社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2007－HPC－111 (11)
2007／8／1

MPIプログラムの自律チェックポインティング方式の実現

松岡俊輔[†] 前田哲宏[†,☆]
窪田昌史[†] 北村俊明[†]

大規模な並列計算機やPCクラスタ，グリッドなどのノード数が増加するにつれ，ノードやネットワークの故障に対処することが必要不可欠となってきている．MPIは，それらの並列処理環境上で事実上の標準並列ライブラリとして広く用いられている．そのため，同期チェックポインティングやメッセージロギングの機能を備えた耐故障性MPIが多数提案されているが，それらは大きなオーバヘッドをもたらすという問題がある．本稿ではオーバヘッドの低減を図るプログラム主導の自律チェックポイント方式を提案し，その有効性をNAS Parallel Benchmarksを用いて評価する．

Implementation of Self-Checkpointing for MPI Programs

Shunsuke Matsuoka,[†] Tetsuhiro Maeda,[†,☆] Atsushi Kubota[†]
and Toshiaki Kitamura[†]

With the increase of nodes in parallel computing platforms such as large-scale parallel machines, PC clusters and Grids, it has become necessary to handle frequent failures in node and network. On those platforms, MPI is widely used as a de-facto standard library for parallel computing. Many researches have focused on fault tolerant MPI and proposed fault tolerant message passing protocols based on coordinated checkpointing or message logging, which bring about large overhead in performance. In this report, we propose a program-initiated self-checkpointing which aims to reduce the overhead in performance. We evaluate the effectiveness of the proposed method using the NAS parallel benchmarks.

1. はじめに

近年，パーソナルコンピュータ（PC）の低価格化とパフォーマンスの向上を背景に，多数の計算機をネットワークで接続して構成されるPCクラスタがハイパフォーマンスコンピューティング（HPC）分野などで用いられている．PCクラスタは汎用製品を用いて構成できるため価格性能比に優れ，また計算機ノード数を増減することで利用用途や予算に応じて規模を自由に拡大・縮小できるなどの特徴があり，広く利用されている．

しかし，PCクラスタでは構成要素に汎用部品を多数使用するためシステム全体の信頼性が低い．そのため大規模な計算機クラスタ上で長時間に及ぶ並列プログラムを実行すると，実行中に計算機ノードの故障が発生しプログラムを最後まで実行できなくなることがある．その場合，故障した計算機ノード以外のノードで始めから実行しなおす必要があり，有用な計算が失われてしまう．

そのため，大規模な並列プログラムを長時間に渡り実行するためには並列プログラムに耐故障性を持たせることが重要となる．

MPI(Message Passing Interface)は，分散メモリ環境において並列プログラムを開発するための汎用通信インタフェースである．並列プログラムの開発のためにデファクトスタンダードとして使用されている．MPI標準では，多くのMPI関数の戻り値としてエラーコードを取得することでユーザプログラムに故障を通知することができるようになっている．しかし，どのような故障であるのかといった情報を知ることはできない．また，故障からの復旧方法を提供していない．

並列プログラムに対して耐故障性を付加する方法として，チェックポインティングを用いる方法がある．チェックポインティングは，プロセスの実行時にプロセスのイメージを保存しておき，故障の発生時に利用可能なノードへプロセスを移動させ，チェックポイントからプロセスを再開させるというものである．

プログラム規模の増大や計算機ノード数の増加に伴い大規模並列プログラムの開発はますます困難なものとなってきているため，プログラムに耐故障性を持た

† 広島市立大学
Hiroshima City University
☆ 現在，新日鉄ソリューションズ株式会社
Presently with NS Solutions Corporation

せるために追加のコストを払いたくないという要求がある．そのためチェックポインティングを用いてプログラムに追加の API なしに耐故障性を付加する MPI ライブラリの研究がなされている．チェックポインティングを用いてプログラムに追加の API なしに耐故障性を付加する MPI ライブラリとしては MPICH-V[1] や LAM/MPI[6] などがある．こららのライブラリでは並列プロセスを正しくリスタートするためには全てのプロセス間で通信が一貫した状態で再開されなければならないため，同期チェックポインティングやメッセージロギングを行うことにより通信の一貫性を保障している．この一貫性の保障のため同期のためのオーバヘッドなどが発生する．

