社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

# Speculative チェックポインティングの設計と実装

山形　育平[†]　　實本　英之[†]　　中田　秀基[††,†]　　松岡　　聡[†††,†]

† 東京工業大学　〒 152–8552 東京都目黒区大岡山 2–12–1
†† 産業技術総合研究所　〒 305-8561　茨城県つくば市梅園 1–1–1
††† 国立情報学研究所　〒 101-8430　東京都千代田区一ツ橋 2–1–2
E-mail: †{yamagat1,jitsumo0,matsu}@is.titech.ac.jp, ††hide-nakada@aist.go.jp

**あらまし**　並列プロセスでの同期チェックポインティングでは，チェックポイントを単一レポジトリに保存する時にディスク I/O の負荷が時間的に集中する．本研究ではこれを解決する方法として，インクリメンタルチェックポインティングを改善した投機的チェックポインティングを提案する．これはチェックポインティングの合間にページ更新予測に基づく投機的なチェックポインティングを行うことによりチェックポインティングのタイミングを分割させ，ディスク I/O 負荷を分散させるものである．また，このプロトタイプとして，逐次プロセスの投機チェックポインタを実装した．これを使用し，並列環境での評価を行なった．その結果，投機的チェックポインティングを実装したものは実装していないものと比較して最大 33 %チェックポイント時間の削減効果が観測され，投機的チェックポインティングの有効性が示された．

**キーワード**　同期チェックポインティング,I/O 負荷, 投機的チェックポインティング

# Design and implementation of Speculative Checkpointing

Ikuhei YAMAGATA[†], Hideyuki JITSUMOTO[†], Hidemoto NAKADA[††,†], and Satoshi MATSUOKA[†††,†]

† Tokyo Institute of Technology　2–12–1 Ookayama, Meguro-ku, Tokyo, 152–8552 JAPAN
†† National Institute of Advanced Industrial Science and Technology　1–1–1 Higashi, Tukubashi, Ibaraki, 305–8561 JAPAN
††† National Institute of Informatics　2–1–1 Hitotsubashi, Chiyoda-ku, Tokyo, 101–8430 JAPAN
E-mail: †{yamagat1,jitsumo0,matsu}@is.titech.ac.jp, ††hide-nakada@aist.go.jp

**Abstract**　Checkpointing parallel processes causes high temporal and spatial pressure to I/O subsystems. To decrease the pressure, we propose a new Checkpointing technique, called Speculative Checkpointing, that improves upon incremental checkpointing by speculatively distributing the checkpointing workload and avoiding the necessity of file synchronization. Experimentation with our prototype Speculative Checkpointer on a variety of parallel workload on a cluster showed marked improvements when speculation works effectively, exhbiting up to 33% improvement over conventional incremental checkpointing schemes. We expect that, in a production environment with larger number of nodes and dedicated backend checkpointing storage this improvement would be even higher.

**Key words**　parallel coordinated checkpointing, pressure to I/O, speculative checkpointing

## 1. はじめに

並列プロセスのチェックポイントでは通信の一貫性を保つため，すべてのプロセスが同期してチェックポイントを行う同期チェックポイントというものが一般的である．この同期チェックポイントの場合，生成されたチェックポイントファイルを特定の単一レポジトリに保存することが多い．しかしこの場合，ディスク I/O への負荷が時間的にも空間的にも集中することになる．

これを解決する方法としてインクリメンタルチェックポインティング [1] [2] がある．これは直前のチェックポインティングから更新のあったページだけをチェックポインティングするもので，チェックポイントをとるデータ量を減少させる可能である．しかしこの方法ではディスク I/O の空間的集中の問題は解

決するが，時間的な集中の問題は解決していない．

本研究では，同期チェックポインティングで起こるディスク I/O の時間的集中を解決する，投機的チェックポインティングを提案する．これは各チェックポインティングの合間に投機的なチェックポインティングを行うことにより，チェックポインティングデータの大きさを変えることなくチェックポインティングのタイミングを分割させる手法である．また，プロトタイプである逐次プロセスでの投機チェックポインタを実装し，並列環境での評価を行った．その結果，投機チェックポインティングを行わないもとの比べ，最大 33%のチェックポイント時間の削減効果が観測された

## 2. 投機的チェックポインティング

今回設計する投機的チェックポインティングとは，チェックポイントを取るデータの大きさは変えずに，チェックポインティングを行うタイミングを分割することにより，ディスク I/O の処理を時間的に分散させるものである．

