ユーザ透過な耐故障性を実現する MPI へ向けて

高宮安仁[†] 松岡聡[†,††]

コモディティクラスタリングシステムにおける，ノード数規模の拡大，計算実行時間およびメモリ空間の急激なスケールアップに伴い，アプリケーションおよびシステムの障害発生の潜在的可能性への対処が急務となっている．しかし，クラスタ等の並列計算分野では，これまでこうした耐故障性についてのソフトウェア開発が重視されておらず，十分ではなかった．また，信頼性, ユーザ透過性, 実行時オーバヘッドの兼ね合いをユーザが指定することのできる，柔軟な耐故障性機構が求められているが，従来のクラスタ向け耐故障性システムでは，単一のポリシ/機構専用のものがほとんどであった．加えて，実アプリケーションを用いた場合のオーバヘッドも明らかではなかった．本稿では，耐故障性機構をもつ MPI である，Parakeet システムを提案する．Parakeet システムを用いることによって，ユーザは性能を損ねることなく，容易に耐故障性，リカバリのポリシ/機構を指定できる．本稿では予備段階として，ユーザレベルチェックポインタ，およびプロセスマイグレーションと Coordinated Checkpointing の一部を MPICH 上にユーザ透過に実装した．予備的な評価の結果，Parakeet システムは移植性を保ちつつ効率的であり，本研究の将来的な目標であるプラグアンドプレイクラスタリングの基礎技術として有用であることがわかった．

Towards MPI with user-transparent fault tolerance

Yasuhito Takamiya[†] and Satoshi Matsuoka[†,††]

Rapid increase in the number of nodes as well as the massive scale of computing in terms of both time and memory space for commodity clustering is mandating the handling the potential failure of applications and system as the norm, while inherent fault tolerance and recovery have not been integral part of software tools being developed for parallel computing on such clusters. Moreover, flexible fault tolerance mechanisms in which the user could manage the balance of reliability vs. transparency vs. execution overhead would be vital, but most previous work on cluster fault tolerance have made available only a single policy and/or mechanism, and moreover, their overhead have not been exactly measured for practical applications. Instead, we propose a new fault tolerant MPI system called Parakeet which allows various fault tolerance and recovery mechanism could be easily specified by the user, while retaining the efficiency. As a preliminary basis, we have implemented a prototype of user-level, coordinated checkpointing and migration protocol on top of MPICH in a user-transparent fashion. By specifying new protocols based on the underlying Parakeet mechanism, one could achieve Plug-and-Play management of large-scale clusters. Preliminary benchmarks show that Parakeet is portable and efficient, and could well serve as a basis for Plug-and-Play clustering.

1. はじめに

数百〜数千ノード超の実用的な大規模 PC クラスタリングシステムの構築へ向け，克服しなければならない問題として，システムの信頼性と容易な利用方法の確立が挙げられる．とくに， 1) 期待されるクラスタリングシステム性能のアプリケーションレベルにおける保証，2) 耐故障性等のシステムの信頼性，3) クラスタリングシステムの運用性，アプリケーションへの適応性，等が挙げられる．こうした一連の問題のうち，要求 1) に関しては，研究レベルでは解決されたと言ってよい．なぜならば,AM[1), Fast-Messages[2) など，ユーザーレベル通信ライブラリの研究によって，ほぼ理想値に近い実効値を得ることができることが示されているからである．しかし，要求 2) 3) については，十分な成果が得られていないのが現状である．

汎用 PC を多数結合するクラスタリングシステムでは，システム全体としての信頼性は単体の PC に比べ指数的に低下する．並列処理の代表的なアプリケーションである, MPI[3) を用いた大規模な科学技術計算では，実行時間が数日から数ヶ月に及ぶことも珍しくない. このため, 不意のシステムダウンの結果, 並列タスク全体の再実行を余儀なくされる, といったことが頻繁に起こる. MPI 標準では，こうした障害へ対処するための手段として, 障害発生時 MPI 関数の返り値

† 東京工業大学 Tokyo Institute of Technology
†† 学術国際情報センター GSIC

としてエラーを返すことを要求している. ユーザーはすべての MPI 関数の返り値をチェックすることにより, エラーハンドリングを行うことができる．しかし, 現実問題として MPI 関数の返り値すべてをチェックし, エラーハンドリング処理を明示的に記述するのは大変コストの高い作業である. また，既存の MPICH や LAM といった MPI の実装では，実行時の性能低下を避けるため，依然としてエラーハンドリングのできない種類の障害が存在する. 加えて，クラスタノード全体のクラッシュ，停電といった大規模な障害に対しては，エラーハンドラによる方法では耐故障性が無い．従来, こうしたエラーハンドラで対処できない種類の障害への耐故障性を得るために, ユーザーは実行中のプロセス状態をファイル等に保存する, チェックポインティング処理を用いる場合がほとんどであった. しかし, 並列タスクでのチェックポインティング処理は, 個々のタスクの通信パタンや通信トポロジーに依存しているため, 並列タスク毎にチェックポインティング処理を書き起こす必要があり, 非常にコストが高かった.

