クラスタにおけるコンシステントチェックポイントの実現

西岡利博[†] 堀敦史[††]
手塚宏史[††] 石川裕[††]

本論文はクラスタのためのコンシステントチェックポイント（CCP）機構について述べるものである．本論文で提案するCCPは，ネットワークプリエンプションと呼ばれる並列プロセスのプロセス切替えの機構をCCPの無矛盾性の保証に応用し，また，CCPのスケーラビリティを確保する目的で，並列プロセスの状態をクラスタの各ノードが有するローカルディスクに出力するようにしたものである．NAS並列ベンチマークプログラムにより評価したところ，通常実行時のオーバヘッドはほとんどないことが確認できた．しかしローカルディスクの性能のバラツキにより，理想的なCCPのスケーラビリティは得られなかった．

A Consistent Checkpointing for Clusters

Toshihiro Nishioka,[†] Atsushi Hori,[††] Hiroshi Tezuka[††]
and Yutaka Ishikawa[††]

This paper describes a consistent checkpointing (CCP) technique for clusters. The proposed CCP uses a process switching mechanism of parallel processes (called network preemption) to ensure the checkpoint consistency. To obtain the scalability, local disk of cluster node is used to save a snapshot of a parallel process. The proposed CCP is implemented and is evaluated using NAS parallel benchmark programs. The evaluation results show that the CCP causes negligible overhead at usual execution. But the ideal scalability of checkpointing cannot be obtained because the I/O performance of local disks are widely scattered.

1. はじめに

クラスタは，商用並列計算機に比べ，構築費用が安価，構築期間が短い，自由度が高い，などといった理由から注目を集めている．しかしクラスタを含む並列計算機は，多数の部品から構築されるために信頼性の面で単体の計算機に劣る．一方，並列処理の代表的なアプリケーションである大規模な科学技術計算では，実行時間が数時間から数日に及ぶことは珍しくない．なんらかの原因によるシステムダウンはそれまでの長時間に及ぶ計算結果を無効にしてしまう．チェックポイントは，計算の途中の状態をディスク等のより安全なストレージ（「ステイブルストレージ」と呼ぶ）に保存し，システムがダウンしたならば直前に保存した状態を復元しそこから計算を再開することにより，不意のシステムダウンの影響を低減する技術である．

クラスタを含む並列計算機では，ある計算を行うために複数のノード上でそれぞれプロセスが並列に実行される．本論文ではこのプロセス群を「並列プロセス」と呼ぶ．この計算の途中の状態を保存/復元するためには，並列プロセスを構成するすべてのプロセスの状態を保存/復元することとなる．しかし各プロセスのチェックポイントには時間的なズレが生じるため，タイミングによってはチェックポイントに矛盾を生じうる．矛盾したチェックポイントとは，文献10)によれば次のように定義される．

> あるメッセージ m があって，m を発信したプロセスが m の発信より前の時点でチェックポイントされ，m を受信したプロセスが m の受信より後の時点でチェックポイントされているとき，そのようなチェックポイントは矛盾している．

すなわちすでに受信されているメッセージがまだ発信されていないチェックポイントは矛盾している．

並列プロセスの無矛盾性を維持した復元が可能な

† エム・アール・アイ システムズ株式会社
M.R.I. Systems, Inc.
†† 新情報処理開発機構つくば研究センタ
Tsukuba Research Center, Real World Computing Partnership

チェックポイントは，コンシステントチェックポイント（以下，CCP）と呼ばれている[3]．CCPでは復帰時に無矛盾性をどのように保証するかが問題であり，このための方式が提案されてきた．これには大きく分けて，チェックポイント時に無矛盾性を保証する方式と，復帰時に失われた無矛盾性を回復する方式とがある．前者を実現する方法として，ネットワーク中にメッセージがない状態を作り出してチェックポイントする方法がある．たとえばsync-and-stop[10]では，並列プロセスを構成するすべてのプロセスの実行を停止し，ネットワーク中の通信メッセージを特別な*marker*メッセージによりflushすることで，チェックポイント時に無矛盾性を保証する．後者の提案としてはたとえば文献9)があげられる．

ネットワークプリエンプションは並列プロセスの切替え手法の一種であり，ギャングスケジューリングと呼ばれるスケジューリング方式で使用される[6),12)]．ギャングスケジューリングとは，並列プロセスを構成するすべてのプロセスをいっせいに停止し，別の並列プロセスを構成するプロセス群にいっせいに切り替えるスケジューリング方式である．ギャングスケジューリングでは，プロセスが同じ並列プロセスに属する他のプロセスと通信しようとするとき，相手のプロセスも高い確率で稼働しているので通信が効率的になると期待されている．ユーザ並列プロセス内での通信はユーザレベルで実現すれば通信効率が向上すると期待されるが，ギャングスケジューリングのもとでこれを実現する場合，ある並列プロセスがネットワークにアクセスした後で入れ替わりにスケジューリングされてきた別の並列プロセスが，直前の並列プロセスが行った操作の影響で，誤ったパケットを受け取ったり，通信エラーを生じたりしないようにしなければならない．ネットワークプリエンプションはこれを実現するものである．ネットワークの状態をネットワークコンテクストと呼ばれるデータで表現し，これをすべてのノードでいっせいに退避/復帰することで，ネットワークの状態を，ある並列プロセスのものから別の並列プロセスのものへと入れ替える．ネットワークプリエンプションによってネットワークの状態を復帰された並列プロセスが正しく動作するためには，メッセージの送受信に関する状態が各ノードで矛盾しない状態でネットワークコンテクストに記録されている必要がある．上述のCCPの無矛盾性に反するような状態のネットワークコンテクストが得られると，その並列プロセスは正しく復帰できない．したがってネットワークプリエンプションは，CCPと同じ意味での無矛盾性を保証する機能を備えている．

