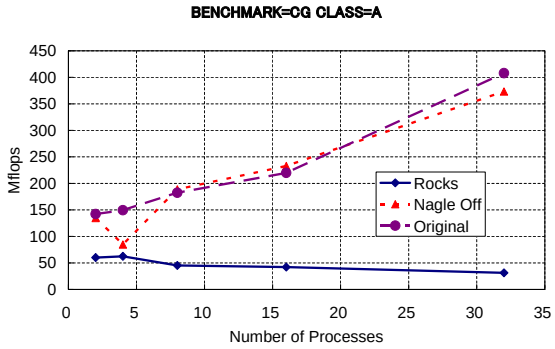

図 6 NPB-CG における Nagle アルゴリズムの有無と性能の関係

している jumpshot[9] で視覚化した。結果、MPI_Wait に多くの時間がかかることが確認できた。これは、以下の理由であると考えられる。

- Rocks がバッファリングなどの処理を行うために、ACK の送受信が遅れ、擬似的に遅延 ACK アルゴリズムと同じ状態になっている
- 遅延 ACK アルゴリズムと Nagle アルゴリズムを同時に用いると、大きな性能低下が起こる

以上を考慮し、オリジナルの Rocks、Nagle アルゴリズムの無効化した Rocks を用いた MPICH と、オリジナルの MPICH で NPB-CG を測定した（図 6）。

結果、Nagle アルゴリズムを無効にしたとき、ほぼオリジナルの MPICH と同程度の性能が得られることがわかる。32 プロセス用いたとき、Nagle アルゴリズムが無効のものはオリジナルのおおよそ 92%の性能となっている。また、プロセス数が増加するに従い、その性能も伸びていることから、スケーラビリティの面においても問題がないと考える。しかしながら、Rocks 自体がいまだ多くのバグを持っているために、現在 NPB-CG 以外のアルゴリズムを安定して動かすことが出来ない。このために、今回のチューニングは NPB-CG にのみ適したものであるという可能性がある。

7. 関 連 研 究

7.1 Cocheck

並列プロセスのチェックポインタの実装例は少ないが、いくつかのモデルが提案されている。Cocheck[7] は Condor[5] のチェックポイントライブラリ[6] を用いて実装されており、典型的な Coordinated Checkpointer である。これは、すべてのプロセスが送信中のメッセージをすべて送信し終わった後にチェックポインティングを行う。各プロセスはチェックポインティングが指示された後に通信が終了すると他のすべてのノードに ready メッセージを送る。すべてのノードの ready メッセージが集まると、チェックポインティング可能な状態であると判断する。この方法では、チェックポインティング時の同期が本研究で提案する方法に比べ大きなものになってしまう。Cocheck は Condor の並列チェックポインタとなるはずであったが、同期のオーバーヘッドが大きいことから現在は使われていない。

7.2 FT-MPI

チェックポインティング／リスタートとはまったく違った方法で耐故障性を実現する既存研究も存在する。FT-MPI[4] がその一つで、 Harness[2] 上に実装されるプロセスレプリケーションに基づく耐故障性 MPI である。プロセスレプリケーションとは同じ状態を持つプロセスの複製 (replica) を作ることである。これを用いることにより障害が発生した時、用意しておいた複製に切替えることにより処理を続けることができる。しかしながら、このアルゴリズムでは冗長なプロセスを作らねばならず、計算資源を有効に活用できない。また、FT-MPI ではコミュニケータの状態とプロセスの状態を拡張し、柔軟なエラー対応が行えるようにしてある。また障害の起こったプロセスも作りなおすことができる。実際の流れは、エラーを検出し、もしエラーが出ていればそのプロセスを作り直すというものになる。そのため、ユーザーコードを改変する必要があり、ユーザー透過であるという条件を実現できない。

8. まとめと今後の課題

本研究では通信路の耐故障性を確保するライブラリとして、Rocks ライブラリを用い、ckpt ライブラリと組み合わせることで、MPI プロセス上にチェックポインティング／リスタート機能を実装した。その上で、評価実験を行い、NPB-CG において 32 プロセス用いてもオリジナル Rocks の 8%という非常に小さいオーバーヘッドで実行可能であることを確認した。

今後の課題としては以下のものがあげられる。

1 ノード 2 プロセスへの対応

現状のシステムでは、2 プロセス以上のマイグレー

ションが行えないため、1 ノードに 1 プロセスしか実行できない。Rocks を改良する、或いは共有メモリのチェックポインティングにより、1 ノードに複数のプロセスが存在してもマイグレーションが行えるようにする。

複数のチェックポインティングアルゴリズムの比較
各デーモンの実装を完成させると共に、Coordinated Checkpointing 以外のアルゴリズムでの実装も行う。

リポジトリの設計
リポジトリが一箇所の場合、チェックポイント時に負荷が集中する上、リポジトリ自身が故障した場合に耐故障性を失う。また現状の NFS ではノード数が増えると劇的に性能が低下する。このため、耐故障性を持ちスケーラビリティのあるリポジトリの設計を行う必要がある。

複数の耐故障性通信路を用いての比較
Rocks 以外の耐故障性通信路を用いて実装評価を行い、システムデザインの耐故障性 MPI への適性を検証していく。

謝　　辞

本研究の一部は科学技術振興事業団・戦略的創造研究「低電力化とモデリング技術によるメガスケールコンピューティング」による。また、実験結果を解析するに当たり、ご協力いただいた Rocks の製作者、Victor C. Zandy 氏に感謝する。

参 考 文 献

1) D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, Vol. 5, No. 3, pp. 63–73, Fall 1991.

2) Beck, Dongarra, fagg, Geist, Gray, Kohl, Migliardi, K.Moore, T.Moore, P.Papadopoulous, S.Scott, and V.Sunderam. *HARNESS: a next generation distributed virtual machine*. Elsevier Science B.V., 1999.

3) E. Elnozahy, D. Johnson, and Y. Wang. A survey of rollback-recovery protocols in message-passing systems, 1996.

4) G. Fagg and a Dongarra. FT-MPI: Faulttolerant mpi,supporting dynamic applications in a dynamic world, 2000. Euro PVM/MPI User's Group Meeting 2000 ,Springer-Verilag, Berlin, Germany, 2000, pp.346-353.

5) Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

6) Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of unix processes in condor distributed processing system. Technical report, University of Wisconsin-Madison Computer Sciences #1346, 4 1997.

7) Jim Pruyne and Miron Livny. Managing Checkpoints for Parallel Programs. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, IPPS'96 Workshop)*, Vol. 1162, pp. 140–154. Springer, 1996.

8) D. L. Russell. State restoration in systems of communicating processes. Technical report, IEEE Transactions on Software Engineering, SF6:2, 3 1980.

9) G. William and E. Lusk. User's guide for mpich, a portable implementation of mpi.

10) VictorC. Zandy. ckpt: A process checkpoint library, 2002. http://www.cs.wisc.edu/žandy/ckpt.

11) Victor C. Zandy. Rocks: Reliable sockets, 2002, 2001. http://www.cs.wisc.edu/žandy/rocks.

12) 高宮安仁, 松岡聡. ユーザー透過な耐故障製を実現する mpi へ向けて. 情報処理学会・電気通信処理学会 並列処理シンポジウム JSPP2002 論文集, pp. 217-224, 2002, 2002.