以上より, 要求 2) 3) をみたす並列タスクのための耐故障性機構として，低オーバーヘッドなロールバック/リカバリプロトコルおよびプロセスマイグレーション機構, およびそれらを各アプリケーションへ容易に, ユーザー透過に適用するための機構が今後不可欠であるといえる.

本研究では，MPI との互換性を保ちつつ， MPI 上へユーザ透過に各種チェックポインティングプロトコルやマイグレーションのポリシを適用できる Parakeet システムを提案し，P4 ライブラリへの改造を行う方式，および Rocks[4] とチェックポインタを用いた方式について，予備的な実装を行った．評価では，現在実装が完了している，チェックポインタと Rocks の統合による Parakeet ライブラリの基本性能について，interpositioning されたシステムコールのオーバヘッド, チェックポインティング性能, 及び Application Test として NAS Parallel Benchmarks[5] 実行時のオーバヘッドについて調査した．ただし, 分散チェックポインティング，マイグレーションについては未完成のため，評価を行っていない．評価の結果，Parakeet ライブラリによるオーバヘッドは通信を行う場合でも数 % 以下であり，十分許容できるものであることがわかった．チェックポインティング性能について, プロセスのメモリ使用量とチェックポインティング時間・リスタート時間, チェックポイントサイズがほぼ正比例し, チェックポインタとして理想的な振舞いを見せることがわかった. また, 今回開発したチェックポインタはダイナミックリンク可能となっており, 既存のアプリケーションへ容易に適用可能であることがわかった.

2. 背　　景

MPI 上で耐故障性を実現する手法として，主に 1) チェックポインティングを用いた手法と 2) レプリケーションを用いた手法, の 2 種類が提案されている.

チェックポインティングを用いた手法[6)~8)] としては, 既存の MPI を改造し，並列タスク全体の無矛盾なスナップショットを保存する機構を提供するものや[7], クラスタ構成の変化イベントに対するリスナをユーザーが定義し，イベントに応じてユーザーが個々のプロセスのチェックポイントタイミングを明示的に指定できるように MPI に独自の拡張を行ったもの[6] がある. これらのシステムでは，チェックポインティング，もしくはマイグレーション一方のみを実現している場合がほとんどである．採用されているチェックポインティングプロトコルとしては Coordinated Checkpointing が主流であり，筆者らの調べた範囲では, 実システムにその他のプロトコルを実装した例は見当たらない.

一方，レプリケーションを用いた手法[9),10)] では，コミュニケータの不整合の検出，回復のための API を提供するもの[9] や，`MPI_Send()` 等の引数として，レプリカプロセスを指定し，障害時には自動的にレプリカプロセスが実行を続行するシステム[10] がある．これらの手法では，レプリケーション対象プロセスや, 障害からの復帰方法等を明示的にユーザが指定する必要があるので，既存のプログラムを大幅に変更する必要がある.

両者共通の問題として，Independent Checkpointing 時に起こりうる Dommino Effect の可能性の回避や，ロールバックのためのメッセージ間の依存情報追跡，および Log-Based Rollback Recovery のロールバック処理などをユーザが明示的に指定するのはコストが高く，各種プロトコルの深い知識が必要とされることがある．また，一般に, 各アプリケーションに対してどのプロトコルが有効であるかが不明であるため, 実際は複数のプロトコルの実装，比較が必要である点, また，すべてのプロトコルを表現可能な汎用チェックポインティング/ ロールバック API を定めることが困難であるといった問題点がある.

3. 分散チェックポインティング

この章では，メッセージパッシングプログラミングモデルにおける，様々な ロールバック/リカバリプロトコルを紹介し，比較する．この分野については，すでに理論面から数多くの研究がなされている[11]．並列タスクのような，通信を行う複数プロセスからなる系の状態は，各プロセス自体の状態 (volatile checkpoint) と，通信チャンネルの状態からなる．分散チェックポインティングとは，各プロセスの状態と，通信チャンネルの状態を Stable Storage へ保存し，障害時にこ