チェックポイントでは，ステイブルストレージの選択によって，そのチェックポイント機構により回復が可能な障害の種類が決まる．これまで実装されたCCPの大半は，商用の並列計算機を対象に並列ファイルシステムを用いている（たとえば文献2)，9)，10)）．ファイルサーバに集中的に並列プロセスの状態を保存する方式は，ファイルサーバに障害が生じない限り，あらゆる障害からの回復が可能となる．ステイブルストレージの選択はチェックポイントの性能にも影響を与える．チェックポイントにおいて，その性能がノード数の増加に連れて向上するならば，そのようなチェックポイント方式はよりスケーラブルなチェックポイント方式であるといえる．ファイルサーバに集中的に並列プロセスの状態を保存する方式は，並列計算機（あるいはクラスタ）とファイルサーバを結ぶネットワークのバンド幅によってCCPのバンド幅が抑えられてしまう．

クラスタでは計算ノードにローカルディスクを付属させる構成をとることは難しくない．CCPのステイブルストレージとしてノードのローカルディスクを採用すれば，それらのローカルディスクの障害からの回復は困難となるが，OS，メモリ，プロセッサ等の障害からの回復は可能であり，なおかつステイブルストレージのバンド幅はノード数の増加に比例して増加するので，集中的なファイルサーバを使用するのと比較して，よりスケーラビリティを確保しやすい方式といえる．

2. 本研究の目的と範囲

本論文はクラスタ上における次の方式のCCPを提案するものである．

- 無矛盾性の保証にネットワークプリエンプションを応用する．
- ステイブルストレージに計算ノードのローカルディスクを使用する．

この方式のCCPを実装し，性能を測定した結果を示すことで，この方式のCCPの有用性を客観的に判断できる評価を示す．

ネットワークプリエンプションによるCCPが可能であることを示すことは，ネットワークプリエンプション技術を，ギャングスケジューリングのためのネットワーク切替え機構というだけでなく，ネットワークの状態を無矛盾性を保証しながら保存できる，並列計算環境において有用な汎用的機構としてとらえた，ギャングスケジューリング以外への応用があることを示す

という意味がある．

チェックポイントに関する研究では，保存するデータを圧縮したり，前回のチェックポイントから変わっていない情報は保存しないなどの工夫によって，書き出すデータ量そのものを制限しようという試みも行われている[10]．本論文ではこれらの高速化技法は対象外とする．これらの技法は本論文で提案する CCP にも適用可能である．保存するデータの量には手を加えないことにより，本論文で提案する CCP の基本的な性能を評価する．

チェックポイントの実装という観点では，カーネルレベルで実装するか，ユーザレベルで実装するか（たとえば文献 8)）という違いにより，それぞれ利点欠点がある．ユーザレベルでチェックポイントを実装する場合，システムコールの結果としてカーネル内部に存在するプロセスの状態を完全には復元できなくなる．本論文の CCP はユーザレベルで実装するが，本論文はユーザレベルの実現でカーネルの状態の完全な復元を果たす技法を提案するものではない．大規模並列処理は多くの場合数値計算が主体であり，ユーザレベルで実現した場合に生じる制約が大きな障害となる場合は少ない．一方，ユーザレベルで実装すれば開発期間が短縮され，かつ，異なる OS や異なるプロセッサで構成されるクラスタへの移植が容易になる．これにより，構築期間が短く最新の技術を応用できるというクラスタの利点を活かせることにもなる．

3. CCP 方式

本論文の CCP は，CCP 対象の並列プロセス自身と，それをネットワークプリエンプションによってスケジューリングする別の並列プロセスとによって実行する．この別の並列プロセスを，以下では並列 OS と呼ぶ．

本論文の CCP で保存/復元する「並列プロセスの状態」は，各ノードにおける「プロセスの状態」と「ネットワークの状態」の組がノード数だけ集まった集合と定義する．「プロセスの状態」はプロセスのメモリイメージなどである（詳細は 4 章で述べる）．「ネットワークの状態」はネットワークプリエンプションによって退避されるネットワークコンテクストである．

プロセスのメモリ空間に対するアクセスはそのプロセス自身が行うのが最も効率が良いので，プロセスの状態は CCP 対象の並列プロセス自身が各ノードのローカルディスクに保存する．ネットワークコンテクストは通信ハードウェアの状態を含むものであり，ユーザ並列プロセスが直接アクセスできてはならないため，並列 OS がローカルディスクに保存する．

アプリケーションプログラムは，自身がチェックポイントされるかどうかを意識せずにプログラムできることが望ましい．アプリケーションプログラムが誤って CCP のための通信を受け取ったりその逆が生じたりしないようにするため，CCP のための同期などの通信はすべて並列 OS で行う方針とする．

保存されるプロセスの状態とネットワークコンテクストとが矛盾しないようにするため，ネットワークコンテクストおよびプロセスの状態の保存を行っている間，メッセージの送信を禁止し続けることとする．このため並列 OS は，CCP の開始時と終了時に全ノードの同期をとる．