FT-MPI[3] は，MPI の仕様を拡張しプロセスに障害が発生した場合の情報が得られるようになっている．ユーザは障害復旧時の処理などを記述することになるが，ユーザの負担は大きいと思われる．ABARIS[5] は，障害発生時の故障検知と復旧方法のコンポーネントを選択可能とする MPI フレームワークである．ユーザはコンポーネントを記述するか，同期チェックポインティングなど適切なコンポーネントが提供された環境で，それらを選択することになる．

本研究では，並列プログラムが処理過程で必要となる同期ポイントに着目し，その同期ポイントでチェックポインティングをユーザプログラム側からリクエストするための関数呼び出すことでチェックポインティング時のオーバヘッドを小さくする自律チェックポインティング方式を提案する．

以下，2 節で我々が前提としている並列プログラミング環境について説明し，3 節で自律チェックポインティング方式の提案とその実現について述べる．4 節で性能評価の結果を示し，最後に 5 節でまとめとする．

2. 耐故障性と非均質性をサポートする並列プログラミング環境

PC クラスタやグリッドの普及に伴い，アーキテクチャや性能が異なる様々な計算機から構成される並列処理環境が一般的になってきている．また，これらの並列処理環境では，専用並列計算機などとは異なり，並列プログラムだけでなく，複数のジョブが CPU 資源を要求して競合しながら処理されることも多いと考えられる．我々は，このような非均質な並列処理環境において，各ノードの性能を引き出しつつ，耐故障性も高めるような，柔軟な並列処理環境を提供することを目指している．

2.1 非均質性への対応

並列処理環境の各計算ノードの処理能力に応じて並列タスクである MPI プロセスを分散させるため，抽象プロセッサという概念を導入する．抽象プロセッサは処理能力が全て等しい仮想的なプロセッサである．

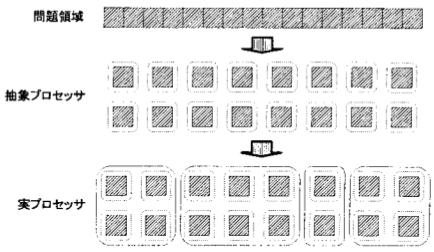

図 1 抽象プロセッサによる問題の非均質分散

各抽象プロセッサには MPI プロセスが 1 つづつ対応するものとする．

計算ノードには，計算ノードの処理能力に応じて複数の抽象プロセッサを割り付けることで負荷の分散を行う．**図 1** に示すように，問題領域 (データや処理) を抽象プロセッサに均等に割り付ける．次に抽象プロセッサを各計算ノードの実プロセッサの処理能力に比例した個数だけ割り付けてやることで，非均質環境におけるタスク分散を実現している．

抽象プロセッサが並列タスクの最小単位であるため，抽象プロセッサの数を変更することで並列処理の粒度を調整することが出来る．

実行時に負荷が動的に変化するような環境においてプログラムの実行を高速化するため，各ノードに割り付ける抽象プロセッサ数を実行時に変化させて動的負荷分散も行う．

我々は，計算ノードの処理能力に応じて，並列タスクである MPI プロセスを分散させるコードを直接記述するのはユーザの負担が大きいと考え，並列化コンパイラによって生成する方式[4] の採用を検討している．抽象プロセッサのノード間の移送 (マイグレーション) は，プロセスマイグレーション，あるいは仮想計算機による移送[7] によって実現する．

2.2 抽象プロセッサを用いた耐故障性

PC クラスタ上で MPI の並列プログラムを実行中に，クラスタ内の一部のノードが故障した場合を考える．このとき，故障していないノードを用いて並列プログラムの実行を続けるには，以下のような方法が考えられる．

- 代替ノードへ MPI プロセスを割り付け直す．
- 故障したノードを除外し残りのノードで縮退実行する．

故障したノードと同じ性能をもつノードを代替ノードとして使用して並列ジョブを再開させるには，故障したノードで実行されていた抽象プロセッサを代替ノードで実行させればよい．