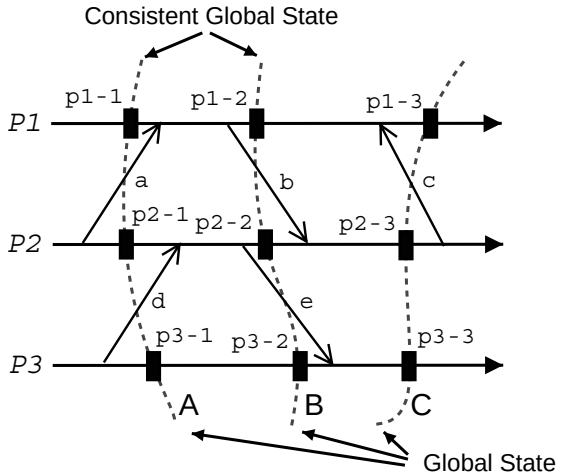


図 1 Consistent Global State

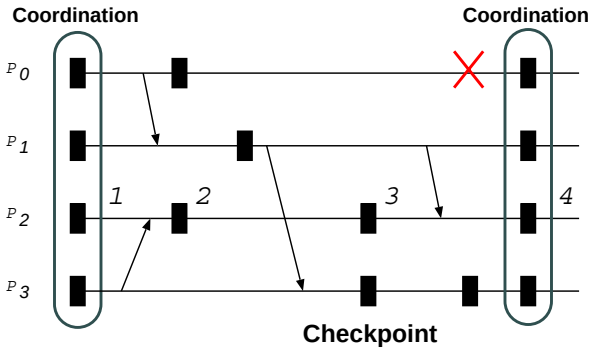
れを復元し, リスタートする技術である．Chandy と Lamport によって，無矛盾な ロールバック/リカバリを満たすための条件が示されている[12]．たとえば, 図 1 において，プロセス P1 の状態 p1-1,P2 の状態 p2-1, P3 の状態 p3-1, 加えて通信チャンネル a, d の状態を保存した Global State A は無矛盾にリスタートできる (Consistent Global State)．しかし，Global State C は無矛盾にリスタートできない．これは，Global State C からのリスタート時に，通信チャンネル C 上のメッセージが複製されてしまうからである．

メッセージパッシングプログラミングモデルにおけるロールバック/リカバリプロトコルは，以下の 2 種類に分類できる．

Checkpoint-based Rollback-Recovery はプロセスの復帰の際に，チェックポイントを用いて復帰するプロトコルである．上記 Consistent Global State を達成するための，Coordinated Checkpointing や，各プロセスが独立にチェックポインティングを行い，障害時には Dependency Tracking を行いロールバックする Independent Checkpointing, 通信パタンによってチェックポインティングを決める Communication-Induced Checkpointing がある．

Log-based Rollback-Recovery は復帰の際に，チェックポイントに加え，非決定性イベントログを用いて復帰するプロトコルである．これらは，PWD[13] を前提としており，system call や外部との I/O 等，非決定性イベントをログに保存/リプレイすることによって，最も最近のチェックポイント以降の状態までを復帰させることが可能である．主なプロトコルとして，ロールバックの局所化を狙った Pessimistic Logging, ノンブロックロギングを行う Optimistic Logging, また，両者の利点を兼ね備えるが，ロールバック等がより複雑な Causal Logging 等がある．

このほか，両プロトコル共通の最適化として,Coordination 時に同期に参加するプロセスを最小限にする手法，およびノンブロック Coordination, 不要なチェックポイントファイルを捨てるためのチェックポイント GC などがある．




```
pmpi-run -np 4 -ckpt_proto lazy interval 60
                -lazyness=4 -with-nonblockng ./foo
```

図 2 `mpirun` の引数による rollback-recovery protocol 指定．プロトコルとして Lazy Coordination, Coordination の間隔として laziness=4, Coordination の最適化として，nonblockng Coordination を指定.

4. Parakeet システム

4.1 Parakeet システム概要

本稿で提案する Parakeet システムは，ユーザ透過なチェックポインティング，マイグレーションを用いて，MPI-1.2 プログラムへ耐故障性を提供するシステムである．Parakeet システムは，ロールバック/リカバリ・チェックポインティング・マイグレーションの各機構を提供する Parakeet ライブラリと，各ノード上で動作し, 実行時のプロセス管理やモニタリング, チェックポインティング起動のイベント送信を行う Parakeet デーモンから構成される．

ロールバック/リカバリ機構は，ユーザーの書いた既存の MPI プログラム上に, 自動的なチェックポインティング・ロールバック・障害時のマイグレーション機構を提供する. ユーザーは，各プロトコルを明示的に実装するかわりに, プログラムを Parakeet ライブラリとダイナミックリンクすることにより, プロトコルを決定する各種パラメータを rc ファイルや `mpirun` の引数として指定できる (図 2)．指定できるパラメータには，プロトコルの種類，Coordination 間隔，チェックポインティング頻度などがある．選択するプロトコルの有効性は, アプリケーションや環境に依存するため，ユーザーはまず複数のプロトコルについてテスト実行し，テスト実行の結果により, チューニングを行ったり, 使用するプロトコルを別のプロトコルへ切り替えることが可能となっている．このため，1) プログラ