本論文の CCP では次の手順でチェックポイントを行う．**図 1** の中の番号は次の説明中の番号と対応している

(1) 各ノードにおいて，CCP 対象プロセスが並列 OS に対して CCP 開始の同期を要求する．
(2) 並列 OS は CCP 開始の同期を行う．同期が完了すると，各ノードにおいて，CCP 対象プロセスによるメッセージの送信を禁止し，ネットワークプリエンプションによって CCP 対象プロセスのネットワークコンテクストを得て，ローカルディスクに保存する．その後 CCP 対象プロセスに対して同期の完了を伝える．
(3) 各ノードの CCP 対象プロセスは，並列 OS から CCP 開始の同期の完了を知らされると，そのプロセスの状態をローカルディスクに保存する．保存が終了したら，並列 OS に対して CCP 終了の同期を要求する．
(4) 並列 OS は CCP 終了の同期を行う．同期が完了すると，各ノードにおいて，禁止されていたメッセージの送信を許可し，CCP 対象プロセスに対して同期の完了を伝える．各ノードの CCP 対象プロセスは，並列 OS から CCP 終了の同期の完了を知らされると，チェックポイントを終了し，プロセスの実行を再開させる．

リスタートの手順もチェックポイントのそれとほぼ同様である．プロセスの状態はリスタート対象の並列プロセス自身が復元し，ネットワークコンテクストは並列 OS が復元する．リスタートで必要になる通信はすべて並列 OS が行う．通信が入り混じらないように，リスタートの開始時と終了時に並列 OS が同期を行い，その間，リスタート対象プロセスによるメッセージの送信は禁止される．

図 2 の中の番号は次の説明中の番号と対応している

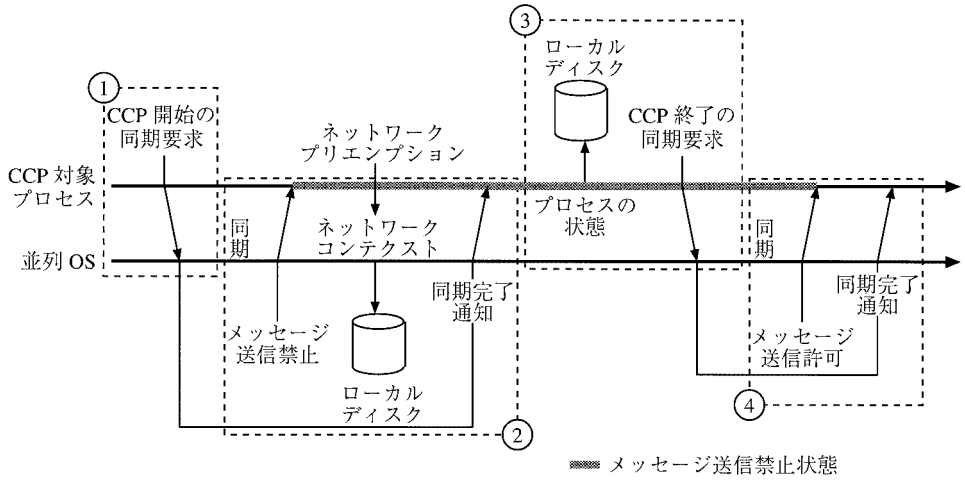

図 1 チェックポイントの手順
Fig. 1 Checkpointing procedure.

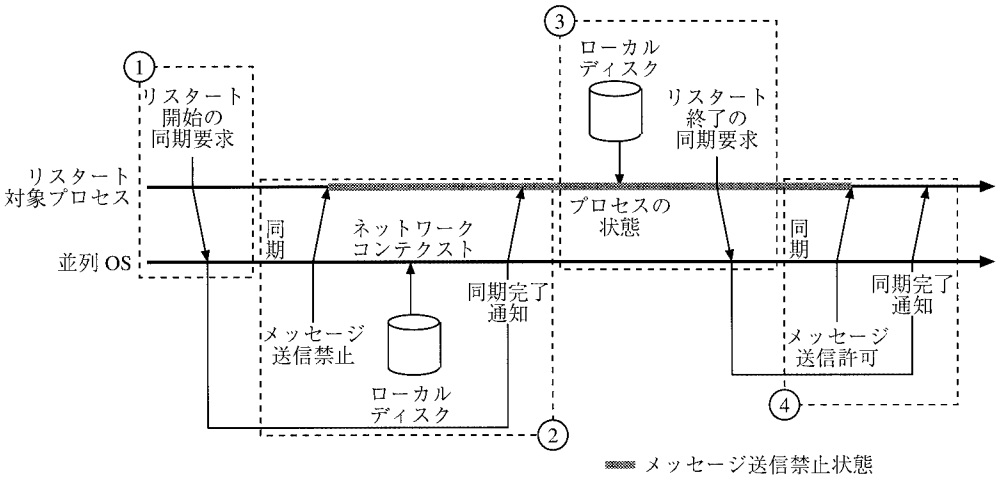

図 2 リスタートの手順
Fig. 2 Restarting procedure.

(1) リスタート対象プロセスを起動する．リスタート対象プロセスは，各ノードにおいて，並列 OS に対してリスタート開始の同期を要求する．

(2) 並列 OS はリスタート開始の同期を行う．同期が完了すると，各ノードにおいて，リスタート対象プロセスによるメッセージの送信を禁止し，ローカルディスクからネットワークコンテクストを読み出して，復元する．その後リスタート対象プロセスに対して同期の完了を伝える．

(3) リスタート対象のプロセスは，並列 OS からリスタート開始の同期の完了を知らされると，ローカルディスクからプロセスの状態を読み出して復元する．復元が完了したら並列 OS に対してリスタート終了の同期を要求する．