多くの同期チェックポインティングでは，すべてのプロセスが同じタイミングで同じレポジトリにチェックポイントを書き込むため，時間的にも空間的にも I/O 処理が集中する．インクリメンタルチェックポインティングを行うことにより，データ量は減少するが，時間的に集中するという点では解決していない．

この I/O の処理を軽減させる方法として，チェックポイントを分割する方法が考えられる．つまり，規定のタイミングを待たずにチェックポインティングできるものは前もって行っておこうという考えである．これにより，チェックポインティングするデータの量を変えずにタイミングを分けることが可能になり，ディスク I/O 処理を分散させることができる．

### 2.1 チェックポインティング

投機的チェックポインティングでは，インクリメンタルチェックポインティングの合間に投機的なチェックポインティングを行なう．ここでは通常のインクリメンタルチェックポインティングとは違い，

- 前のインクリメンタルチェックポインティングから更新があったページ

かつ

- 投機的チェックポインティングから次のインクリメンタルチェックポインティングの間では更新されないと予測されるページ

を投機的にチェックポインティングする．

予測が成功した場合，このページは次のインクリメンタルチェックポインティングまでは更新されない．したがってこのページは次のインクリメンタルチェックポインティングには含めない．こうすることにより，チェックポイントデータの大きさを変えることなく，チェックポインティングのタイミングを分割することができる．

図 1 が予測成功の場合である．時刻 t1 と t2 の間に page3 の投機チェックポイントがとられたとする．この場合，時刻 t2 では page3 のチェックポイントをとる必要がないため，時刻 t2 の

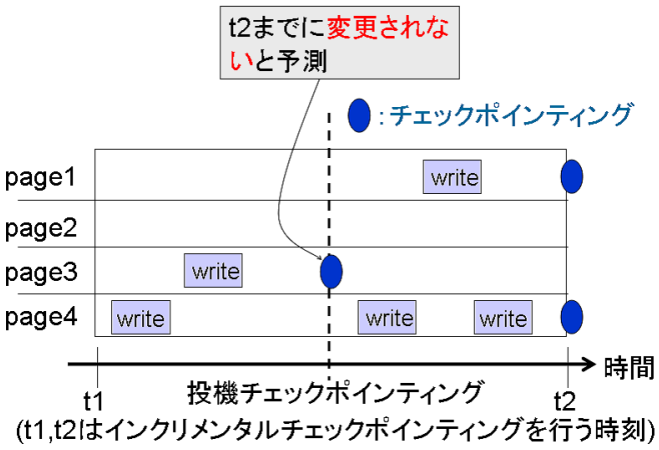


図 1　投機的チェックポインティングの予測成功

Fig. 1　A success scenario of speculative checkpointing

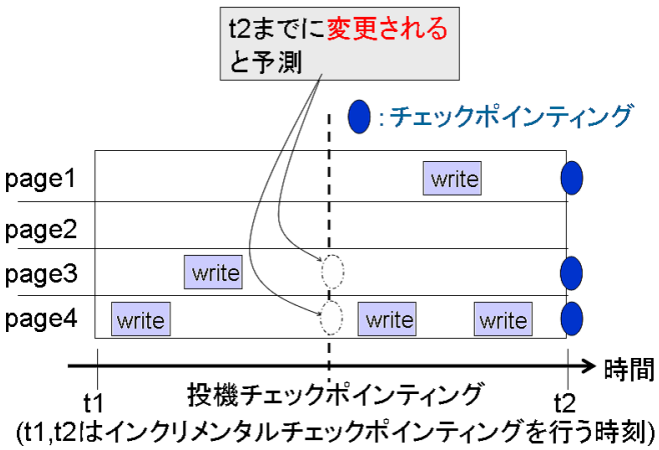


図 2　投機的チェックポインティングの予測失敗 (1)

Fig. 2　The first failure scenario of speculative checkpointing

I/O 負荷が軽減される．

次に予測が失敗したときを考える．予測が失敗する，ということは

（1） 投機チェックポイントを取るべき時に取らない場合

（2） 投機チェックポイントを取るべきでない時に取る場合

の 2 通り考えられる．

(1) は，投機的チェックポインティングの時点で，次のインクリメンタルチェックポインティングまでに更新されると予測してチェックポイントを取らず，実際は更新されなかったという場合である．この場合予測が外れたため次のインクリメンタルチェックポインティングを行う必要がある．しかし，これは投機的チェックポインティングを行わない場合と同じ挙動である．したがって予測失敗によるデメリットは I/O 負荷の分散が達成されなくなるのみである．