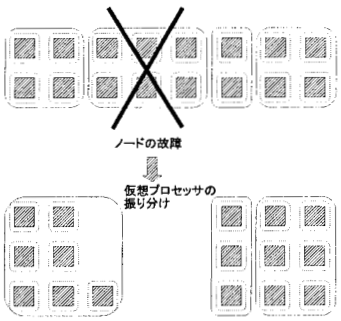

図 2　故障時の縮退実行

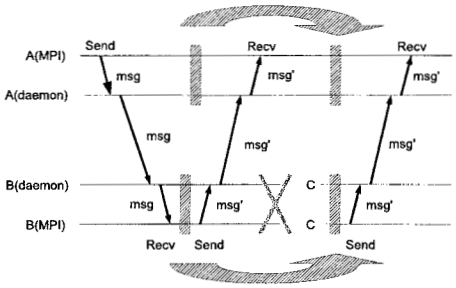

図 3　通信の一貫性が保たれている例

代替ノードがなく故障していない残ったノードで縮退実行をする場合，残ったノードで負荷が適切に分散されるように抽象プロセッサを再割り付けする．**図 2** のように 4 つのノードに多数の仮想プロセッサを割り付けられて並列ジョブが実行中に，1 つのノードが故障して他の 3 つのノードで縮退実行されるとする．16 台の仮想プロセッサを，残りの 3 ノードに，例えば 7:3:6 の割合で再割り付ければ，縮退実行時にもノードの性能に応じて負荷を分散することができる．

3. 自律チェックポインティング方式

本節では，前節で述べた抽象プロセッサの移送によって耐故障性をサポートする並列プログラミング環境を前提とした場合の耐故障性の実現方法について述べ，これに適した自律チェックポインティング方式について提案する．

3.1 チェックポインティングによる耐故障性の実現

並列プログラムに耐故障性を持たせる方法として，チェックポイントベースのロールバックリカバリがある．チェックポイントベースのロールバックリカバリでは，プロセスイメージ（コンテキストやメモリイメージ等）を定期的にディスクに保存（チェックポイント）し，障害発生時は利用可能な計算ノードにおいてイメージの復旧（マイグレーション，リスタート）を行う．

MPI プログラムのような並列プログラムでは，プロセス間で通信を行っている．そのため，並列プログラムをチェックポインティングする場合，プログラムが正しくリスタートされるためには通信の一貫性を保たなければならない．

3.1.1 通信の一貫性の保障

通信の一貫性とは，全ての送信済みのメッセージは全て受信されている，全ての未送信メッセージは全て受信されていない状態を言う．通信の一貫性が保たれない状態でリスタートされた場合，まだ受信されていないメッセージが送信側では送信済みとして扱われメッセージの送信が行われないことや，すでに受信されたメッセージが送信されることが起き，プログラムが正しくリスタートできなくなる．**図 3** において，ノード A,B にそれぞれ計算を行うプロセス（MPI プロセス）と通信のみを担当するデーモン（コミュニケーションデーモン）があり，MPI プロセス間でメッセージのやり取りを行う場合はコミュニケーションデーモンを介して通信を行っている．ただし，すべての MPI 実装においてこのようにコミュニケーションデーモンが必要なわけではなく，実装によって異なる．

図中で，ノード A がメッセージ msg を送信しノード B が受信している．その後，ノード A, B でそれぞれチェックポイントがとられた後，ノード B がメッセージ msg' を送信しノード A が受信している．チェックポイントでの通信の一貫性が保たれているので，故障が発生しチェックポイントからリスタートされると，正しくノード B がメッセージ msg' を送信しノード A が受信する．

図 4 では，チェックポイントからリスタートされるとノード B はメッセージ msg の受信を行おうとするが，ノード A は既にメッセージ msg の送信が完了しているためプログラムは正しくリスタートできない．また，ノード A ではメッセージ msg' の受信を行おうとするが，ノード B はメッセージ msg の受信までブロックされるためメッセージ msg' の送信が行われない．

3.1.2 同期チェックポインティング

同期チェックポインティング (coordinated checkpointing) では，チェックポインティング時に全ての MPI プロセス間で同期し，通信の一貫性が保障された状態でチェックポインティングを行う．