(4) 並列 OS はリスタート終了の同期を行う．同期が完了すると，各ノードにおいて，禁止されていたメッセージの送信を許可し，リスタート対象プロセスに対して同期の完了を伝える．各ノードのリスタート対象のプロセスは，並列 OS からリスタート終了の同期の完了を知らされると，リスタートを終了し，プロセスの実行を再開させる．

4. 実　　装

4.1 並列 OS の実装

本 CCP は，並列 OS として SCore-D [5),6),12)] を採用し，3 章で並列 OS が有するとされた機能を SCore-D の上に実装した．SCore-D は，複数の並列プロセ

スをギャングスケジューリングする多重プログラミング環境を実現したクラスタ管理ソフトウェアである．SCore-D は Unix 上にユーザレベルで構築されており，クラスタの各ノードでデーモンプロセスとして実行される．ギャングスケジューリングを実現するために，SCore-D はネットワークプリエンプション機構を有している．SCore-D の，ネットワークプリエンプションを利用した並列プロセス切替え時間は，64 プロセッサの場合に 4 msec 以下である[6),12)]．

SCore-D は，スケジューリング対象の並列プロセス群を管理するための情報をメモリ上に保持している．CCP の際 SCore-D は，この情報をネットワークコンテクストを保存するのと同じタイミングでローカルディスク上に保存し，再起動の際にネットワークコンテクストと同じタイミングで復元する．

4.2 プロセスの状態の実装

プロセスの状態を完全に復元するためには，レジスタなど CPU の状態，プロセスのメモリイメージ，使用中のハードウェアの状態，そのプロセスを管理している OS カーネル内の状態などすべてを復元することが必要である．

本論文の CCP では，これらのうち，ほぼすべての種類のプログラムで復元が必要となる以下の要素を保存する．

- メモリ空間の内容
- CPU の状態
- OS カーネルの状態の一部

これらの保存および復元は，ユーザプログラムにチェックポイントライブラリをリンクさせる形式で実装した．これらの保存に関する具体的な実装は文献 7) に示されたものと基本的に同じである．以下では概略を示す．

プロセスのメモリ空間の範囲を求めるために，ホストの OS が提供する `procfs` を利用した[4)]．チェックポイント時には，まずシステムブレーク値を保存し，続いて，`procfs` から提供されるプロセスのメモリ空間に関する情報を参照しつつ，メモリイメージを保存する．保存するのは，データセグメント，スタックセグメント，および，`brk()` と `mmap()` によって実行中に確保された領域である．テクスト領域は実行可能ファイルを読み込んで復元するものとし，保存しない．リスタートの際は，まずシステムブレーク値を復元し，続いてメモリの内容を復元する．もともと `mmap()` で確保されたメモリ領域の復元は，同じく `mmap()` でメモリ空間を確保してから，その内容を読み込む．

レジスタなど CPU の状態の保存にはシグナルハンドリングの機構を利用する．本方式の CCP では，チェックポイントの機能は `QUIT` シグナルのハンドラとして登録され，`QUIT` シグナルを受信することによって動作を開始する．プロセスがシグナルを受けると CPU の状態がプロセスのメモリ空間の一部に退避されるので，メモリイメージの一部としてこれを保存する．リスタート時には，メモリイメージの復元の処理を行った後，シグナルハンドラから復帰することにより CPU の内容も復元される．

OS カーネル内の情報は一般には保存/復元できない．ただし，シグナルハンドラの設定，ペンディングシグナル，パーシステントなファイルのためのファイルディスクリプタについては，一般的な科学技術計算プログラムにも使用される可能性が高いので，チェックポイント時にシステムコールを使用して OS から状態を得て，プロセスのメモリ空間上に記録することによってメモリイメージの一部として保存し，リスタート時にそれを読み出してシステムコールにより復元する．ファイルディスクリプタに対応するパス名など，CCP の時点では知ることができない情報については，`open()` など，それらの情報を得られるシステムコール関数を入れ替え，それが呼ばれた時点で記録するようにする．この方式は文献 7) に記されたものと基本的に同じである．

5. 評　　価

5.1 評 価 方 針

CCP に関連してアプリケーションが被る可能性のある性能的なデメリットとしては，チェックポイントを行うことによって実行時間が増大することのほかに，チェックポイントの有無にかかわらず，チェックポイントの機構を導入することにより生じる並列処理性能の低下にも気をつけなければならない．

チェックポイントライブラリが存在することで，プロセスのサイズがいくぶん増えたり，チェックポイントライブラリが，`open()` など，いくつかのシステムコールの実行に際して情報の記録を行うことによるオーバヘッドがある可能性がある．またリスタートからの実行では，通常の実行と比較して何らかのオーバヘッドがあるかもしれない．以下ではまず，これらを実際のプログラムで評価する．

これらのオーバヘッドがほとんどないことが示されれば，アプリケーションの実行に際して本 CCP を使用するかどうか，および，どのくらいの頻度で使用するかを評価するには，チェックポイントの性能評価を知ればよいことが分かる．たとえば 10 時間走行する

表 1 RWC PCC-II の仕様
Table 1 RWC PCC-II specifications.

# of Compute Nodes	128
Network	Myrinet
Node Processor	PentiumPro
Chip Set	440FX
Clock [MHz]	200
Memory [MB]	256
Disk [GB]	4
Local OS	Linux

図 3 RWC PCC-II
Fig. 3 RWC PCC-II.