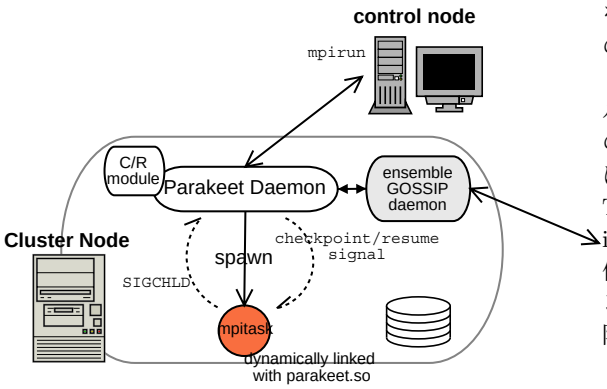


図 3 Parakeet システムアーキテクチャ

ムのコーディング, 2) 複数プロトコルによるテスト実行, チューニング, 3) 実際に用いるプロトコルの選択, 4) 本番のクラスタ上での実行, という行程を容易に行うことができる.

実行時のチェックポインティング/ロールバックタイミングは, ユーザによって指定されたパラメータにより, 自動的に決定される. プログラムにリンクされた Parakeet ライブラリは通信パタンを追跡し,Parakeet デーモンは得られた通信パタンをもとに, 各プロセスへチェックポイント起動シグナルを送信する. このため, ユーザによるソースコード中への明示的なプロトコル実装の挿入は不要であり, 各プロトコルの実装の詳細をユーザーから隠蔽している.

チェックポインティング/マイグレーションは, Parakeet ライブラリ側でシステムコールを interpositioning することで実現されている.Parakeet ライブラリは共有ライブラリとして提供されており (`libparakeet.so`), 既存の MPI プログラムを実行時にダイナミックリンクすることにより, Parakeet システムの提供するフォールトトレランスを利用できる. チェックポインティング等の実装はすべてユーザレベルで行われているため, ユーザーはクラスタで利用する OS を変更せずに, 容易に耐故障性を得ることができる.

4.2 実　　装

Parakeet システムのアーキテクチャを図 3 に示す. 各ノード上では Parakeet デーモンが動作しており, MPI プロセスの生成, モニタリング, チェックポインティングシグナルの送信を行う. プロセスのクラッシュ, Coordination 開始, チェックポイント GC 開始といった各ノード上のイベントの通知には Ensemble GOSSIP デーモン[14] による, アトミックなグループ通信を行う. 各ノードで起動される MPI プロセスは起動時に, チェックポインティングライブラリ (`libparakeet.so`) とダイナミックリンクされる.

今回, MPI 上へのチェックポインティング, マイグレーションの実装として, Rocks (Reliable Sockets)[4] を用いた方法と, p4 ライブラリへの改造を行う方法の 2 種類を実装した.

Rocks は, `accept()`, `write()` 等のシステムコールを interpositioning することにより, IP アドレスの変化, 物理層の切断といった障害後も, ユーザ透過に復帰できる仕組みを提供している. 実装としては, TCP/IP スタック中の Send/Receive バッファの他に, in-flight なメッセージを保存するバッファを設け, 送信側/受信側で送信/受信したパケット数をカウンティングすることによって, in-flight メッセージの損失を防いでいる.

今回の Rocks を用いた実装では, チェックポインタと Reliable Sockets を改造し, 双方が interpoisitioning しているシステムコールの部分を統合した. まず,MPI タスク起動部分について, 各ホストで動作する MPI プロセスのチェックポインティングおよびネットワーク障害への耐故障性を得るために次の改造を行った. 通常の MPI タスク起動では,`mpirun` 起動ホスト上のマスタプロセスから `rsh` 経由で各ホストに`-p4slave` オプション付きの同一のプロセスが起動される (図 4 (a)).Parakeet での MPI タスク起動では, すべての MPI プロセスは `ssh` 経由でチェックポインティング監視プロセス `pkckpt` の子プロセスとして fork され, チェックポインティングライブラリおよび Rocks ライブラリがダイナミックリンクされ, チェックポイントシグナルハンドラが登録される (図 4 (b)). この際, マスタプロセスと子プロセス間の `ssh` は `rockd` 経由での Rocks ライブラリを用いた通信が行われるので, ネットワーク障害への耐性がある.

各 MPI プロセス間の通信の耐故障性について, 通常の MPI では OS によって割当てられるランダムなポート番号が使われるために,Rocks ライブラリを通信に用いることができない. そこで,Parakeet ではたとえば 8888 などのあらかじめ決めておいた固定ポート番号を通信に用いることによって,Rocks の制限を回避している.