図 2 が (1) の場合である．page3 が予測を失敗し，投機的チェックポインティングを行っていない．したがって t2 でチェックポイントをとっている．投機的チェックポインティングを行わないものと比較すると，チェックポイントのデータ量は変わっていない．

(2) は，更新されないと予測して投機的チェックポインティン

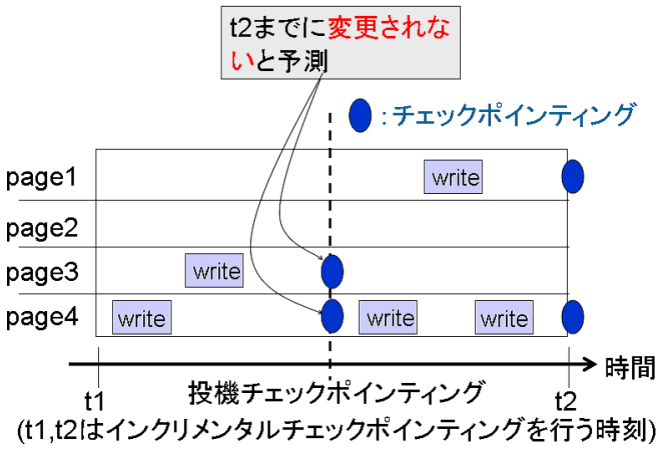


図 3　投機的チェックポインティングの予測失敗 (2)
Fig. 3　The second failure scenario of speculative checkpointing

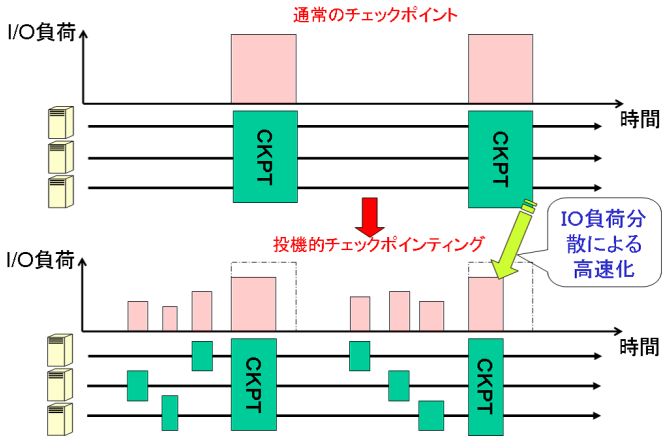


図 4　投機的チェックポインティングを行うことによる変化
Fig. 4　change from speculative checkpointing

グを行ったのだが，実際は更新され，再びチェックポイントを取る場合である．投機的チェックポインティングを行ったページを，再び次のインクリメンタルチェックポインティングで含めることになる．したがって，投機的チェックポインティングを行わないものと比較して，チェックポインティングの回数が 1 回増加する．

図 3 が (2) の場合である．page4 が予測を失敗しているため次の t2 の時点でもチェックポインティングが必要である．そのため，投機的チェックポイントを行わないものと比較してチェックポイントのデータ量が増加する．

したがって投機的チェックポインティングを考える際は両者の投機失敗のペナルティを，それぞれのデメリットにあわせてどのように軽減するか，ということが重要である．

### 2.2　リスタート

投機的チェックポインティングでは，リスタートを行う場合最新のインクリメンタルチェックポインティングのタイミングまでロールバックしたのち，リスタートする．

これは，投機的チェックポインティングは各インクリメンタルチェックポインティングの時刻における I/O の負荷の時間的集中を軽減するための技術であり，リスタートすることを想定して設計していない．そのため図 1 の page4 のように投機的チェックポインティングを行なう時点ですでに更新されているにもかかわらず，チェックポイントが取られないページが存在する．したがって投機チェックポイントからリスタートを行ってしまうとこの更新が失われてしまうためである．

しかし，投機チェックポイントからリスタートできることを保証しないことより，投機的チェックポインティングではフラッシュの必要がないというメリットがある．また，他のプロセスと同期を取る必要もない．したがって，各プロセスが投機的チェックポインティングのタイミングをずらすことが可能になり，チェックポイントを保存する際のディスク I/O の処理を時間的に分散させることができる (図 4 参照)．