並列アプリケーションを，1 時間ごとに 1 回チェックポイントするとする．チェックポイント機構の導入にともなう通常実行時のオーバヘッドがほとんどないなら，仮に 1 回のチェックポイントに要する時間が 1 時間であったとすると，実際の実行時間は倍の 20 時間になる．ユーザは，許容できるオーバヘッドと事故の場合に失う計算時間を検討し，CCP の頻度を決めることができる．

チェックポイントのスケーラビリティについても評価を行う．ノード数を変化させた場合の，ノードあたりのチェックポイントの性能の変化について評価する．

5.2 評価環境

評価のプラットフォームには RWC PCC-II [11]（以下，PCC）を用いた．**表 1** に PCC の主要な仕様を示す．また PCC の概観を**図 3** に示す☆．

実装された CCP は最大 128 ノードで評価される．文献 2) にも記述されているが，このような大規模なプラットフォームにおける CCP の評価は数少ない．

☆ 文献 11) では 64 プロセッサ構成であったが，現在は 128 プロセッサ構成に拡張され，ローカル OS は Linux に変更されている．SCore-D も Linux に移植された．

表 2 CCP ライブラリの有無による実行時間の違い
Table 2 Elapsed Times with/without the CCP library.

# of nodes	CCP lib.	EP	LU	MG
128	なし	35.86	167.7	7.03
	あり	35.6	169.9	6.86
64	なし	71.72	304.9	12.03
	あり	71.29	306.7	11.88

単位：sec

評価用のプログラムとして NAS 並列ベンチマークプログラム[1]（以下，NPB）を用いた．文献 9) においても CCP の評価用アプリケーションとして NPB が用いられている．プログラムのクラスはすべてクラス B である☆☆．

アプリケーションプログラムの実行はすべて SCore-D により行われる．本論文の計測において SCore-D のギャングスケジューリングは停止（実際には十分長い時分割間隔を指定）している．このため計測結果にはスケジューリングの影響は含まれない．

5.3 通常実行時のオーバヘッドの評価

本節ではチェックポイントしていない状態でのオーバヘッドを評価する．最初に，チェックポイントライブラリがリンクされているときとリンクされていないときの実行時間を比較する．

表 2 は，128 ノードと 64 ノードの場合における EP，LU，MG の実行時間について，CCP ライブラリを付加した場合（CPP lib. “あり”）とそうでない場合（CPP lib. “なし”）の実行時間（単位：秒）を比較したものである．プログラムの実行時間として，NPB プログラム自身が計測した値をそのまま用いている．この計測ではチェックポイントは行われていない．この表からは実行時間に数%の違いが見られるが，測定誤差の範囲と考えられる．この結果から，本論文の CCP では，CCP の機能を追加（ライブラリを付加）することによる実行性能への影響はほとんどないことが確認された．

次に，チェックポイントからリスタートした後の実行がリスタートしない実行に比べて遅くなるかどうかを評価する．**表 3** は NPB の通常実行時の実行時間（NPB プログラム自身が計測した値をそのまま用いた）とチェックポイントからリスタートして実行したときの実行時間を比較したものである．チェックポイントからリスタートして実行したときの実行時間の計測では，NPB が実行時間を計測し始めるより前の時

☆☆ NPB では同じアルゴリズムに対して異なる規模のプログラムが用意されており，これをクラスと呼ぶ．クラス B は，クラス C に次いで 2 番目に大きいクラスである．

表 3 正常実行とリスタート実行の実行時間の違い
Table 3 Elapsed times of normal and restarted execution.

# of nodes	実行	EP	LU	MG
128	正常実行	35.73	168.3	7.00
	リスタート	35.69	170.9	7.16
64	正常実行	71.29	305.1	11.82
	リスタート	71.25	302.1	12.57

128 ノード，単位：sec

表 4 チェックポイント消費時間の内訳
Table 4 Breakdown of checkpoint elapsed time.

	消費時間（sec）			
	EP		LU	
総消費時間	0.340	(100.0%)	1.598	(100.0%)
準備時間	0.023	(6.7%)	0.013	(0.8%)
書込時間	0.219	(64.4%)	1.119	(70.0%)
終了時間	0.098	(28.8%)	0.466	(29.2%)
書込量	1.15 (MB)		4.70 (MB)	

点で 1 度だけチェックポイントした後，プログラムを停止し，その後チェックポイントからリスタートさせたときにプログラムが表示した実行時間を記録した．実行時間に数%の違いが見られるが，測定誤差の範囲と考えられる．この結果から，本論文で提案する CCP では，リスタート後の実行も，通常の実行とほぼ変わらない性能で実行されることが確認された．

5.4 チェックポイントの消費時間の内訳

本節では CCP に消費される時間の内訳とその傾向を示し，消費時間と書き込まれるデータ量との関係を評価する．**表 4** は PCC で EP および LU の 64 ノード（クラス B）のチェックポイントを行った際に，その中のあるノードでその消費時間を計測したものである．

この中で,「総消費時間」はチェックポイントライブラリのシグナルハンドラの中に滞在していた総時間を意味する．以下はその内訳である.「準備時間」は SCore-D が行う次の処理のための時間である．

- すべてのノードで CCP の開始を同期する．
- CCP 対象プロセスについて SCore-D が管理している情報をローカルディスクに保存する．
- CCP 対象プロセスのネットワークコンテクストをローカルディスクに保存する．

「書込時間」はシグナルハンドラが行う処理であり，次を含んでいる．

- `procfs` からプロセスのメモリ空間に関する情報を得る．
- ローカルディスクにメモリイメージを書き込む（バッファのフラッシュ動作も行う）．

「終了時間」は SCore-D が行う次の処理のための時間である．

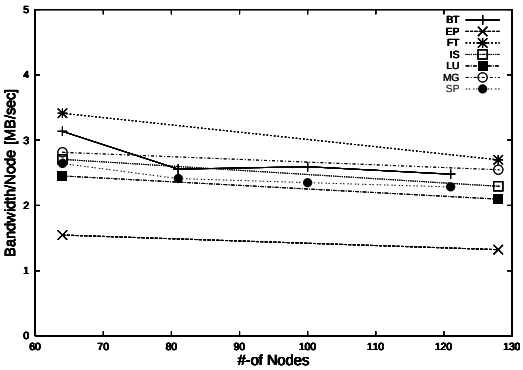

図 4 NPB における CCP のバンド幅
Fig. 4 CCP bandwidth at NPB.

