Parallel Erasure Codingを用いた Incremental Checkpointingの最適化

東京工業大学　理学部情報科学科
05B17766　中村　俊介
指導教員　松岡　聡

平成21年8月3日

概 要

大規模なHPCクラスタ環境において、長時間のプロセスを実行する上で、何かしらの不具合により計算が途中で停止してしまったり、オーバーヒートなどによってノード自体が故障し、計算が途中で終了してしまうことがある。このような事態に対処できるように、クラスタ環境における耐故障性の備えが必要となる。

一度実行したプロセスは途中で終了もしくは中断してしまうことがあるので、新たにプロセスを最初からやり直すということを防ぎ、かつ今までの計算過程を無駄にすることなく再利用可能な計算の耐故障性を実現する技術が必要となる。そのような技術の一つとして、チェックポインティングいうものが存在する。

チェックポインティングとは現在実行中のプロセスのイメージや状態をストレージや共有メモリなどに保存して置く技術であり、故障などでプロセスが停止してしまった場合は、ノード復旧時にチェックポインティングを取っておいた地点からロールバックすることで、プロセスを復帰させることができる。しかし、チェックポインティングはあくまでメインのプロセスではないので、かかるオーバヘッドは極力少なくするべきである。

ところが、近年の大規模環境では1) メモリサイズが大きくなる一方で、ディスクI/O帯域があまり向上していないことが要因によりチェックポイント時間が増加する一方であり、このままではMTBF(平均故障間隔)を超え、チェックポイントが実用不可能となってしまう。また、2) ノード数の増加に伴いMTBFは更に短くなり、多重故障が発生するといった問題がある。

これらの問題の為に、チェックポインティング技術に対する様々な点で改善が求められている。従来では、1) にはDiskless CheckpointやIncremental Checkpointによってチェックポイントの I/O と転送時間を削減し、2) はReplicationやErasure Codingを用いたデータ冗長化によって対応してきた。

本研究ではIncremental Checkpoint と Erasure Codingの併用を提案する。具体的にはIncremental Checkpointで縮小したチェックポイントサイズの元で高速にその転送を行い、更にErasure Codingにより多重故障に対応できるようなチェックポインティングシステムを確立した。また、ノード数の増加に対応してそれらをPipeline並列化したデーモンとして実装、評価した。

本手法を適用させた結果、従来のチェックポイントとその転送に比べて、2つのベンチマークプログラム (MAT,NPB LU) に対してそれぞれ83.8%,28.4%の高速化を図ることができた。故に本手法はチェックポイントの性能向上において有効でかつ、実用的であることを示した。

目 次

図 目 次

表 目 次

第1章　序論

1.1　本研究の背景

近年、天文物理学や遺伝子工学や医療などの分野において、非常に時間を要する計算やデータ処理を必要とする研究が増加しており、大規模な HPC クラスタ技術に対する需要が高まりつつある。例えば、物理学の分野では Cornell Pump 社がこれまで 200 時間も要していた渦巻きポンプの設計検証に対して HPC クラスタを用いることでわずか 2 時間で行うことができるようになり、100 倍の高速化が実現し、製品開発期間の大幅な短縮化を実現したということがニュースで報じられている。また、本大学の TSUBAME では総メモリ容量が 20TB にのぼり、大規模並列アプリケーションのメモリオーダーは TB オーダーになりうるため、単一のストレージに格納するにはコストが大きすぎるので、クラスタ技術の必要性が強く叫ばれている。

ところで、このような長時間のプロセスを実行する上で、何かしらの不具合により計算が途中で停止してしまったり、オーバーヒートなどによってノード自体が故障し、計算が途中で終了してしまうことがある。このような事態に対処できるように、クラスタ環境における耐故障性の備えが必要となる。

また、Condor[2] のように遊休計算機を有効に活用しようとするジョブスケジューリングシステムも存在する。Condor のユーザーは計算資源として割当てられた計算機の内、アイドル状態のものにジョブを投入することができるが、もし、計算機が他のユーザーによって使われており、優先度の高いジョブが投入された時は、計算を即座に停止して、他の利用可能な計算機にマイグレーションされるような設計になっている。

このように、一度実行したプロセスが途中で終了もしくは中断してしまうことがあるので、新たにプロセスを最初からやり直すということを防ぎ、かつ今までの計算過程を無駄にすることなく再利用可能な計算の耐故障性を実現する技術が必要となる。そのような技術の一つとして、チェックポインティングいうものが存在する。

チェックポインティングとは現在実行中のプロセスのイメージや状態をストレージや共有メモリなどに保存して置く技術であり、故障などでプロセスが停止して

しまった場合は、ノード復旧時にチェックポインティングを取っておいた地点からロールバックすることで、プロセスを復帰させることができる。また、永久故障してしまった場合にもチェックポイントファイルを別の新しいノードへ渡し、そこで復旧することでプロセスマイグレーションも可能となる。

しかし、チェックポインティングはあくまでメインのプロセスではないので、かかるオーバヘッドは極力少なくするべきである。ところが、近年の大規模環境では 1) メモリサイズが大きくなる一方で、ディスク I/O 帯域があまり向上していないことが要因によりチェックポイント時間が増加する一方であり、このままではMTBF(平均故障間隔) を超え、チェックポイントが実用不可能となってしまう。また、2) ノード数の増加に伴い MTBF は更に短くなり、多重故障が発生するといった問題がある。

これらの問題の為に、チェックポインティング技術に対する様々な点で改善が求められている。

1.2 本研究の目的

従来では、1) には Diskless Checkpoint や Incremental Checkpoint によってチェックポイントの I/O と転送時間を削減し、2) は Replication や Erasure Coding を用いたデータ冗長化によって対応してきた。

Incremental Checkpoint はチェックポイントを取る前後でプロセスイメージの差分を取り、その変更部分のみをチェックポイントファイルとしてダンプするという手法であり、単位時間あたりの変更点が少ないプロセスほどチェックポイントサイズを削減するのに有効とされている。また、Erasure Coding は k 個の data を m 個の coding へとエンコードすることで m 重冗長化をもたせるコーディング手法であり、単純なレプリカ手法と比べると低スペースかつ任意の多重故障に対応できる強力な冗長化手法である。

本研究では Incremental Checkpoint と Erasure Coding の併用を提案する。具体的には Incremental Checkpoint で縮小したチェックポイントサイズの元で高速にその転送を行い、更に Erasure Coding により多重故障に対応できるようなチェックポインティングシステムを確立した。また、ノード数の増加に対応してそれらを Pipeline 並列化したデーモンとして実装、評価した。

1.3 本研究の成果

本手法を適用させた結果、従来のチェックポイントと予備ノードへの転送時間に比べて、2つのベンチマークプログラム (MAT,NPB LU) に対してそれぞれ 83.8%,28.4% の高速化を図ることができた。故に本手法はチェックポイントの性能向上において有効でかつ、実用的であることを示した。