Coordinated Checkpointing 要求の際には, 各プロセスについてチェックポインティング終了したプロセスから順次実行をストップする. このことによって, チェックポイント後のプロセスから他プロセスへのメッセージ送信が起こらないため, Global State を逆向きにカットするメッセージが発生せず, Consistent Global State が実現できる. 全プロセスがストップ後, Parakeet デーモンから `SIGCONT` が送信され, リスタートする. チャンネル中の in-flight メッセージの再現については, Reliable Sockets のバッファ中のメッセージがチェックポインティングライブラリによってバッファごとチェックポイントされ, リスタート後に再送される.

p4 への改造を行う実装については, CoCheck[7], Hector[15] が行っているように, p4 へチャンネルの

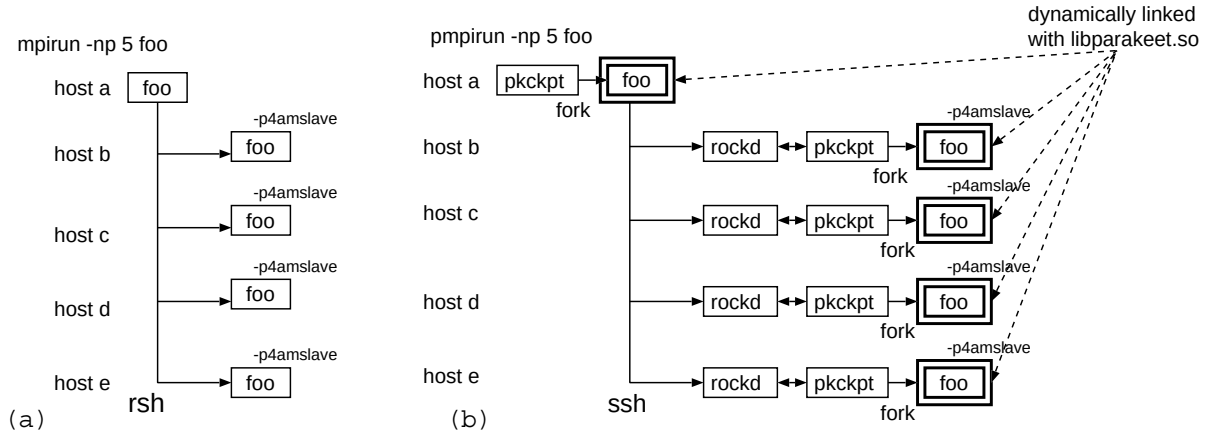


図 4 MPI タスクの起動 (a) 通常の MPI タスクの起動. (b) Parakeet での MPI タスクの起動.

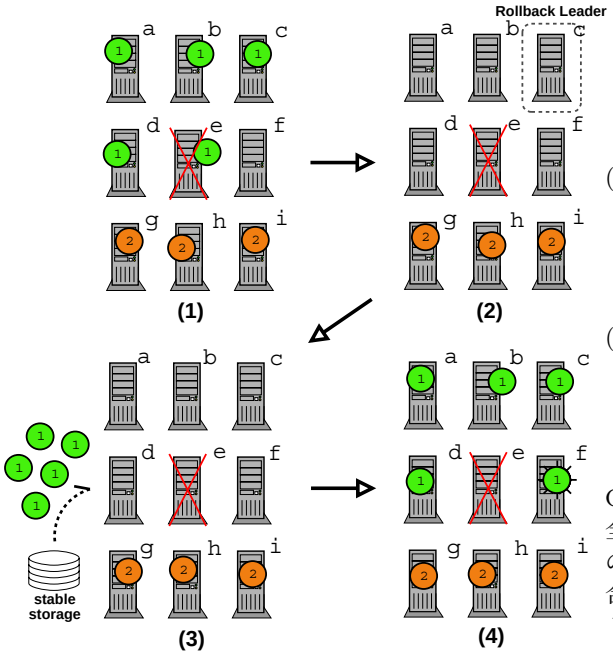


図 5 fail 時の rollback 機構

flush 機能等を追加した. ただし, 現段階では実装が完全ではないため, 評価は行っていない.

4.3 ユーザ透過なロールバック/マイグレーション

ユーザー透過なロールバック/マイグレーションのプロトコルは次のとおりである (Coordination を行うプロトコルの場合). ただし, この部分はまだ部分的な実装しか行っていない.

(1) 図 5 (1) において, ノード a, b, c, d, e では並列タスク 1 を, ノード g, h, i では並列タスク 2 を実行しているとする. ここで, ノード e が突然クラッシュすると, MPI タスク全体が異常終了する.

(2) 生き残っているノード上の `Parakeet daemon` は `SIGCHLD` を受けとり, 生き残った `Parakeet daemon` 間で Leader Election を行う. Leader となったノード上の `Parakeet daemon` は, rollback メッセージを他の `Parakeet daemon` へブロードキャストする (図 5 (2)).

(3) Stable Storage ☆ に保存されている前回 Coordinated Checkpoint 時のイメージを, 対応する各ノードに戻し, クラッシュしたノードで動作していたプロセスイメージは, 他の健全なノードへマイグレーションする (図 5 (3)).