- すべてのノードで CCP の終了を同期する．

各ノードで保存するデータの大きさが異なったり，ローカルディスクの出力性能が異なったりしている場合の処理時間の差は,「終了時間」に含まれる．表 4 の結果は一例であるが,「書込時間」が最も大きな割合を占め,「書込量」が多くなればなるほど増大するという関係は，すべての評価結果に共通している．

5.5 チェックポイントのオーバヘッドの評価

本節では CCP の性能が示す傾向とそのスケーラビリティを評価する．CCP の性能はそのバンド幅によって評価する．CCP のバンド幅 p は次式で定義される．

$$p = (\Sigma_{i=1}^{n} s_i)/t/n$$

s_i はノード i において保存されるプロセスのメモリイメージのサイズである．t はそのチェックポイントが要求されてから完了するまでに消費された時間であり，表 4 の「総消費時間」を指す．

本節の評価には，NPB のプログラムを一定時間ごとにチェックポイントするようにして実行させ，その結果得られたチェックポイントごとの消費時間の内訳，および，メモリイメージを保存したファイル（チェックポイントファイル）の大きさを記録したデータを用いる．

図 4 は，NPB における CCP のバンド幅を同じプログラムでノード数を変えた場合の変化に着目して示したものである．

図 4 では，横軸はノード数であり，縦軸は CCP のバンド幅である．描かれているグラフは，NPB のプログラムをいくつかの異なるノード数で実行し，その際に行ったチェックポイントすべてのバンド幅の平均値をプロットし，同じプログラムでノード数の異なる点どうしをノード数の順に線で結んだものである．プログラムによってプロットされている点の数が異なるが，これは，プログラムのアルゴリズムによって実行

表 5 NPB のチェックポイントファイルの大きさ
Table 5 Checkpoint file size at NPB.

ノード数	EP	FT	IS	LU	MG
64	2,064	47,327	6,348	5,815	6,470
128	2,064	32,136	3,691	4,178	3,961

単位：KB

ノード数	BT	SP
64	20,425	8,879
81	17,818	8,142
100	15,443	7,345
121	13,424	6,578

単位：KB

可能なノード数がそれぞれ異なるためである．すなわち，BT および SP が 2 乗（64，81，100，121），それ以外が 2 べキ（64，128）となっている．

図 4 から，プログラムによって CCP のバンド幅はおよそ 1 ∼ 3.5 MB/sec の範囲に散らばっており一定ではないこと，および，同じプログラムどうしを比較すると，ノード数が増えると CCP のバンド幅がやや低下する傾向があることが読み取れる．特に FT と BT でバンド幅が低下する傾向が大きい．

表 5 は，本節の評価を通じて CCP が作成したチェックポイントファイルの大きさの，ノードあたりの平均値を示したものである．図 4 ではプログラムによって CCP のバンド幅が一定ではないことが示されたが，表 5 と比較すると，おおむねファイルサイズの大きなプログラムほど高いバンド幅を示していることが分かる．これは，チェックポイントの際には，プロセスのメモリイメージをファイルに出力する以外にも，CCP の開始時と終了時に同期のオーバヘッド，および，ネットワークコンテクストと SCore-D が管理する並列プロセスの情報をローカルディスクに保存するオーバヘッドがあるために，ファイルサイズが小さい場合にはその影響が無視できないからと考えられる．

また表 5 から，同じプログラムでもノード数が増えると，保存されるメモリイメージのノードあたりのサイズは減る傾向にあることが分かる．図 4 と表 5 の比較から，保存されるメモリイメージのサイズが減ることは，CCP のバンド幅の低下につながると考えられるので，同じプログラムでもノード数が増えると CCP のバンド幅がやや低下する原因の 1 つであると考えられる．

最もバンド幅の小さい EP と最もバンド幅の大きい FT について，CCP のオーバヘッドがどの程度であるかを**表 6** に示す．ここで，raw はチェックポイントしない場合の実行時間（NPB プログラムが表示した実行時間）であり，5 回の実行の平均である．ckpt はチェックポイントした場合の実行時間であり，3 回実行した中での中央値である．#ckpt は中央値を記録した実行でのチェックポイント回数である．overhead は ckpt − raw である．表 5 から，他のプログラムと比較して FT は実行時間に対するチェックポイントファイルの大きさが特に大きいため，EP と比較して大きなオーバヘッドが生じている．

表 6 CCP によるオーバヘッド
Table 6 CCP overhead.

program	raw	ckpt	#ckpt	overhead
ep.B.64	71.72	78.12	12	6.40
ep.B.128	35.86	47.48	13	11.62
ft.B.64	123.6	220.0	10	96.4
ft.B.128	58.34	149.1	9	90.8

（#ckpt を除き，単位：sec）

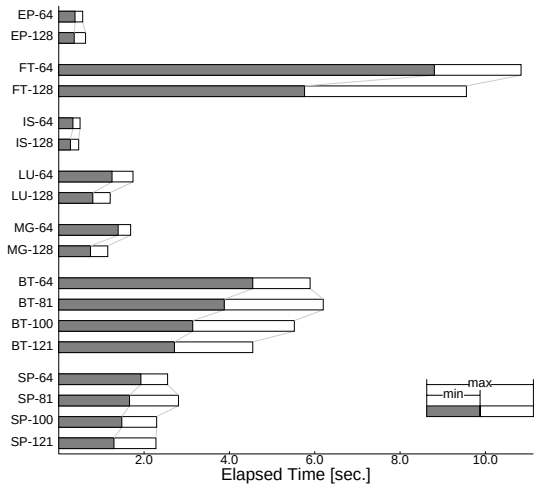

図 5 メモリイメージ保存の消費時間
Fig. 5 Elapsed time to write memory images.