1.4 本論文の構成

第 2 章では、背景としてフォールトトレラントシステムについて述べ、大規模システムにおける耐故障性の必要性と近年どのようなことが問題視されているかについて説明する。

第 3 章では、本研究と関連すると思われる研究を紹介する。

第 4 章では、まず本研究の要となる Erasure Coding について背景を述べた後に、第 5 章で、Incremental Checkpoint with Parallel Erasure Coding の提案とその設計について詳しく説明する。

第 6 章では、Incremental Checkpoint with Parallel Erasure Coding の実装について説明する。

第 7 章では、比較評価を行う。

第 8 章で、まとめと今後の課題について述べる。

第2章　背景

2.1　大規模分散システムにおける耐故障性の必要性

耐故障性を備えたシステム (フォールトトレラントシステム) とは、その構成部品の一部が故障しても正常に処理を続行できるシステムである。障害が発生した場合、単純な設計のシステムでは少しの障害でも全体が停止してしまうが、フォールトトレラントシステムでは完全に機能を保ったまま処理を続行するか、障害の重大性に応じて機能を低下させながら続行させることが可能である。耐故障性は連続稼動が求められるシステムや人命に関わる高い安全性が必要なシステム、高セキュリティを求められる金融システムなどで特に求められている。

耐故障性は個々のマシンの特性というだけではなく、マシン間の連携についての規則の特性でもある。例えば、TCP はパケット通信ネットワーク内に不完全なリンクや高負荷のリンクがあっても信頼性の高い双方向通信ができるように設計されている。これは受信側でパケット喪失、パケット二重化、順序変更などがあるものとしてプロトコルが設計されているためであり、結果として通信性能が低下してもデータの正確性が損なわれてないようになっている。

フォールトトレラントシステムにおける障害復旧にはロールフォワードとロールバックに分けられる。システムに障害が発生してエラーとなったとき、ロールフォワード復旧の場合はその時点のシステム状態で復旧を行い、処理を進める。一方、ロールバック復旧の場合はチェックポインティングのような技術を用いてシステム状態を少し前の状態に戻してそこから処理を再開する。幾つかのシステムではそのエラーの種類や発生箇所によってこの 2 種類を使い分けている。

大規模分散システム環境の各ノードは商用でも扱われているような非常に安価で故障しやすいものであり、また高性能な演算が可能な環境であるため、長時間かかるジョブ実行が行われている。このような環境の中で様々に起こる障害に対してシステム全体の耐故障性を保障するには、フォールトトレランスシステムとして設計/実装される必要がある。

2.2 大規模分散システムにおけるチェックポイントの有用性とその問題点

コンピュータシステムは大規模かつ複雑になると、そのMTBF(平均故障間隔)は小さくなる。そのような状況において並列コンピューティングでは、多数のプロセッサーを用いて長時間の処理を行うことがある。このため、アプリケーション実行中の状態 (全てのリソース確保状況や変数郡の状態など) をコアダンプのような形で定期的に保持しておき、障害が発生したときに最初から処理をやり直すのではなく、途中までの保存された状態からリスタートできるようにする必要がある。この技術をチェックポインティングと呼ぶ。チェックポイントで生成されたファイルは更にそれを別のノードに渡し、それを使用してそのノードから実行を再開することによりプロセスマイグレーションも可能となる。

しかし、あくまでチェックポイントはメインプロセスではないため、かかるオーバーヘッドは極力抑えてるようにしなければならない。ところが、近年の大規模環境では 1) メモリサイズがノード数と 1 ノードあたりのサイズの増加に伴って大きくなる一方で、並列ファイルシステム全体のディスク I/O 帯域があまり向上していない (表 2.1) ことが要因によりチェックポイント時間が増加する一方であり、このままではMTBF(平均故障間隔) を超え、チェックポイントが実用不可能となってしまう。また、2) ノード数の増加に伴いMTBF は更に短くなり、多重故障が発生するといった問題がある。

表 2.1: TOP500 上位スパコンの性能比較

	ASCI Red(1997)	Jaguar(2008)	比較
メモリサイズ	1.2TB	300TB	250 倍
システム I/O 帯域	4GB/s	240GB/s	60 倍

これらの問題の為に、チェックポインティング技術に対する様々な点で改善が求められている。

次章ではまずそのようなチェックポイント技術について従来行われてきた研究について紹介し、更に本研究で重要となっている Erasure Coding を扱った研究について紹介する。

第3章　関連研究

本研究に関連のある研究として、まずチェックポイント自体に関して行われる研究を述べ、次に Erasure Coding を利用した研究を述べる。

3.1　Checkpointing の研究

チェックポインティングの関連研究として Libckpt と Diskless Checkpointing を挙げる。

Libckpt は、Incremental Checkpointing や Forked Checkpointing などを実装したチェックポイントライブラリであり、チェックポイントにかかるデータ容量やチェックポイントによる実行プロセスのパフォーマンスの低下が小さくなるような最適化が行われている。

また、Diskless Checkpointing は、チェックポイントを取る上で大きなボトルネックとなるストレージへの保存という部分に着目して、チェックポイントをストレージに書き込むのではなく、代わりにメモリとプロセッサーの冗長化をうまく利用することで、チェックポイントの I/O 時間を削減している。

以下で、それらの研究に関して詳細を述べていく。

3.1.1　Libckpt

Libckpt[3] はユーザーに非常に透過なチェックポイントツールとして実装されており、ソースコードの一行を変更し、libckpt のライブラリファイルと一緒にコンパイルするだけでシンプルにチェックポイントができる。更に Libckpt は.ckptrc という設定ファイルにチェックポイントのオプションパラメータを記述することにより、チェックポイントの対象となるアプリケーションの特徴に合った最適なチェックポイントが可能となる。

以下ではこの Libckpt がチェックポイントに対して行うことができる最適化の内容を詳しく述べていくが、その前に純粋に Libckpt がどのようにチェックポイントを取っているかを説明する必要がある。

Sequential Checkpointing

上で示した設定ファイル.ckptrc に何もオプションを記述せずに Libckpt の管理下に置かれたプログラムを実行すると、決められた時間ごとにタイマー割り込みが発生し、アプリケーションの実行が一時停止する。そして、Libckpt はその一時停止したプロセスのメモリやレジスタの状態をローカルにチェックポイントファイルを作成してそこに書き込む。プロセスが途中終了してしまい、プロセスの復旧が必要な時は、オリジナルファイルから実行可能な部分を再読み込みし、その後メモリやレジスタの状態をチェックポイントファイルから再構成することで、チェックポイントを取った地点からのプロセスの続行が可能となる。この方法を Libckpt では Sequential Checkpointing と呼ぶ。

チェックポイントを取るインターバルやチェックポイントファイルを読み書きを行うディレクトリなどは.ckptrc にオプションとして記述することでユーザーが指定することができる。