## 3.　投機的チェックポインティングの実装

本研究ではこのプロトタイプとして逐次プロセスの投機チェックポインタを実装する．これは既存のインクリメンタルチェックポインタにページ予測アルゴリズムを追加したものである．

### 3.1　インクリメンタルチェックポインティング

投機的チェックポインティングを実装するに際し，ベースとなるチェックポインティングライブラリとして Libckpt [4] を選択した．これは投機的チェックポインティングはインクリメンタルチェックポインティングを利用し，Libckpt がこれを実装しているからである．

Libckpt は Linux と Solaris 用に提供されている逐次プロセス用のチェックポインタであり，C と Fortran のプログラムをチェックポインティングが可能である．

### 3.2　ページ更新予測アルゴリズム

投機的チェックポインティングにおいては，投機的チェックポインティングと次のインクリメンタルチェックポインティングの間に更新されないページを予測することが重要である．

メモリの更新予測アルゴリズムとしてはさまざまなものが考えられる．例として過去のメモリのアクセスパターンから解析するという手法，プログラムの局所性を利用するという手法が考えられる．

現段階での実装では，ある一定期間更新されないページを投機的チェックポインティングの対象としている．

このアルゴリズムを考える際に注意するのは予測失敗時のデメリットである．2 章で説明した通り，予測失敗には

（1）　投機チェックポイントを取るべきときに取らない場合
（2）　投機チェックポイントを取るべきでないときに取る場合

の 2 種類があり，(1) によるデメリットは I/O 負荷の分散がその分達成されなくなるのみであるが，(2) では投機的チェックポインティングを行わないものと比較して，チェックポインティングの回数が一回増加するデメリットがある．したがってアルゴリズムを決定する際には，極力予測失敗 (2) を起こさないものを選ばなければならない．

本研究で実装したアルゴリズムでは，予測失敗 (2) のような

表 1　スペック表

| | |
|---|---|
| ノード | APRRO 1124i |
| CPU | Athlon MP 1900+ × 2 |
| Memory | 768MB DDR (PC2100 256MB × 3) |
| OS | linux kernel 2.4.21 |
| Compiler | gcc v2.95.4 |

事態が起こるには，ある一定期間更新されず，かつ更新される際は連続して更新され，かつそれが投機的チェックポインティングのタイミングをまたがらなければならない．したがって，予測失敗 (2) のような事態は起こりにくいと考えられる．

現在このある一定時間更新されないページというのを，チェックポインティングインターバル数を基準にして実装をしている．このインターバル数は変更可能である．

## 4. 予 備 評 価

### 4.1 逐次プロセスでの評価

まずは本研究で実装した投機アルゴリズムがどのようなプログラムに対して有効であるかを検証する．そのために逐次プロセスの投機チェックポインタを使用して様々なプログラムで投機的チェックポインティングを行い，その結果を評価する．

#### 4.1.1 評 価 環 境

実験は東京工業大学松岡研究室の PrestIII クラスタの 1 ノードを使用して行なった．スペックは表 1 のとおりである．

まず単純なメモリアクセスのプログラムとして，300MB のメモリを順番に変更していくことを 2 周行う MEMWRITE というプログラムを設計し，評価を行った．また，NAS Parallel Benchmark [12] の serial2.3 の BT，CG，EP，LU，MG でも評価を行った．今回は BT の ClassA のみを紹介する．

チェックポインティングのインターバルは次のように設定した (図 5 参照)．

(a)　インクリメンタルチェックポインティング
インターバル 120s

(b)　インクリメンタルチェックポインティング
インターバル 60s

(c)　インクリメンタルチェックポインティング
インターバル 120s ＋投機的チェックポインティング

また以下では，(c$x$) とは投機アルゴリズムを"$x$ チェックポイントインターバル連続で更新されなかったページ "としたチェックポインタのことを示すものとする．

チェックポイントの保存先は

- ローカルディスクのディレクトリ
- すべてのノードに NFS でマウントされたディレクトリ

の 2 つで実験を行なった．

#### 4.1.2 実 験 結 果

まず MEMWRITE について図 6 に結果を示す．(c1〜4) では投機的チェックポインティングされており，特に (c1) に関しては投機的チェックポインティングされたページ数が全体の約 25%になっている．(a) とチェックポインティングの間隔を半分にした (b) とを比較すると，チェックポインティングを行なったページ数に少ししか差がないことから，このプログラムはあるページに関して一度更新されるとある一定時間更新されないプログラムであることがわかる．このようなプログラムにおいては投機アルゴリズムが適当であることがわかる．