また，故障が発生した場合，故障ノードで実行していたプロセスを利用可能なノードにマイグレーションし，全てのプロセスをチェックポイントからリスタートする．

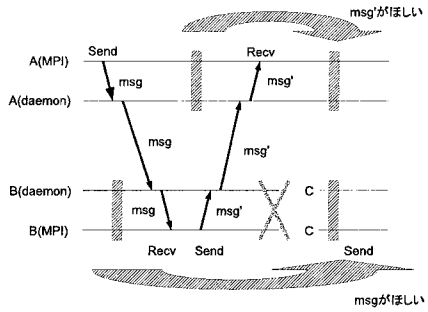

図 4 通信の一貫性が保たれていない例

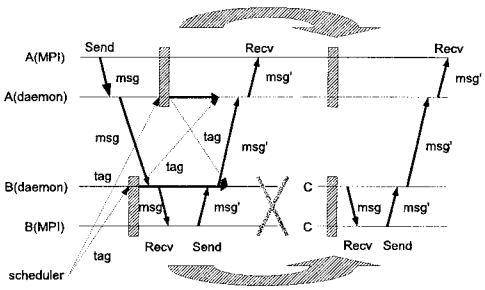

図 5 MPICH-V/cl

同期チェックポインティングでは，チェックポインティング時に同期をとるためにすべてのプロセス間で通信しあうため，プロセス数が多くなると同期コストが大きくなる．

3.2 MPICH-V

MPICH-V[1] は MPICH をベースにチェックポインティングプロトコルを実装した MPI ライブラリである．MPICH-V ではさまざまなプロトコルが実装されているが，ここでは MPICH-V/cl[1),2)] における同期チェックポインティングの処理の流れを**図 5** を用いて説明する．MPICH-V/cl では，MPI プロセスはコミュニケーションデーモンを介して他の MPI プロセスと通信する．チェックポインティングはスケジューラがチェックポイントタグをすべての MPI プロセスのコミュニケーションデーモンに送信することで開始される．コミュニケーションデーモンはチェックポイントタグを受信すると MPI プロセスのチェックポインティングを行い，同時にチェックポインティングを開始したことを他のコミュニケーションデーモンに通知するために自分以外のコミュニケーションデーモンにチェックポイントタグを送信する．他のコミュニケーションデーモンからチェックポイントタグを受信すると，チェックポイントを開始した後に受信したメッセージのうち，チェックポイント開始前に送信を開始したメッセージを保存する．

チェックポインティングによって得たプロセスイメージは，数ノード (デフォルトでは 5 ノード) に 1 つ稼動されるチェックポイントサーバによって収集されるが，同時に各ノードでローカルにも保持される．これは，耐故障のための冗長性を高めるとともに，障害発生時に故障していないノードでの復旧を高速化するために行われる．

障害が発生した場合はすべての MPI プロセスをロールバックし，リスタートする．リスタートされると保存したメッセージを用いて通信の一貫性を保つ．

3.3 自律チェックポインティング

2 節の抽象プロセッサを用いる並列プログラミング環境において，同期チェックポイントによって耐故障性を実現することを考える．同期チェックポインティングでは，チェックポインティング時に同期が必要なためオーバヘッドが大きい．また，故障が発生しチェックポイントからリスタートされた場合でも，リスタートしたことをユーザプログラムは知ることができない．抽象プロセッサのノードへの再割り付けを行えるようにするには，ユーザプログラムにリスタートしたことを通知する必要があると考えられる．

この問題に対して，プログラム中からチェックポインティングを行う関数 `global_ckpt()` を全てのプロセス間で通信の一貫したポイントで呼び出すことにより，同期を行うことなくチェックポインティングを行う方式が本研究の提案手法である**自律チェックポインティング**である．

全 MPI プロセスが同期可能なポイントでは送受信が完了していない通信は存在しないため，自律チェックポインティングが可能である．データ並列性を利用した SPMD(Single Program Multiple Data) 型の MPI プログラムでは，このように，プログラム上で同期可能なポイントは頻繁に現れることが期待できる．

例えば，並列に実行可能な処理をタイムステップを刻んで繰り返すような処理の場合，以下のように数タイムステップに 1 回 `global_ckpt()` を呼び出してチェックポイントを取ることが可能である．

```
for (t=0; t<tmax; t++) {
  // 並列処理
  if (t % 20) global_ckpt();
}
```

自律チェックポインティングを使用するためには，MPI のプログラムを修正して関数 `global_ckpt()` を呼び出すことになるが，並列化コンパイラによりバリア同期可能なポイントを自動的に見つけ出して，自律チェックポインティングおよび抽象プロセッサの再割

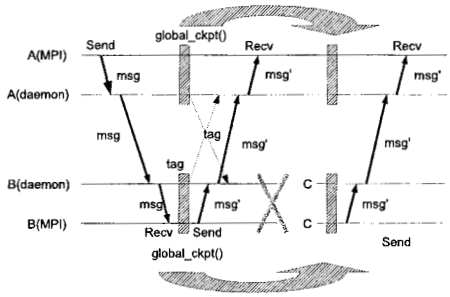

図 6 自律チェックポインティング

```
1: if (myrank == 0)
2:   MPI_Send(...);
3: else if (myrank == 1)
4:   MPI_Recv(...);
5: global_ckpt();
```