ところが、Sequential Checkpointing はチェックポイントごとに実行プロセスの複製を毎回作っており、また、チェックポイントを行う時に実行プロセスを一時的に停止していることから、実行プロセスに与えるパフォーマンスの低下は大きくなってしまう。このため、Libckpt にはチェックポイントの対象プロセスの特徴に応じた幾つかのオプションが次のように実装されている。

Libckpt に実装されているオプション

- Incremental Checkpointing
- Forked Checkpointing
- ユーザーによる Checkpointing
  - メモリ開放
  - synchronous checkpointing

以下では、Libckpt のオプションについて説明することで、チェックポイント技術にどのような最適化が施されてきたかについて示す。

Incremental Checkpointing

Sequential Checkpointing は、チェックポイントを取る度に実行プロセスの複製をチェックポイントファイルとして作成するようになっているが、これは、プロセスイメージの変更頻度が小さい時には適さない。このような時は、プロセスのイ

メージ全てをチェックポイントファイルに書き出すのではなく、古いチェックポイントファイルと現在のプロセスを比較し、変更のあった dirty page のみを新しいチェックポイントファイルに書き出す Incremental Checkpointing[4][5] という手法が使われる。

Libckpt ではこの Incremental Checkpointing をページプロテクションを用いて実装している。具体的なページプロテクションを用いた Incremental Checkpointing の過程を以下に示す。

1. チェックポイントが行われた後に mprotect() をシステムコールすることにより全てのページのプロテクションを read-only にする。
2. それらのページに書き込みがあったときは、Libckpt のハンドラによって SEGV シグナルが発生し、そのページのプロテクションを read-write に変更する。
3. そのページは変更があったとして、dirty page としてフラグが立てられ、次回のチェックポイント時にこの dirty page のみチェックポイントを行う。

このように Incremental Checkpointing を行うことで、毎回のチェックポイントを取る箇所が少なくなるので、チェックポイント自体は早くなるが、チェックポイントファイルには dirty page の情報しか無い為、古いチェックポイントファイルを残して置かなければならない。また、従来の Sequential Checkpoiting と比べるとチェックポイントファイルからの復旧に時間を要してしまうというデメリットがある。

Libckpt ではこのようなデメリットに対処するために、チェックポイントファイルの数に上限を.ckptrc に設定することができる。具体的には.ckptrc に maxfiles⟨n⟩ と指定すると、Incremental Checkpoint によって生成されたチェックポイントファイルが n 個を超えると Libckpt が ckpt_coa という命令を呼び出し、それらの n 個のファイルを 1 つのファイルにまとめてくれるような仕組みになっている。

Forked Checkpointing

Forked Checkpointing[6][7] は main memory checkpoint と copy-on-write checkpoint から成る。

main checkpoint は、チェックポイントを行っている間に Sequential Checkpointing のように、I/O 処理によってメインプロセスを停止するのではなく、メインプロセスとそのチェックポイントを並行して行う方法である。これにより、チェックポイントによるメインプロセスへのオーバーヘッドを改善することができる。

Libckpt では、これを Unix の fork() システムコールを用いていて実装している。fork を用いた main memory ckpt は具体的に以下のような過程で行う。

1. チェックポイントの際に親プロセスであるメインプロセスから fork をシステムコールし、自身の子プロセスを生成する。
2. 親プロセスから分岐して生成された子プロセスは親プロセスとは別の制御スレッドとして動き、チェックポイントファイルの生成・書き込みを行う。

しかし、ここでプロセスを fork するときのデータ領域のコピーによるコストが大きな問題となる。この解決策の 1 つとして挙げられるのが、copy-on-write である。copy-on-weite は親プロセスからのアドレス空間のコピーを最適化するものであり、具体的には fork が行われる時点では実際のコードやデータ領域のコピーを行うのではなく、その参照先だけを渡すというやり方にして、実際にそれらが使用されるときに初めてコピーが行われるという仕組みになっている。fork を用いたチェックポイントのほとんどがこの copy-on-write を用いている。

ユーザーによるチェックポインティング

これまでは、実行プログラムを修正することなくチェックポイントが可能という点ではほぼ透過的な改善手法を紹介してきたが、Libckpt では、チェックポイントしたいアプリケーションの特徴に合わせてそのアプリケーションプログラム内にユーザーが明示的にチェックポイントの条件を記述することでチェックポイントにかかるオーバーヘッドを改善するような仕組みを幾つか持っている。

メモリ開放

まず、チェックポイントの対象となるメモリ領域を開放することで、チェックポイントにかかるオーバヘッドの削減を図る方法がある。つまり、メモリ領域が dirty でない、もしくはライフタイムが過ぎているときにはチェックポイントファイルから開放することができる。

Incremetal Checkpointing ではメモリを開放する領域の位置の識別を自動的に行っていて、上で挙げたように mprotect() システムコールと SEGV シグナルによるハンドラでページ変更を監視することで実装されているが、これではページ単位でしかメモリ開放ができないので、完全なメモリ開放を実現することができない。

このため、Libckpt ではメモリ開放を次のような手続き呼び出しをユーザーがプログラム中に挿入することで明示的に宣言することができる。

- exclude_bytes(char* addr, int size, int usage)
- include_bytes(char* addr, int size)

exclude_bytes() により次のチェックポイントから指定されたメモリ位置を排除する。これはユーザーがチェックポイントのリカバリーを必要としないと分かってい

る箇所に対して用いる。引数 usage は CKPT_READONLY と CKPT_DEAD を指定できて、前者はユーザーがそれ以降のチェックポイントでその対象メモリ位置に対して include_bytes() を呼び出すまでは readonly とすることができ、後者はメモリが dead したということを明示的に Libckpt に伝え、次に書き込みがあるまでは読み込まれないようにすることができる。

include_bytes() により、次のチェックポイントから指定されたメモリ領域を取り込むことができ、exclude_bytes() の効果をキャンセルできる。Libckpt では一般に明示的に開放されていないプロセスのスタックとデータセグメントのアクティブな領域を全てインクルードしている。

synchronous checkpointing

別のユーザーによる明示的な最適化手法としては、チェックポイントを行うのにメモリ開放が最も効率的に行えるようなプログラムの部分を指定させる synchronous checkpointing という方法がある。

チェックポイントの対象外とする領域は、現在取っているチェックポイントの状態に依存するので、この synchronous checkpointing ではユーザーにチェックポイントを行えば良い箇所を checkpoint_here() という手続き呼び出しをプログラム中に挿入することで、メモリ解放の位置を指定することができる。もちろん、この手続き呼び出しをプログラムの頻繁の訪れるところに置くこともあるので、インターバルの最小値なども指定することができる。

チェックポイントファイルの圧縮

これはチェックポイントファイルを LZW 法のようなアルゴリズムで圧縮することでチェックポイントのサイズを小さくするような方法である。単一プロセッサでは、チェックポイントの書き込み時間より圧縮の方が時間がかかってしまう為に有効ではないが、並列プロセッサでは、他のプロセスへの転送やディスクへのアクセスにかかるオーバーヘッドを軽減するので有用である。