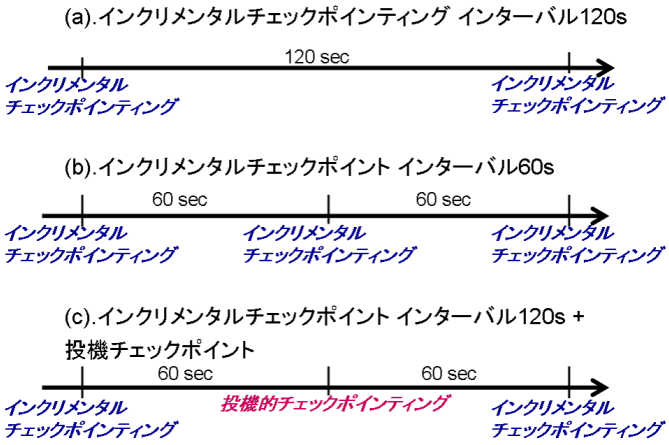


図 5　逐次プロセス実験でのチェックポインティングインターバル
Fig. 5　interval of checkpointing

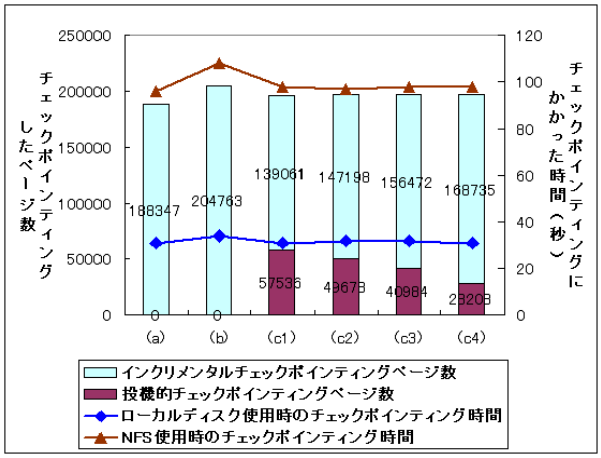


図 6　MEMWRITE での投機的チェックポインティングの有無による結果
Fig. 6　The results of MEMWRITE w/ and w/o speculative checkpointing

次は NAS Parallel Benchmark Serial の BT について結果を示す．測定結果は図 7 のとおりである．(a) と (c1〜4) では実行時間やチェックポイントにかかった時間において，ほとんど差がないことがわかる．また投機的チェックポインティングを行ったページ数が (c1)，(c2) では 27，(c3)，(c4) にいたっては 0 と，非常に小さいことから投機的チェックポインティングをほとんど行っていないことがわかる．ここでは BT を例に出したが，NAS Parallel Benchmark では，すべて同じように投機的チェックポインティングがほとんど行われていなかった．

この原因として考えられるのは，現状の投機ページを予測アルゴリズムの精度が十分でないか，あるいはこのプログラムが常時のメモリ更新を伴い，そもそも本手法にあまり適していないことである．これを検証するのは，詳細なメモリアクセスの

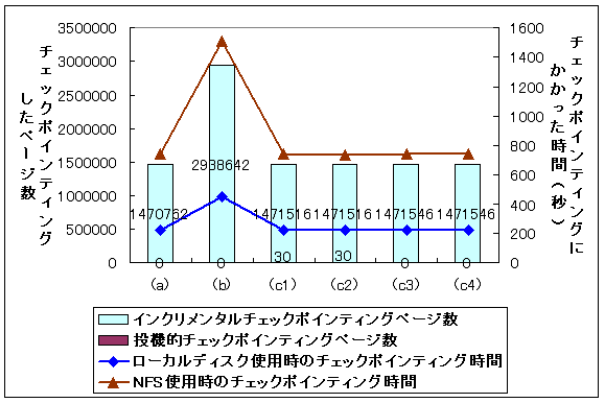


図 7　BT Class A での投機的チェックポインティングの有無による結果

Fig. 7　The results of BT class A w/ and w/o speculative checkpointing


図 8　並列プロセス実験でのチェックポイントインターバル

Fig. 8　configuration of parallel checkpointing interval

トレースが必要であり，今後の課題である．

### 4.2　マルチプロセスでの評価

次に本研究で実装している投機アルゴリズムが有効である場合，並列チェックポインティング時はどのような挙動を示すかを検証する．本来ならば，投機アルゴリズムを実装した，並列同期チェックポインティングで評価を行うべきであるが，その実装が完了していない．そのためそれと同じ効果を持つ状況を擬似的に作り上げ，実験を行った．