(4) マイグレーションしたプロセス以外のプロセスは, チェックポインティングされた状態からのリカバリ時, マイグレーションを行ったプロセスとのチャンネルについて, ソケットディスクリプタを更新し, リスタートする.

Coordinated Checkpointing を行う場合, 並列タスク全体がロールバックすることに注意されたい. この例の場合では, Coordination 間隔が 1 時間であった場合, ロールバック時には 1 時間前の状態からリスタートすることになる.

チェックポインティングライブラリはチェックポインティング時にソケットをクローズ, リスタート時にコネクションを張り直す機構になっている. (4) でのソケットディスクリプタの更新は, この時に Parakeet デーモンへマイグレーションプロセスのマイグレーション先を問い合わせ, 書き換えを行っている. Reliable Sockets による実装ではコネクションの uniformity を提供しているので, この操作は行われていないが, Rocks の制限により, 同時にマイグレーションできるプロセスの数は 1 つという制限がある.

☆ 現在の実装では, Stable Storage は各ノードのローカルディスクとなっている. ノードクラッシュ等の障害からすみやかに回復するためには, Stable Storage を専用のチェックポイントサーバ上に設けるか[7], 近隣ノードのディスクもしくはメインメモリ上に設ける必要がある.

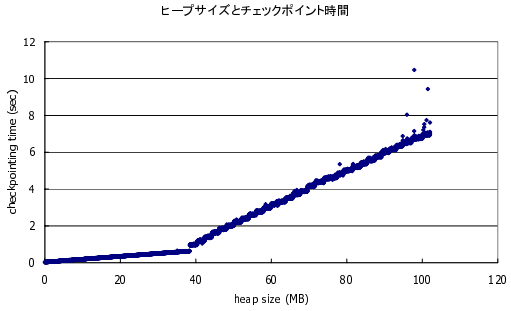


図 6 ヒープサイズとチェックポイント時間

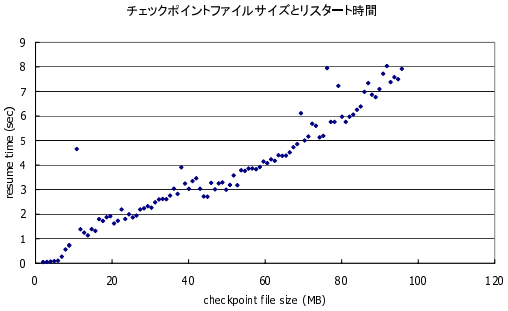


図 7 チェックポイントファイルサイズとリスタート時間

5. 実　　験

実験に用いる環境として，東京工業大学松岡研究室の ion クラスタ[16] (CPU: Mobile Celeron 600MHz, Memory: 128MB, HDD: 20GB, OS: Linux 2.2.17 with PAPI 1.1.5 × 16 nodes, Network: 100base-T switching hub) を用いた．計時関数として，PAPI[17] の `PAPI_get_real_usec(3)` を用いた．実験では，Parakeet ライブラリの基本性能と，実アプリケーションを用いた性能の評価を行った．計測は各 20 回行い、各評価値の最良値を用いた．なお，マイグレーション，Coordinated Checkpointing については，実装が完了していないため，評価を行っていない．

5.1 Kernel Tests

Parakeet ライブラリの基本性能評価として，

- ヒープサイズとチェックポイントファイルサイズ (図 8)
- `sbrk()` を連続して行うプログラムについて，ヒープサイズとチェックポインティング時間 (図 6)
- チェックポイントファイルサイズとリスタートに要する時間 (図 7)
- netpipe-2.4 を用いた，Parakeet ライブラリリンク時のスループット (図 9)
- 逐次版 NAS Parallel benchmarks のチェックポイントライブラリのリンクによるオーバーヘッド (図 10)

について計測した．

図 6, 7, 8 に見られるように，ヒープサイズ/チェックポイント時間，ヒープサイズ/チェックポイントファイルサイズ, およびチェックポイントファイルサイズ/リスタート時間はほぼ比例している．図 9 について，どのブロックサイズでもスループットはリンク有/無時でほとんど変わらないことがわかり，Rocks によるオーバーヘッドが低いことがわかる．図 10 では，いくつかのベンチマークについて，リンク無時の性能よりもリンク後の方が性能が良い．これは，interpoisitioning 対象のプログラムで実行されるシステムコー