図 7 自律チェックポインティング

り付けを行う MPI プログラムを自動生成することで，ユーザの負担を軽減し，また，逐次プログラムレベルでのソースコードの可搬性を保つことができる．

3.4 自律チェックポインティング方式の実現

我々は，提案方式の自律チェックポインティングを MPICH-V/cl をベースにして実装を行った．**図 6** は自律チェックポインティング方式によるチェックポイントと，障害発生時の復旧について示したものである．

MPICH-V/cl では，外部のチェックポイントスケジューラによって，定期的にチェックポイントを行うための制御タグが送信されていたが，本方式では不要である．

チェックポイントからの実行の再開のため，各 MPI プロセスのデーモン同士でチェックポインティング終了の制御タグを通信する．これは，障害発生時にどのチェックポイントまで実行を巻き戻す必要があるのかという情報を各 MPI プロセスが共有するために必要となる．チェックポイントサーバによってスナップショットイメージが収集される機能は，MPICH-V/cl と同様である．

なお，MPICH-V/cl のベースにした場合，コミュニケーションデーモンにおけるメッセージの取り扱いを自律チェックポインティング方式にあわせて変更する必要がある．**図 7** は自律チェックポインティングを行うプログラムの例である．プログラムはランク 0 のプロセスからランク 1 のプロセスへメッセージを送信し，その後チェックポインティングを行うというものである．

MPI_Send() 関数によるメッセージの送信は，MPICH-V/cl では MPI プロセスからコミュニケーションデーモンにメッセージが渡された時点で呼び出しから戻る．そのため，MPI_Send() 関数の呼び出しから戻った時点では，送信先の MPI プロセスでメッセージが受信されているとは限らない．つまり，MPI_Send() 関数の後に global_ckpt() 関数を呼び出し，チェックポインティングを開始した時点では，コミュニケーションデーモン内にメッセージがある場合がある．一方で，MPI_Recv() 関数によるメッセージの受信ではメッセージが受信されるまでブロックされ，受信が完了した後，関数呼び出しから戻る．

そこで，通信の一貫性を保つために MPI プロセスが MPI の通信関数から戻るだけでなく，コミュニケーションデーモン内で保持されているメッセージがすべて送受信完了した時点でチェックポインティングを開始するように，MPICH-V/cl の実装を変更する必要がある．

4. 性能評価

今回実装した自律チェックポインティング方式の MPI ライブラリを用いて，NAS Parallel Benchmarks3.2 の EP クラス B, LU クラス A を実行し，チェックポインティングによるオーバヘッドの評価を行った．比較対象として，MPICH-V/cl や自律チェックポインティング方式のベースとなるオリジナルの MPICH(バージョン 1.2.7p1) と，MPICH-V/cl(MPICH1.2 用) を用いた．

評価に用いた PC クラスタは 8 ノード構成で，各ノードの構成は**表 1** のようになっている．

表 1 PC クラスタの仕様

CPU	Intel Xeon 2.8GHz（ノード当たり 2 台）
主記憶	1GB
ネットワーク	Gigabit Ethernet
OS	Fedora Core 3
コンパイラ	GCC 3.4.2 -O3

評価に用いたベンチマークは，通信量の少ない EP と，通信量の多い LU を用いた．それぞれの実行時間を**表 2**，**表 3** に示す．チェックポイント間隔は 30 秒程度になるようにした．

通信量が少ない EP では，自律方式，MPICH-V/cl とオリジナルの MPICH1 との間の差は小さい．MPI プロセス数が 16 の場合のみ，自律方式と MPICH-V/cl の実行時間が MPICH1 に比べて増加しているが，使用した PC クラスタが 8 ノードのため，この場合のみ 1 ノード内の 2 つの CPU の両方を MPI プロセスに割り付けたためである．

MPICH-V/cl と，それをベースに実装した自律方式では，MPI の本来の計算を実行するプロセス以外にコミュニケーションデーモンやチェックポイントサー

パ，チェックポイント取得時のために MPI プロセスから fork したプロセスなどが必要に応じて生成され，実行される．MPI プロセス数が 1 から 8 までの間は，ノード内で MPI の計算に使用されていない CPU をそれらのプロセスが利用できるが，MPI プロセス数が 16 の場合は，ノード内の 2 つの CPU 資源に対し 2 つの MPI プロセスと上記の各種デーモンなどが競合することになり，MPICH1 に比べ実行時間が増加している．