Libckpt は単一プロセスのチェックポインタとして実装されているため、この機能は実装されていない。

3.1.2 Diskless Checkpointing

大規模な分散システムにおいて、チェックポイントのサイズが巨大なものになってくるとチェックポイントをディスクへ保存することが大きなボトルネックとなってくる。

そこで、並列分散システムのチェックポイントからストレージを排除し、代わり

としてメモリやプロセッサーの冗長化をうまく使った Diskless Checkpointing[8][9] という手法をここで紹介する。Diskless Checkpointing とはアプリケーションが動いているプロセッサー (今後これを data ノードと呼ぶ) のプロセスのチェックポイントを取りそのチェックポイントファイルをディスクに書き込むのではなく自身のノードのメモリに書き込み、そして更に、そのメモリ内に保存したチェックポイントをエンコードしてチェックサムを生成し、アプリケーションが走っているプロセッサーとは別にチェックポイント専用のプロセッサー (今後これを coding ノードと呼ぶ) を用意してそれを保存するという方法である。また、data ノードに故障が発生した際の復旧は、まず故障していない data ノードが自身のメモリにあるチェックポイントを元にロールバックし、そして故障ノードの代替ノードを選択して故障していない data ノード全てのチェックポイントと coding ノード上にあるチェックサムから故障したノードのチェックポイントを計算して新しい代替ノードに置く。そして計算したチェックポイントからプロセスを復旧することができる。

disk-based checkpoint と diskless checkpoint には以下のような違いがある。

- disk-based checkpoint
  - 図 3.1(a) のようにディスクに自身のアドレス空間 (スタック、ヒープ、グローバル変数) とレジスタの内容を保存。
  - リカバリー時はアドレス空間とレジスタをチェックポイントの内容で上書きし、そのチェックポイントを取った地点から実行を再開する。
- diskless checkpoint
  - 図 3.1(b) のように自身のメモリにアドレス空間とレジスタをコピーする。
  - アプリケーション全体のコピーがそれぞれの data ノードのメモリに置かれるため、Incremental Checkpointing や fork() を用いる必要がある。

チェックポイントの encoding

単純な encoding 方法として挙げられるのは、parity である。parity とは、データ通信においてデータの誤りを検出する手法の一つとして古くから知られているが、ここでの parity とは、1 つの coding ノードを用意しておき、そこに data ノードそれぞれに対してビット単位で parity を取ったものを置くことで、最大 1 つまでノード故障に耐えうることができるというものである。

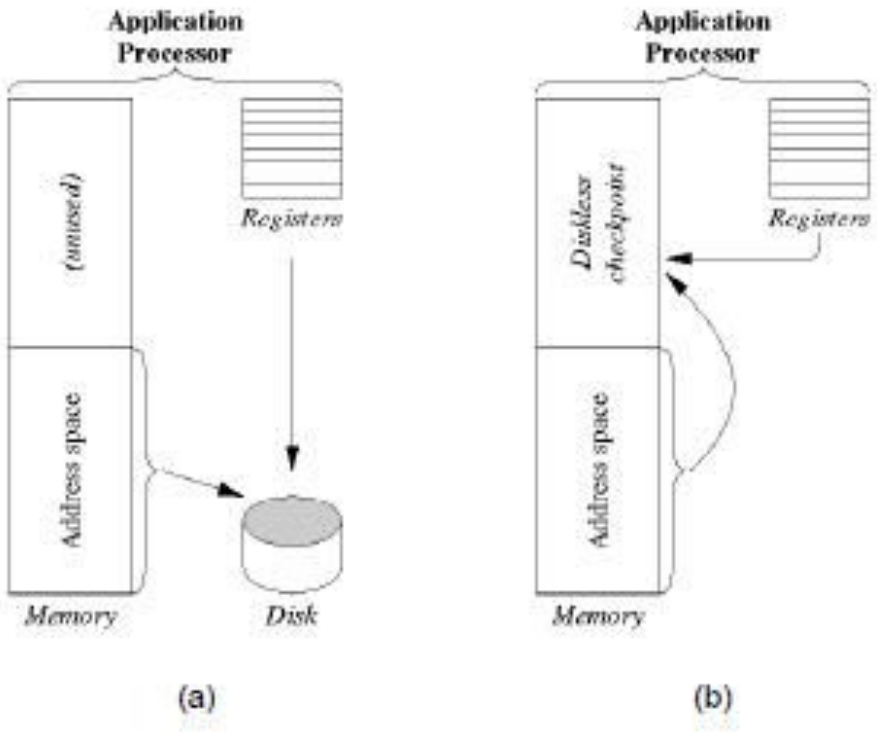


図 3.1: Disk-based Checkpoint と Diskless Checkpoint の保存方法

つまり、i 番目の data ノードの j 番目のバイトを b_i^j で表すとすると、n 個の data ノードに対して parity を取ったとき、coding ノードの parity の j 番目バイトは以下のように計算できる。

$$b_{ckp}^j = b_1^j \oplus b_2^j \oplus \ldots \oplus b_n^j \tag{3.1}$$

もしこのとき、任意の i 番目 data ノードが故障して、新しい i' 番目のノードで復旧させたいときは以下のように取った parity と他の生きている data ノードから parity を再計算することで復旧が可能となる。

$$b_{i'}^j = b_1^j \oplus b_2^j \oplus \ldots b_{i-1}^j \oplus b_{i+1}^j \ldots b_n^j \oplus b_{ckp}^j \tag{3.2}$$

故障したノードが coding ノードである場合でも新しい coding ノードを用意してもう一度 parity を再計算し直せばいいので、実質上どのノードでも 1 つまでの故障に耐えうることができる。故障したノードが復旧したら、残りのデータノードは最新の checkpoint へとロールバックしてその地点から実行を開始する。この parity を用いた手法は RAID5 のようなディスクアレイ技術 [10] に用いられている。

ここで説明した parity 技術では 1 個の coding ノードを用意して単一故障にのみ耐えうるものであったが、その他にも複数の coding ノードを用意して複数の多重故障に対応した技術が幾つか存在する。例えば、mirroring[11] は n 個 data ノードに対して n 個の冗長ノードを用意して data ノードごとに自身のコピーをその冗長ノードに置くこと方法であり、EVENODD coding[12] は 2 個の冗長ノードを用意することで 2 個までの故障に耐えうることできる方法であり、また、Reed-Solomon Coding[13] は m 個の冗長ノードを用意することで m 個までの故障に耐えうることができる方法である。