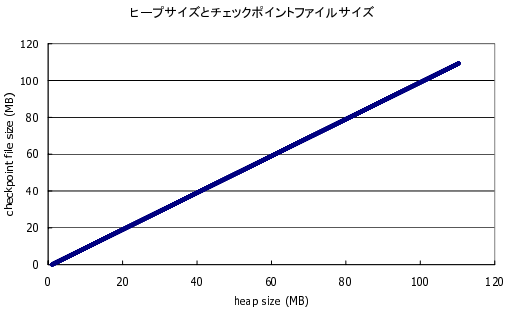


図 8 ヒープサイズとチェックポイントファイルサイズ

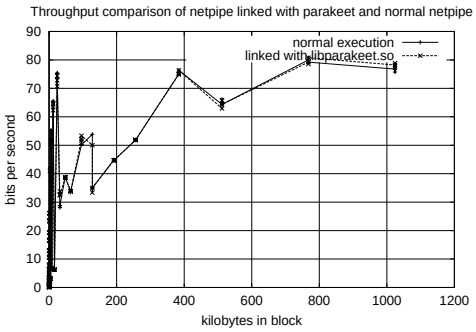


図 9 netpipe-2.4 による，Parakeet ライブラリリンク時，および通常実行時のスループット

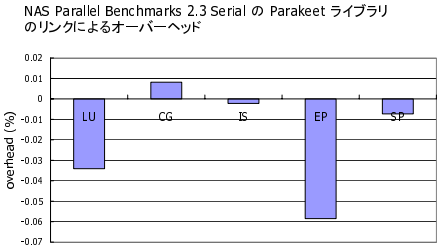


図 10 NAS Parallel Benchmarks 2.3 Serial の Parakeet ライブラリのリンクによるオーバーヘッド

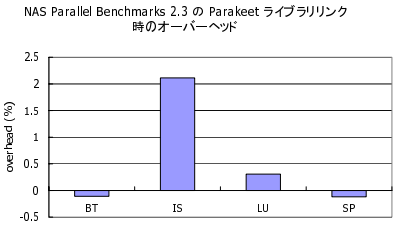


図 11 NAS Parallel Benchmarks 2.3 の Parakeet ライブラリリンク時のオーバーヘッド．使用ノード数は 16 ノード．

ルよりも，interpositioning により置き換わった後のシステムコールの方が高速であるためであり，チェックポインタのオーバヘッドが低いことがわかる．

5.2 Application Tests

NAS Parallel Benchmarks を用いて，Fail Free 時の Parakeet ライブラリのリンクによるオーバーヘッドを計測した (図 11)．使用したノード数は 16 ノードである．NAS Parallel Benchmarks のような，`read()`，`write()`，を内部で頻繁に実行するプログラムでは，Rocks の影響が大きいと考えられる．しかし，計測結果より，オーバヘッドは最大でも 2 % 程度と低く，Rocks のオーバヘッドはほとんど無いことがわかる．

6. 関 連 研 究

2 章で述べたように，MPI にチェックポインティング，マイグレーションを実現しようとする研究・実装は多数行われている．しかし，未実装であったり，PC クラスタで主に用いられる Linux 等に対応したものは少ない．

CoCheck[7] は MPI に sync-and-stop を実装し，coordinated checkpointing およびマイグレーションを実現したものである．CoCheck は元々，Condor[18] 上での並列タスクチェックポインティングをサポートする目的で開発されたが，sync-and-stop 等のオーバヘッドが高いため，実用には至っていない．

Starfish[6] は Parakeet と同様，Ensemble グループ通信ライブラリを用いており，ノードの追加/削除等に対するイベントリスナを定義し，状況に応じたプログラミングを行える．問題点として,Starfish は Ocaml のバイトコード実行であり,MPI のデータ通信部分に Ensemble を使っているため，そのオーバヘッドが大きいことが挙げられる．また，チェックポインティング API を提供することにより，Coordinated Checkpointing 以外の各プロトコルも対象としているが，API の詳細，各プロトコルの表現能力等は不明である．

SCore-D[19] には，時分割スケジューリングで用いられるギャングスケジューリングのために，ネットワークチャンネルの状態の退避，復元を行うネットワークプリエンプション機構がある．西岡ら はこの機構を用いて，sync-and-stop 方式に類似した方法で，通信プロトコル透過にコンシステントチェックポイントを実現している[8]．ただし，SCore-D におけるネットワークコンテキストはネットワークトポロジー的な絶対位置に依存しているので，マイグレーションについては実現していない．

MPI/FT[20] は nMR (n-Modular Redunduncy) とチェックポインティングを用いており，MPI の種類 (MPI-1.2, MPI-2)，アプリケーションのクラス (SPMD, Master/Slave) 等に応じて nMR のモジュール数，rank 数等の nMR のパラメータ，およびチェックポインティングの有無を決定する．MPI/FT は fail-stop ではなく，fail-through を目的としており，そのための障害検知レイヤを提供している．

FT-MPI[9] は MPI-2 を拡張し，`MPI_Send()` 等の関数が fail した場合にコミュニケータの状態が `invalid` となった場合，コミュニケータを新たに作りなおすことにより，復帰するための関数を定義している．Implicit Fault Tolerance[10] はプログラミングモデルを SPMD Master-Worker のみに限定し，FT-MPI で行われるレプリケーション/復帰をある程度自動的に行う．