#### 4.2.1　評 価 環 境

擬似的な並列環境というのを次のように実装した．

- 複数プロセスで同時に同じプログラムを起動
- インクリメンタルチェックポインティング時はプロセス間で同期
- 投機的チェックポインティングでは同期なし

実験環境は同じく PrestoIII クラスタを使用した．各ノードのスペックは予備実験と同じである (表 1)．また評価プログラムは MEMWRITE で行なった．

チェックポイントはすべてのノードに NFS でマウントされたディレクトリに保存する．これによりチェックポイントを同一レポジトリに保存することを実現し，同期チェックポインティングで起こるディスク I/O の負荷集中を仮想している．

チェックポインティングインターバルは次のように設定した (図 8 参照)．

(I)　インクリメンタルチェックポインティング
インターバル 120 秒

(II)　インクリメンタルチェックポインティング
インターバル 120 秒+投機的チェックポインティング (同時)

(III)　インクリメンタルチェックポインティング
インターバル 120 秒+投機的チェックポインティング (分散)

#### 4.2.2　実 験 結 果

表 2 に結果を示す(注1)．(III) の括弧の中は，投機的チェックポインティングを行ったことによる，チェックポインティング時間の削減された割合である．

表 2　並列プロセス実験での投機チェックポイントの実行結果 [チェックポインティングにかかった時間 (s)/総実行時間 (s)]

| | 1 プロセス | 2 プロセス | 4 プロセス | 8 プロセス | 16 プロセス | 32 プロセス |
|---|---|---|---|---|---|---|
| (I) | 60 | 104 | 188 | 420 | 985 | 2282 |
| | 1004 | 1056 | 1142 | 1405 | 1985 | 3488 |
| (II) | 66 | 110 | 194 | 393 | 855 | 2180 |
| | 1007 | 1062 | 1150 | 1361 | 1854 | 3295 |
| (III) | 66(-10%) | 89(14%) | 138(27%) | 276(33%) | 764(22%) | 2159(5%) |
| | 1007 | 1043 | 1103 | 1255 | 1769 | 3338 |

総チェックポインティング時間を比較すると，複数プロセスで行った場合 (III) は (I) や (II) とくらべて減少していることがわかる．特に 8 プロセス時に最大 33%チェックポインティング時間の削減効果が観測された．これは投機的チェックポインティングによりディスク I/O の処理が時間的に分散したことによるものである．したがって MEMWRITE のような一度更新されるとある一定時間更新されない性質を持つプログラムでは投機的チェックポインティングの有効性があることを確認した．

また，16 プロセス以降はチェックポインティング時間の削減された割合が小さくなっていることがわかる．これは実験のチェックポインティングのインターバルが 120 秒と小さいことから，投機チェックポイントを取る際に十分に分散しなかったことが原因であると考えられ，十分なインターバルを取った環境での実験を行う必要がある．

## 5.　関 連 研 究

並列プロセスでの同期チェックポインタとしては様々なものが存在する．

(注1)：一番最初のインクリメンタルチェックポインティングが終了するまでの時間は除いた．これは一般的なプログラムは最初に初期化を行うため，最初のチェックポイントではほとんどのページをチェックポイントすることになる．そのためその後のチェックポイントと比べて，非常に長い時間チェックポイントにかかり，その結果総チェックポインティングにかかった時間の多くを占めることとなる．よって投機チェックポイントによるチェックポインティングにかかった時間の削減がわかりにくくなってしまうためである．

### 5.1 Cocheck

Cocheck [9] は Condor [13] のチェックポインティングライブラリ [8] を用いて実装されており，典型的な同期チェックポインタである．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．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 Condor に使用するためには大きすぎることから現在は使われていない．Condor は PC の所有者の仕事を阻害しないように，細かい粒度でチェックポインティングを取る必要があったためである．

### 5.2 LAM/MPI

LAM/MPI は MPI の実装の一つで，LAM [10] を使ったものである．この MPI では同期チェックポインティングにより耐故障性を保持している．方法としては，チェックポインティング開始時に同期をとり，通信路を Drain する，Cocheck と同じアルゴリズムである．このチェックポインタは，通信路をマネージメントする crtcp というレイヤと実際に逐次プロセスをチェックポインティングする BLCR [11] というチェックポインタから成っている．BLCR は，カーネルモジュールとして供給されており，読み込みに管理者権限が必要になる．