チェックポイントまたはそのリカバリー中に起こる故障について

Diskless Checkpointing では、チェックポイントを取っている最中もしくはリカバリー途中に故障が起こったときのことも考えられている。Diskless Checkpointing のチェックポイントは、Cordinated Checkpointing という各 data ノードが一貫してチェックポイントを取る方法を用いているので coding ノードが次の Coordinated Checkpoint が終わるまでそれぞれのチェックポイントが有効かどうかを確認することでこれに対処している。全ての coding ノードが現在のチェックポイントの encoding を計算し終えると、過去の encoding を削除して data ノードでは過去のチェックポイントを削除する。また、リカバリー時は最新のチェックポイントに対する encoding が全て有効であれば、それから故障ノード分のデータをリカバリーする。もし、有効で無いものがあれば、過去のチェックポイントと encoding があるので、それらから過去の故障ノード分のデータを復元して、他の data ノードはそのチェックポイントの地点にロールバックすることで、実行を再開することができる。

上から分かるように全て encoding の保存が完了するまで、各 data/coding ノードは 2 つのチェックポイント/encoding を保持する必要があるため、メモリの利用方法が Diskless Checkpointing において重要になってくる。そこで、Libckpt で挙げたような Incremental Checkpointing を用いることによってチェックポイントサイズを小さくすることで、これに対処している。

encoding の送信・計算方法

チェックポイントの encoding を計算する際に幾つかのデザインの選択肢がある。まず、チェックポイントの送信方法であるが、これは通常チャンクごとにチェックポイントを分割して送信を行い、encoding は chunk-by-chunk で計算される。これによって 1 ノードあたりの 1 つのチャンクを読み込むスペース量を制限すること

ができる。実際の送信には broadcast または fan-in という方法が使われる [13]。

broadcast は、図 3.2 のように各 data ノードごとに全ての coding ノードに対して broadcast によってチャンクを送信する方法で、coding ノードは全ての data ノードからチャンクを受け取るとそれから encoding を計算する。また fan-in は、図 3.3 のように各 coding ノードごとに全ての data ノードのチャンクを集めてそれらから encoding を計算する。この集め方には幾つかのアルゴリズムがあり、2 分木のように data ノード間で 2 つずつチャンクを集めたり、もっと複数のチャンクごとに集めて最終的に coding ノードに送る方法が考えられる。

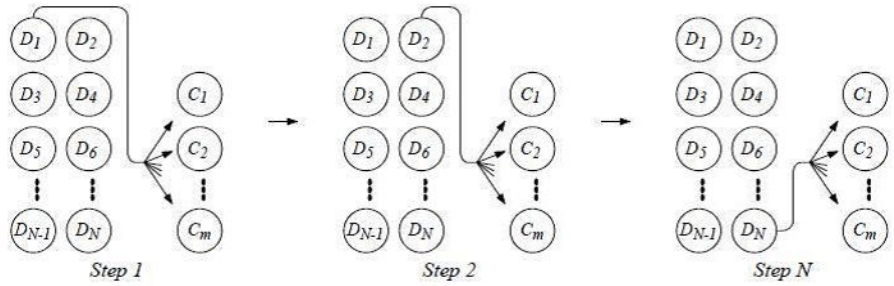


図 3.2: Broadcast アルゴリズム

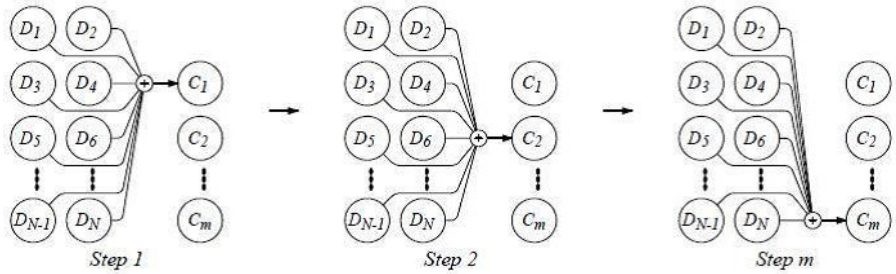


図 3.3: Fan-in アルゴリズム

3.2 Erasure Coding の研究

Diskless Checkpointing のところで述べた通り、チェックポイントの encode 方法には様々なものがあり、そのそれぞれが Checkpoint によってかかるレイテンシーやサイズなどといったオーバーヘッドに大きな影響を与えることになる。

そこでこの節ではその encode 方法のうち、Reed-Solomon Coding の一種である Erasure Coding についてそれを分散システムのストレージに適用することで各オーバヘッドの削減を行った研究を紹介する。

Erasure Coding[14] とは、n 個の data ブロックを m 個の coding ブロックに coding することで、n+m ブロックのうち任意の m 個まで失っても元の状態に復元できるというものである。分散システムにおける RAID1 などで扱われるような mirroring ではそれぞれの data ブロックに対して自身の複製を作ることで複数の障害に耐えうることができたが、これでは冗長化サイズが data ブロックと同じだけ取ることになるでコストが高い。そこでこのような replication 手法の代わりとして Erasure Coding を分散システムに適用することによって、冗長化にかかるサイズコストを data ブロックのうちその中で最大サイズのものへと抑えることが可能となる [15][16]。

Erasure Coding の詳細については本研究において重要な部分であるので次章の提案で後述する。

3.2.1 分散システムのストレージに利用 [1]

従来の分散システムのストレージにおける耐故障機能として備えられていたのは例えば RAID に使用されているような単一の parity であったり [17]、k-way replication といった技術である。しかし、単一の parity では一つの障害にしか耐えられないし、 k-way replication では k の値がごく小さいものであると冗長化にかかるコストがあまりにも大きくなってしまう。

そこで、分散システムのストレージにおいて多様な故障に対応できて、かつそれにかかるコストが最小限で済むような仕組みが必要となる。Erasure Coding は従来通信システムで用いられていたが、最近では replication の代わりとしてストレージ内で使用されている。 この Erasure Coing を適切に使用すれば、スペースの効率化やプロテクションの調整が可能となる。

ここで、Erasure Coding が従来の replication 手法よりも強力である簡単な例を示す。
まず、data ブロックとして a と b というものがあると仮定する。次に、この 2 つの data ブロックに対して replication 手法または Erasure Coding を用いると、例えば以下のようにデータを冗長化させることができる。

- replication: (a,b,a,b)
- Erasure Coding: (a,b,a+b,a-b)

両者とも 2 つの data ブロックに対して 2 つの冗長ブロックを追加しているので、冗長化にかかるサイズのコストは等しい。しかし、protection の度合いでは両者異なる。例えば、任意の 2 つのデータが同時に損失したと仮定すると、replication の

場合は損失したものが a,b であれば復元可能だが、両方の a または b が損失したときは復元不可となってしまう。ところが、Erasure Coding の場合はどの 2 つのデータが損失しても復元が可能である。例えば、a,b が損失したときは a+b,a-b の和や差を取ることで復元は可能であるし、a,a+b が損失したときも b と a-b の和から a が導出され、それから続いて a-b も導出できるので復旧可能である。
このように replication と比べると Erasure Coding は protection の点で強力であることが分かる。