これら MPI/FT, FT-MPI および Implicit Fault Tolerance はすべて MPI に独自の拡張を加えており，耐故障性を得るためには，ユーザーが各 MPI のプログラミングモデルに応じたコーディングをする必要がある．一方, Parakeet は耐故障性をユーザー透過に提供することを目的としている点でこれらと異なる．

7. まとめと今後の課題

我々は，MPI 上へのユーザ透過なチェックポインティング/マイグレーションの実現を，p4 ライブラリへの改造と Rocks を用いた方法によって試みた．本稿では予備段階として，ユーザレベルチェックポインタと Rocks を統合し，その基本性能について，interpositioning されたシステムコールのオーバヘッド，及び NAS Parallel Benchmarks 実行時のオーバヘッド等について調査した．結果，オーバーヘッドは最大 2 % と低く，十分許容できるものであることがわかった．

今後の課題として，マイグレーション機構等の未完成部分について，実装を完了させる必要がある．ボトルネックであるチェックポインティングの高速化のため，Checkpointing, Incremental Checkpointing, Checkpoint Compression[21] 等の最適化を実装する必要がある．また，残る Rollback-Recovery プロトコルについても実装を行い，比較する必要がある．

Parakeet を用いたクラスタの動的な再構築，スケジューリングに関する試みとして，我々はテストベッドとして ion プラグアンドプレイクラスタ[16] を構築し

ている．今後は，Dynamite PVM[22] で行われているような，マイグレーションによるロードバランシングの試み，および高速クラスタインストーラ LUCIE[23] とマイグレーションによる，クラスタ運用中のノードの動的な追加，削除, ホットスワップといった機能についても調査する．

参 考 文 献

1) Alan M. Mainwaring, David E. Culler: Active Messages: Organization and Applications Programming Interface. (1995)
2) S. Pakin, V. Karamcheti, and A.A. Chien: Fast Messages (FM): Efficient, Portable Communication for Workstations clusters and Massively Parallel Processors., IEEE Concurrency, Vol. 5, No 2, pp. 60–73 (1997)
3) MPI: A Message Passing Interface Standard, http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html
4) Rocks: http://www.cs.wisc.edu/~zandy/rocks/
5) The NAS Parallel Benchmarks, http://www.nas.nasa.gov/Software/NPB/
6) A. Agbaria and Roy Friedman.: Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations., HPDC (1999)
7) G. Stellner: CoCheck: Checkpointing and Process Migration for MPI, Proceedings of the International Parallel Processing Symposium (1996)
8) 西岡 利博, 堀 敦史, 手塚 浩史, 石川 裕: 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, JSPP 1999, pp. 229–236 (1999)
9) Graham Fagg and Jack Dongarra In J. Dongarra, P. Kacsuk, N. Podhorszki (Eds.): FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world, LNCS 1908, (2000)
10) Paraskevas Evripidou and Soulla Louca and Neophytos Neophytou: Implicit Fault Tolerance, http://www.cs.ucy.ac.cy/~skevos/html/ft_for_mpi.html.
11) E. Elnozahy, D. Johnson and Y. Wang: A survey of rollback-recovery protocols in message-passing systems, Department of Computer Science, Carnegie Mellon University, October, (1996)
12) K. Mani Chandy and Leslie Lamport: Distributed Snapshots: Determining Global States of Distributed Systems., TOCS Vol. 3-1,(1985)
13) E. N. Elnozahy and W. Zwaenepoel: Replicated Distributed Processes in Manetho: FTCS-22: 22nd International Symposium on Fault Tolerant Computing, pp. 18–27 (1992)
14) M. Hayden: The Ensemble System, Technical Report TR98-1662, Department of Computer Science, Cornell University (1998)
15) Robinson, Russ, Flachs, and Heckel: A Task Migration Implementation of the Message-Passing Interface, Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing (HPDC-5), pp. 61–68 (1996)
16) ion Cluster Web Page, http://cluster-team.is.titech.ac.jp/ion.html
17) PAPI: http://icl.s.utk.edu/projects/papi/
18) D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, J. Pruyne: A worldwide flock of Condors: load sharing among workstation clusters, Delft University of Technology, Department of Technical Mathematics and Informatics, Technical Report DUT-TWI-95-130 (1995)
19) Atsushi Hori et al.: An Implementation of Parallel Operation System for Clustered Commodity Computers, Cluster Computing Conference (1997)
20) Batchu, Neelamegam, Cui, Beddhu, Skjellum, Dandass, Apte: MPI/FT: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing, CCGRID (2001)
21) James S.Plank, Micah Beck, Gerry Kingsley, and Kai Li: Libckpt: Transparent Checkpointing under Unix, Usenix Winter 1995 Technical Conference, pp. 220–232 (1995)
22) Dynamite: http://www.hoise.com/dynamite/
23) LUCIE: http://cluster-team.is.titech.ac.jp/lucie/