## 6. おわりに

本研究では並列プロセスでの同期チェックポインティングの際に起こるディスク I/O 処理の時間的な負荷を分散させる投機的チェックポインティングを提案，設計した．またプロトタイプとして逐次プロセスの投機チェックポインタを実装し，並列環境での実験を行った．その結果，特定のプログラムでは最大 33%のチェックポイント時間削減効果が観測され，このアルゴリズムの有効性を確認した．

今後の課題としては，次があげられる．

- より良い更新予測アルゴリズムの探求

現在実装している更新予測アルゴリズムでは，NAS Parallel Benchmark で投機的チェックポインティングの対象となるページを見つけられなかった．より正確な更新予測アルゴリズムを探すことが今後の最も重要な課題である．

- 並列チェックポインティングへの対応

現状では並列チェックポインティングには対応していない．これに完全に対応していくのも今後のもっとも大きな課題のひとつである．

- 実アプリケーションでの有効性の検証

現段階では評価を NAS Parallel Benchmark と MEMWRITE でしか行っていない．この投機的チェックポインティングアルゴリズムが有効であるかを検証するため，様々なアプリケーションで実験していく．

- 投機的チェックポインティングを行なうタイミング

今回の実験では投機的チェックポインティングを静的に決めて行なっていた．今後はどのタイミングで行なっていくのが一番よいか，それを動的に決めるアルゴリズムを探していきたい．

- フォークチェックポイントへの対応

フォークチェックポインティング [5] [6] とはチェックポインティングを行う際，fork(2) を使用することによりメインプロセスとチェックポインティングプロセスを並行して行なう手法である．フォークチェックポインティングにも対応していくすることが今後の課題である．

**謝辞** 本研究は，科学技術振興機構・戦略的創造研究「低消費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．

## 文　献

[1] S.I.Feldman and C.B.Brown.: Igor: A system for program debugging via reversible execution.: ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging, 24(1):112-123, 1989

[2] P.R.Wilson and T.G.Moher.: Demonic memory for process histories.: In SIGPLAN '89 conference on Programming Language Design and Implementation, page 330-343,1989

[3] E.N.Elnozahy, D.B.Johnson and W.Zwaenepoel.: The performance of consistent checkpointing.: 11th Symposium on Reliable Distributed Systems, pages 39-47,1992.

[4] James S.Plank, Micah Beck, Gerry Kingsley, and Kai Li.: Libckpt: Transparent Checkpointing under Unix.: Conference Proceedings, Usenix Winter 1995 Technical Conference, New Orleans, LA, January, 1995

[5] J.Leon, A.L.Fisher and P.Steenkiste.: Fail-safe PVM: A portable package for distributed programming with transparent recovery: Technical Report CMU-CS-93-124, Carnegie Mellon University, 1993.

[6] D.Z.Pan and M.A.Linton.: Supporting reverse executin of parallel programs.: ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging 24(1):124-129, 1989

[7] E.N.Mootaz Elonazahy, Lorenzo Alvisi, Yi-min Wang and David B.Johnson.: A Survey of Rollback-Recovery Protocols in Message-Passing Systems.: ACM Computing Surveys, Vol.34, No.3, September 2002, pp.375-408,2002

[8] Michael Litzkow and Todd Tannenbaum and Jim Basney and Miron Livny.: Checkpoint and Migration of UNIX Processes in Condor Distributed Processing System.: University of Wisconsin-Madison Computer Sciences #1346. April 1997

[9] Jim Pruyne and Miron Livny.: Managing Checkpoints for Parallel Programs.: Job Scheduling Strategies for Parallel Processing, IPPS'96 Workshop. pp.140-154, 1996

[10] Greg Burns and Raja Daoud and James Vaigl.: LAM: An Open Cluster Environment for MPI.: Proceedings of Supercomputing Symposium. 1994 pp 379–386

[11] J.duell, P.Hargrove, and E.Roman.: The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart.: Future Technologies Group white paper, 2003.

[12] NAS Parallel Benchmark. http://www.nas.nasa.gov/Software/NPB/

[13] Condor Project Homepage. http://www.cs.wisc.edu/condor