Erasure Coding を用いたストレージのデザインとアルゴリズム

この研究では Erasure Coding を用いたストレージをデザインする上で以下のような規則に従っている。

1. Shift functionality to clients
   Client ノードを実行用ノード、Storage ノードを受信用ノードとすることで優れたスケーラビリティを供給し、故障からのリカバリーを単純化し、新たな storage ノードの追加に伴うコストを削減することができる。
2. Optimize for common cases, simplify rare cases
   故障が無い通常は一般にはロックや 2 相コミットのようなコスト高なメカニズムを避ける。しかし、稀に故障が起こった場合は locks を介して強い調整を行うことでデザインを単純化する。
3. Hide intrinsics of mechanism being implemented
   Erasure Coding によってアプリケーションのインターフェイスに影響を与えない。例えば、Erasure Coding による冗長化のサイズコストが大きくなるからといってアプリケーションに用いるブロックサイズが大きくなったりはしない。

一貫性　並列性と故障が存在する中で、強い整合性を保証する。つまり、多面からの書き込みと読み込みが同時に発生すると古い値もしくは新しく書き込まれた値が読み込まれる可能性があるが、ここでは書き込まれていない値や別の書き込みによって上書きされた値を読み込んだりはしないということを保証している。

構成　この提案手法は主に次の 2 つの部分から構成される。

1. storage ノードは client ノードからの単純な要求に対して働くように構成される。

2. client ノードはデータを保存・検索・復元するために storage ノードをオーケストレーションする。

また、論理的にこの提案手法は次の 4 つのコンポーネントを持っている。

1. 故障検知とノードのマッピング変更
2. read/write のアルゴリズム
3. リカバリーのアルゴリズム
4. ガベージコレクションのアルゴリズム

Erasure-coded 分散ストレージにおける問題点

分散ストレージで一般的に行われることは故障や並列アクセスに関わらずデータの整合性を供給することであり、それと同時にどのクライアントにおいて信頼性のあるパフォーマンス、ストレージのキャパシティの拡張が求められる。
これらの問題点は replication ベースなストレージにおいてよく知られていることであるが、Erasure Coding を用いた場合はその複雑さの為に更に幾つかの問題点がある。

replication を用いたストレージに良く使われている解決策が Erasure Coding には使用できない理由が以下のように 2 つある。

1. replication は、同一のレプリカとしかカップリングしていないのに対して Erasure Codes は異なる data ブロックとカップリングしている。
2. 検知・補正するのに replication に比べて Erasure Codes の方が発散性が強い。

これらの例を説明するために、先程の (a,b,a+b,a-b) を例にとり、この data ブロックがそれぞれ別の storage ノードに保存されていると仮定する。
今、client ノード c_1 が a を c に変更しようとする。一方で別の client ノード c_2 が b を d に変更しようとする。異なるデータに対して更新が行われるが、Erasure Coding では c と d はカップリングしているので、並列実行される必要がある。最終的に (c,d,c+d,c-d) という結果を得るために、c_1 と c_2 の衝突はロックを用いることで避けることができる。つまり、c_1 は 4 つの data ブロックをロックし、a を読み込んで上書きする。c_2 についても同様な操作を行う。しかし、ロックはコスト高なのであまり使用したくないものであるし、c_1 が上書き途中に失敗すると (c,d,c+d,a-d) のような不整合が起こってしまう。

ここでもし、2 つ以上のノードが故障すると、この不整合により正しくデータを復元できなくなるだけでなく、どのブロックが不整合なのかということも検知で

きなくなってしまう。例えば、2,3 番目のノードが故障したときは (c,-,-,a-b) となるが、故障した 2 番目のノードは c-(a-b) というデータとして復元されてしまう。

故障検知とノードの再マッピング

この手法では、周期的に ping を用いてノード故障はクライアントがノードのアクセスした時に検知される。ノード故障を検知したクライアントは消失したデータを再構築する為にコストのかかる命令を発行する。もし、故障ノードが永久に復旧しないようなら、別の新しいノードにそのデータを移す。

そのような場合、新しいノードが利用可能だとすれば、ディレクトリサービスのようなあるメカニズムを適用することが出来る。つまり、クライアントにこの新しいノードへ位置づけて、故障により再配置された論理ノードへアクセスできるようにする。storage ノードにはデータが有効かどうか、または初期化されていないゴミかどうかを示すフラグが用意されている。

read/write アルゴリズム

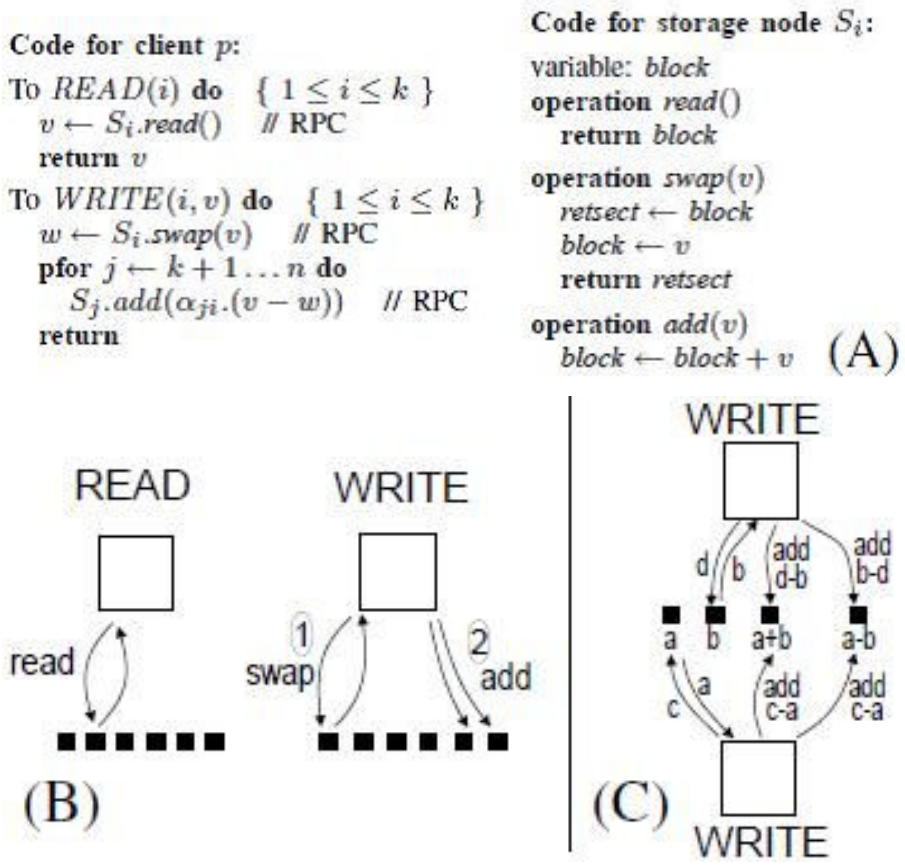
以下ではこの研究での手法の読み書きのアルゴリズムについて簡単に説明する。

図 3.4 は提案手法のアルゴリズム全体のコアとなる部分を単純に表したものである。これは Erasure Codes に従う n コのノード内のデータを単に保持するアルゴリズムで、耐故障性の面については考えられていない。