図 5 は，各プログラムごとの「書込時間」（表 4）を表している．プロセスのメモリイメージのサイズは，チェックポイントごとに，および，各ノードごとに異なる．同じプログラムを同じノード数で実行した際に行われたすべての CCP から，CCP の「書込時間」が最も平均的だった CCP を求め，その CCP において，最も「書込時間」の長かったノードにおける「書込時間」と，短かったノードにおける「書込時間」を，図 5 に min と max で示した．「書込時間」が最も平均的だった CCP は次のように求めた．まず全ノードでの「書込時間」を平均した値（「平均書込時間」）を求め，その「平均書込時間」の全 CCP における平均値を計算し，「平均書込時間」がその平均値に最も近い CCP を，「書込時間」が最も平均的だった CCP とした．異

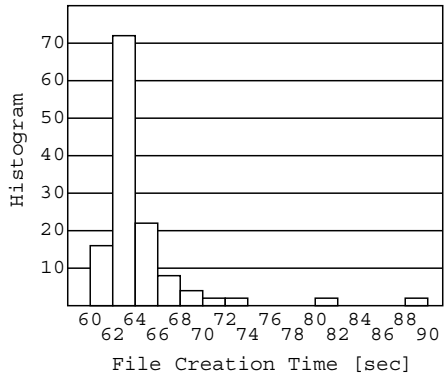

図 6 256 MB のファイル生成時間のヒストグラム
Fig. 6 256 MB file creation time (histogram).

なるノード数で実行された同じプログラムに着目すると，ノード数が増えると，min と max の差は，変わらないか，または拡大する傾向にあることが読み取れる．この傾向は他のノードのチェックポイントが完了するまでの待ち合わせ時間が長くなる方向であり，バンド幅が低下する原因の 1 つとなっていると考えられる．図 4 で，他のプログラムに比較して大きくバンド幅が低下している FT，BT に注目すると，図 5 においても，min と max の差が大きく広がっている．

「書込時間」がノードによってバラつく原因を調べるため，次の評価を行った．**図 6** は PCC の各ノードにおいて，256 MB のファイルをローカルディスクに生成するのに要した時間を計測し，ヒストグラムとして示したものである．PCC ではすべて同じディスク装置を使用しているが，実際の「書込時間」には最大で 50%近いバラツキが見られる．この原因としては，フラグメンテーションの存在や，出力先がたまたまディスクの外周/内周寄りだったために生じたアクセス時間の差異などが考えられる．早いディスクを持つノードが遅いディスクへの保存が終了するのを待ち合わせているために「書込時間」のバラつきが生じると考えると，「書込量」の多い FT や BT でバラつきが顕著になること，および，より多くのノードが使用されるケースでバラつきが大きくなることの説明ができる．このことが，図 5 で見られた，ノード数が増えるにつれてディスクに保存するための時間のバラツキが増える現象の原因の 1 つとなっていると考えられる．

6. 考　　察

6.1 提案方式における制約について

本論文で提案する CCP はユーザレベルで実装している．このため，主に以下の制約が存在する．

アプリケーションプログラムは静的にリンクされていなければならない．動的にリンクされるテクストは，実行ごとに異なったメモリ位置に `mmap()` される可能性があるが，これらは，リスタートの手順の中でメモリイメージが復元されることにより，リスタートの手順が終了した時点ではチェックポイント時点の状態に戻ることになる．しかし，リスタートの手順中に動的にリンクされたテクストが実行されると，まさに実行中のコードの存在するメモリ領域が，チェックポイント時点の内容に復元されてしまう可能性がある．このためチェックポイントライブラリ自身，および，それがリスタート手順で使用する他のコード（本論文の実装では，たとえば `mmap()` など）は，静的にリンクされている必要がある．

パーシステントなファイルのものでない一般のファイルディスクリプタは，OS カーネル内部の状態をリスタート時に復元できない．これにはパイプおよびソケットが含まれる．

6.2 ローカルディスクの障害

ローカルディスクにも故障は生じうる．分散ファイルシステムではサーバのディスクを RAID 構成にすることでディスクの故障に備えることができるが，クラスタのノードに付随するすべてのディスクを RAID 構成にするのはコスト的に負担が大きい．スケーラビリティを損なわずにローカルディスクの故障に対処する方法として，全ノードのチェックポイントファイルのパリティを計算し，それを各ノードが分担して（ただしどの部分も複数のノードで）保持する方式が考えられる．Myrinet のようにディスクよりも高速なネットワークであれば，転送によるバンド幅の低下はある程度避けることができるものと予想される．

7. ま と め

本論文ではクラスタにおけるコンシステントチェックポイントの新しい実装方式を提案した．提案方式は，効率的なギャングスケジューリングの実装技法であるネットワークプリエンプションをコンシステントチェックポイントの実現に応用し，また，スケーラビリティが得られることを期待して，並列プロセスの状態を，クラスタの各ノードが保有するローカルディスクに保存するものであった．