表 2 EP-B の実行結果 (秒)

プロセス数	1	2	4	8	16
自律	358.72	179.47	90.19	45.10	33.95
MPICH-V/cl	359.03	179.52	90.14	45.12	34.64
MPICH1	359.52	179.79	89.86	45.04	23.33

計算中の通信量が多いアプリケーションである LU では，チェックポインティングに要するオーバヘッドが大きいため，自律方式と MPICH-V/cl では MPICH1 に比べて実行時間が増加している．提案方式の自律方式と MPICH-V/cl を比べると，チェックポイント時に通信されるタグなどの制御メッセージの数が少なくなるため，自律方式の方が実行時間が短縮されている．なお，自律方式では，メモリ容量の制約から，プロセス数 16 の場合の実行が異常終了した．

表 3 LU-A の実行結果 (秒)

プロセス数	1	2	4	8	16
自律	274.40	146.42	74.54	39.93	-
MPICH-V/cl	288.55	155.97	77.95	42.40	58.56
MPICH1	267.64	141.44	72.20	38.24	24.95

これらの結果から，通信量の少ないアプリケーションについてはチェックポインティングによるオーバヘッドは小さいが，通信量が多いとオーバヘッドも大きくなることがわかる．また，MPICH-V/cl に対し，提案手法である自律チェックポインティング方式は，チェックポインティング時のオーバヘッドを低減できることが確認できた．

5. おわりに

本稿では，対故障性を備える MPI の機能について議論し，従来方式のチェックポインティングと比べ，並列プログラムが同期するポイントでチェックポイントを行う，プログラム主導の自律チェックポインティング方式を提案した．本方式を MPICH-V/cl をベースに実装し，NAS Parallel Benchmarks を用いて評価したところ，従来方式よりもオーバヘッドが低減されたことが確認された．

今後の課題は，故障時のプロセス移送，縮退実行などに要するオーバヘッドについても実測に基づき評価を行うこと，自律チェックポインティング方式のプロトコルにさらに検討を加え，制御メッセージなどの削減や，チェックポインティングで生成されるイメージファイルの転送に要するオーバヘッドのさらなる削減を図ることなどである．

謝辞 日頃の研究活動や活発な議論を通して貴重なコメントをいただく広島市立大学コンピュータシステム研究室の諸氏に感謝する．

参考文献

1) Bouteiller, A., Lemarinier, P., Krawezik, G. and Cappello, F.: Coordinated Checkpoint versus Message Log for Fault Tolerant MPI, *Proc. IEEE Int'l Conf. Cluster Computing* (2003).
2) Coti, C., Herault, T., Lemarinier, P., Pilard, L., Rezmerita, A., Rodriguez, E. and Cappello, F.: Blocking vs. Non-Blocking Coordinated Chekcpointing for Large-Scale Fault Tolerant MPI, *Proc. the 2006 ACM/IEEE SC—06 Conf.* (2006).
3) Fagg, G. E. and Dongarra, J. J.: FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world, *Proc. Euro PVM/MPI User's Group Meeting (LNCS Vol.1908)*, pp.346–353 (2000).
4) 後藤慎也，窪田昌史，田中利彦，五島正裕，森眞一郎，中島 浩，富田眞治：並列化コンパイラ TINPAR による非均質計算環境向けコード生成手法，並列処理シンポジウム JSPP，pp.205–212 (1997).
5) 實本英之，遠藤敏夫，松岡 聡：フォールト／リカバリモデルを考慮した耐故障性をもつ MPI フレームワーク ABARIS の提案と評価，技術報告 2007-HPC-109(28)/2007-ARC-172(28)，情報処理学会 (2007).
6) Sankaran, S., Squyres, J. M., Barrett, B., Lumsdaine, A., Duell, J., Hargrove, P. and Roman, E.: The LAM/MPI Checkpint/Restart Framework: System-Initiated Checkpointing, *Proc. LACSI Symp.* (2003).
7) 窪田昌史，前田哲宏，北村俊明：非均質環境向けの仮想計算機を用いた動的タスク分散システムの構築，先進的計算基盤システムシンポジウム (SACSIS) 予稿集，pp.190–191 (2007).