S_1~ S_k までが data ノード、S_{k+1}~ S_n までが Erasure Coding に従う coding ノードとする。READ は単にノード S_i(i < k) 上で read 命令を呼ぶだけである。WRITE は値 v を古い値 w を持つノード S_i に書き込むとすると、swap 命令により v と w を swap する。そして coding ノードに α_{ji}・(v-w) を加える。ここでの α_{ij} は有限体 GF(2^h) 上の定数を表している。(この有限体については Erasure Coding において重要な部分を成すので、詳細について次章の提案にて後述する。) pfor は”parallel-for” のことで、並列実行され、pfor が終わると実行はマージされる。

このアルゴリズムの特徴としては、複数のクライアントから並列に書き込みがあっても、ロックや 2 相コミットのような同期を取る必要なく、Erasure Coding の整合性を保つことができるという点である。((C) 参照)

図 3.4: 単純化したアルゴリズム (a):コード,(b):図,(c):クライアントの調整なしに整合性を保ったまま並列書き込みを行う例



リカバリーアルゴリズム

リカバリーアルゴリズムの基本概念は storage ノードから全てのデータを読み込み、Erasure Codes を用いて decode し、結果を書き戻すというものである。

リカバリーにおいて以下の 2 つの問題点が挙げられる。

1. クライアント p がリカバリー実行中に故障したとき、リカバリーは別のクライアントにより行われなければならない。
2. リカバリーと並行しての書き込みを行うとリカバリー完遂後に冗長ブロックは改ざんされているかもしれない。

これらの問題はブロック全体が整合性を保っているかどうかを判断するのに、各ノードは内部で過去の書き込みの識別子を記録したリストを持つことで解決できる。

リカバリーをクライアント p により実行されるとすると、その処理は以下の 3 フェースからなる。

Phase1 p はそれぞれのノードにロックをかける。
(各ノードのロック状態はローカル変数 lmode で維持。lmode=UNL:add,swap 可,lmode=L1 は不可)

Phase2
1. p は全てのノードの内容・状態を読み込み、整合性のあるブロックが k コ以上あるかどうかを調べる。(k は Erasure Codes によって復元に必要なブロックの個数)
2. k コ無い場合は、lmode=L0 とすることで冗長ブロック上のロックの条件を緩める。
   このモードはロックはしたままで実行に add 命令を加えるモードであり、ここで p は未解決の書き込み部分に対して add 命令を行うことでブロックが整合性を持つようにする。
3. 次に、p は lmode を L1 に変えて、これ以上 add 命令が起こらないようにする。
4. もし、p が失敗したら整合性のあるブロックを再探索、そうでないならノードの変数 opmode に RECONS をセットする。

Phase3 p は読み込んだ整合性のあるブロックから Erasure Coding を用いてデータを再構築し、ノードへ復元したデータを書き込み、ノードのロックを解除する。

第4章　Erasure Coding

本研究の提案を述べる前に Erasure Coding の具体的なアルゴリズムとそれを実装したライブラリであり、今回の並列化の対象となる「Jerasure[18]」について説明する。

4.1　Erasure Coding

まず、Erasure Coding の encode/decodeする際に深く関わっているガロア体、ガロア拡大体について詳細を述べる。

4.1.1　ガロア体

ガロア体とは整数をある素数で除算した余りの集合であり、要素が有限で四則演算が閉じている集合である。

もう少し詳しく説明すると、集合 F が次の条件全てを満たすとき、F は加法・乗法に関して法をなすという。

このとき、有限個の要素からなる体を有限体 (finite field) またはガロア体 (Galois Field) と呼ぶ。

1. 集合 F 上に加法・乗法が定義されている。
2. 集合 F に単位元が存在する。(加法の単位元を 0, 乗法の単位元を 1)
3. 集合 F の任意の要素 a に対して、a+b=0 を満たす加法の逆元:b=-a、零元を除いて、a*c=1 を満たす乗法の逆元 c=a^{-1} が存在する。

どんなガロア体も要素数 (濃度) がある素数のべき乗に等しく、かつ、同じ濃度であれば本質的には同じものとなる。

つまり、元 (要素) の総数によってガロア体は完全に決定される。また、集合 F の濃度を p とすれば、濃度 p のガロア体を GF(p) または F_p として記し、さらに、F_p から零元を除いた集合を F_p^* と記すこととする。

ここで、a∈F_p^* において、a の位数 (order)ord(a) とは、a^s=1 となる最小正整数

表 4.1: GF(2) の加算と乗算

加算	0	1
0	0	1
1	1	0

乗算	0	1
0	0	0
1	0	1

s のことを指す。
ord(a)=s ならば、a,a^2, . . . ,a^s は全て相異なる。

このような特徴からガロア体は CRC のようなエラー検出処理や SDRAM や通信で用いられるエラー訂正処理、AES のような暗号処理などで古く使われてきた。特に、コンピュータの世界ではビット列で処理しているため、GF(2) を元にしたものが利用されている。表 4.1 のように GF(2) の加算は XOR であり、乗算は AND を表している。

4.1.2 ガロア拡大体

ある体 F があるとき、体 F の元を係数とする代数方程式の解が F 上に存在するとは限らない。
その「解けない方程式」の解を形式的に用意して、体 F に付加することで拡大体を作ることができる。以下の方法で拡大体を作ることができるが、どの方法も同じことを言っている。

- 体 F の範囲では解けない代数方程式 f(x)=0 の解を F に付加する。
- 体 F に対して、条件 f(x)=0 を満たす新しい元 x を加える。
- 体 F 上の多項式 F[x] の変数 x に対して、条件 f(x)=0 を付ける。
- 体 F 上の多項式 F[x] を既約多項式 f(x)=0 で割った余りの集合を作る。
- f(x) を生成元とするイデアル f(x)F[x] を使って剰余環 F[x]/f(x)F[x] を作る。

体 F の拡大体は F 上の多項式環 F［ x ］の剰余体になっており、拡大する多項式の条件は既約であることでなる。多項式で素数に当たるものを既約多項式といい、その中で周期が最大となるものを原始多項式と言う。

ここで 2^w の要素 (元) をもつ体、2^w のガロア拡大体 GF(2^w) について考える。まず、GF(2) 上の元 {0,1} を係数にもつ w 次の原始多項式 p(x) を考える。新しく