提案した方式を実装し，NAS 並列ベンチマークプログラムを用いて 128 台構成の PC クラスタ上で評価した．実行時間を比較する評価実験の結果，チェックポイントライブラリが追加されたバイナリによる実行時間が，そうでないバイナリによる実行時間とほぼ

変わらないこと，および，チェックポイントからリスタートしたときの実行時間がリスタートではない実行における実行時間とほぼ変わらないことが確認された．

評価環境の PC クラスタにおいては，およそ 1 ～ 3.5 MB/sec のバンド幅が得られた．保存されるデータ量が少ない場合に性能が低いことから，性能が低くなるのは，同期のためのオーバヘッドなどの影響があるためと考えられる．

スケーラビリティについては，同じプログラムでもノード数を増やすと，大きな性能の低下はないものの，やや性能が下がることが確認された．この原因の 1 つには，NAS 並列ベンチマークプログラムには，ノード数を増やすと，ノードあたりのプロセスのメモリイメージのサイズが少なくなるものが多く，メモリイメージのサイズによる CCP の性能の相違から，ノード数が増えることで CCP の性能が低下する点があげられる．また，同じ種類のディスク装置を使用していても，その入出力性能にはバラツキがあることから，ノード数が増えて，より性能の悪いディスクが使われる可能性が増えることで，特に保存するメモリイメージのサイズが大きいプログラムの場合に，CCP のバンド幅が低下することも確認された．

参 考 文 献

1) Bailey, D.H., Barton, J.T., Lasinski, T.A. and Simon, H.D.: The NAS Parallel Benchmarks, NASA Technical Memorandum 103863, NASA Ames Research Center (1993).
2) Chen, Y., Plank, J.S. and Li, K.: CLIP: A Checkpointing Tool For Message-Passing Parallel Programs, *Supercomputing'97* (1997).
3) Elnozahy, E.N., Johnson, D.B. and Zwaenepoel, W.: The Performance of Consistent Checkpointing, *11th Symposium on Reliable Distributed Systems*, pp.39–47 (1992).
4) Faulkner, R. and Gomes, R.: The Process File System and Process Model in UNIX System V, *USENIX Winter'91*, pp.243–252 (1991).
5) Hori, A., Tezuka, H. and Ishikawa, Y.: Global State Detection using Network Preemption, *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, Feitelson, D.G. and Rudolph, L. (Eds.), Lecture Notes in Computer Science, Vol.949, pp.262–276 (1997).
6) Hori, A., Tezuka, H. and Ishikawa, Y.: Highly Efficient Gang Scheduling Implementation, *Supercomputing'98* (1998).
7) Litzkow, M., Tannenbaum, T., Basney, J. and Livny, M.: Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System, Technical Report 1346, University of Wisconsin-Madison Computer Sciences (1997).
8) Litzkow, M.J. and Solomon, M.: Supporting Checkpointing and Process Migration Outside the UNIX Kernel, *USENIX*, San Francisco, CA, pp.283–290 (1992).
9) Naik, V.K., Midkiff, S.P. and Moreira, J.E.: A Checkpointing Strategy for Scalable Recovery on Distributed Parallel Systems, *Supercomputing'97* (1997).
10) Plank, J.S.: Efficient Checkpointing on MIMD Architectures, PhD Thesis, Princeton University (1993).
11) 手塚宏史，堀　敦史，O'Carroll, F.B.，石川　裕：RWC PC Cluster II の構築と性能評価，*HOKKE'98* (1998).
12) 堀　敦史，手塚宏史，石川　裕：ギャングスケジューリングの高速化技法の提案，並列処理シンポジウム JSPP'98，pp.207–214 (1998).

(平成 11 年 12 月 10 日受付)
(平成 12 年 4 月 6 日採録)

西岡　利博（正会員）

1988 年電気通信大学計算機科学科卒業．1990 年同大学院電気通信学研究科情報工学専攻博士前期課程修了．同年（株）三菱総合研究所入社．1998 年エム・アール・アイ システムズ（株）に出向．現在に至る．分散システム，ネットワークセキュリティ等に興味を持つ．

堀　　敦史（正会員）

1979 年早稲田大学電気工学科卒業．1981 年同大学院理工学研究科計測制御工学専攻修士課程修了．同年（株）三菱総合研究所入社．1992 年より技術研究組合新情報処理開発機構に出向．JSPP'98 最優秀論文賞受賞．並列オペレーティングシステムの研究に従事．並列プログラミング言語，並列アーキテクチャ等に興味を持つ．工学博士（東京大学工学部）

手塚 宏史（正会員）

1980 年東京大学教養課程中退．1981 年（株）生活構造研究所入社．1985 年ソニー（株）入社．1988 年（株）ソニーコンピュータサイエンス研究所入社．1990 年ソニー（株）入社．1993 年北陸先端科学技術大学院大学研究生．1995 年より技術研究組合新情報処理開発機構研究員．現在に至る．オペレーティングシステム，リアルタイム処理，マルチメディア処理等に興味を持つ．日本ソフトウェア科学会会員．

石川 裕（正会員）

1987 年慶応義塾大学大学院理工学研究科電気工学専攻博士課程修了．工学博士．同年電子技術総合研究所入所．現在，技術研究組合新情報処理開発機構に出向中．並列・分散システム，適応可能並列プログラミング，言語/環境/処理系，リアルタイム処理等に興味を持つ．日本ソフトウェア科学会，ACM，IEEE 各会員．