α という元を考え、p(α)=0 と仮定すると、α^0,α^1, ... ,α^{x-2}(ただし x=2^w) は全て異なる要素となり α^{x-1}=1 を成立させることができる。
これに零元を加えると、GF(2^w) の元となる。つまり、GF(2^w) は以下のように表せる。

$$GF(2^w) = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{x-2}\}(\text{ただし } x = 2^w) \tag{4.1}$$

4.1.3 Erasure Coding

Erasure Coding は下の図 4.1 のようなモデルとなっており、Erasure Coding の encoding とは k 個の data デバイスの内容を m 個の coding デバイスにエンコードすることであり、decoding とは (k+m) 個の総デバイスのうち、そのうちの故障していな k 個を揃えて再計算することで元のデータを復元できるというものである。

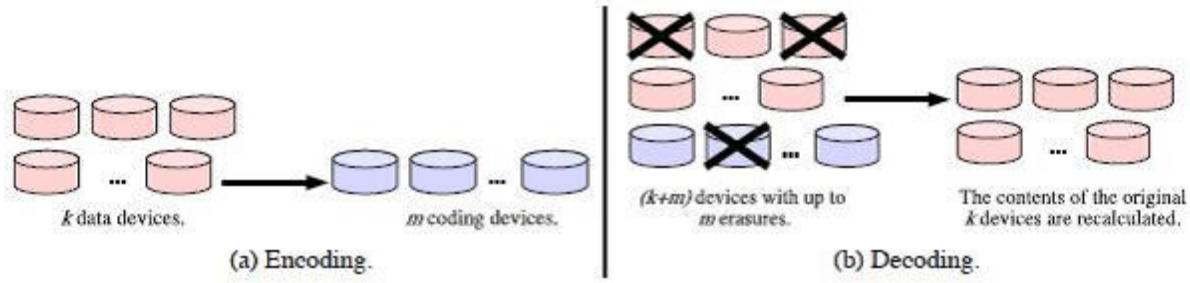


図 4.1: Erasure Coding のモデル

ここで k,m の他に第 3 引数として w を用意する。
これはワードサイズを表しており、各デバイスは w ビッド相当のデータを持っていることを示している。
つまり、data デバイスを D_0~ D_{k-1},coding デバイスを C_0~ C_{m-1} で表すとき、D_i,C_j の格納するデータは「$d_{i,0}$, ... ,$d_{i,w-1}$」「$c_{j,0}$, ... ,$c_{j,w-1}$」として表せる。
また、w を w∈{8,16,32} とすれば、バイトごとに w の集合として捉えることができるので、それぞれのデータデバイスに対してバイトごとにエンコードを行うことができる。
この方法は Reed-Solomon Coding の標準的なものである。
ここで、Erasure Coding の encoding 方法は下の図 4.2 のように行列ベクトル積の形で表すことができる。
左辺の (k+m) × k 行列は Distribution Matrix(DM) と呼ばれている。この DM は identity matrix と coding matrix から構成される。

identity matrix は k × k の基本行列から成る。
また、coding matrix は m × (k+m) の行列であり、各行列要素には GF(2^w) 上の 0 ~ 2^w-1 の整数値が入っている。

行列演算にはガロア体の演算を用いる。つまり、加算には排他的論理和 XOR を用いる。乗算については下で扱う Jerasure の中では様々な方法が実装されている。

また、coding matrix の生成にも様々な方法があり、下の Jerasure を初め各 Erasure Coding のライブラリでその生成方法が実装されている。

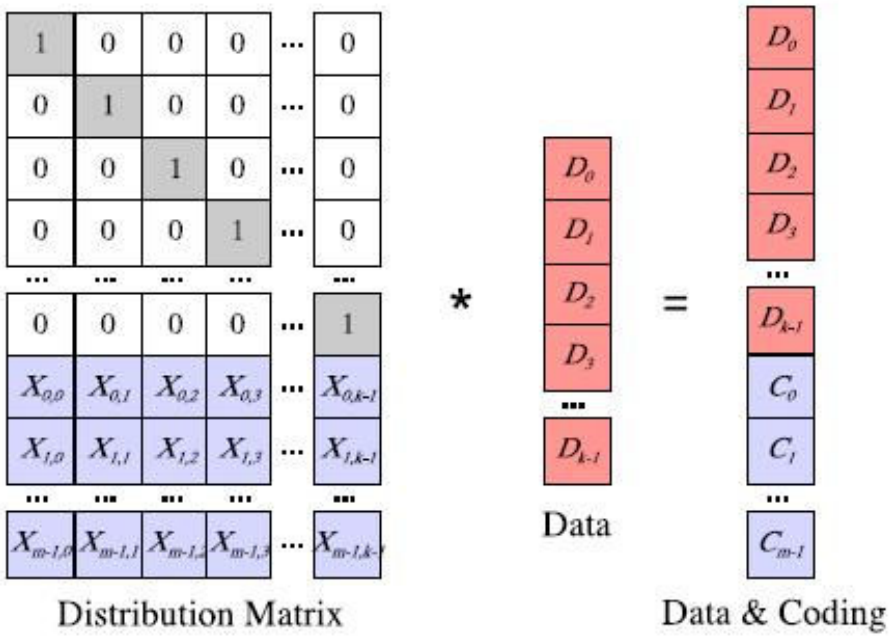


図 4.2: 行列ベクトル積を用いた Erasure Coding のモデル

上の 4.2 は Erasure Coding の encode 方式を表しているが、decode の場合は各ワードごとに DM の行に相当するものを持っているので、故障していないデバイスに相当する行をその中から任意に k 行選んで正方行列を生成し、その逆行列 (Decoding Matrix) と生きている任意の k 個の data&coding とのベクトル行列積を求めることで 元のデータを復元することができる。

4.2 Jerasure

Jerasure は Libckpt の作成者でもある Jame.S.Plank によって実装された単一プロセッサー用の Erasure Coding ライブラリであり、C/C++で実装されている。

上で説明したような Erasure Coding の encode/decode のモデルを実装している

だけでなく、更に、DM の要素が全て 0,1 のビットになるように Binary Distribution Matrix(BDM) というものへと拡張することで、XOR や乗算において演算に対しての最適化を行ったり、また、この BDM が疎密度行列であることに着目して、予めどの data デバイスと BDM の要素に対してどのような演算を行うかということをスケジューリングすることで、encode/decode にかかる演算コストを抑えるような最適化が行われている。

4.2.1 Jerasure サンプルプログラム

Jerasure には、実装されている Erasure Coding ライブラリをデモンストレーションするようなサンプルプログラムが幾つか用意されている。

サンプルプログラムの流れは大まかに以下のような流れになっている。

1. ユーザーが k,m,w といった引数を与えてプログラムを実行
2. 与えられた k,w を元にして k 個の data ブロックをランダム生成
3. k,m,w から coding matrix をそれぞれのアルゴリズムで生成
4. data ブロックと coding matrix からベクトル行列積により m 個の coding ブロックを生成
5. (k+m) 個の data&coding ブロックのうち、その中から m 個のブロックをランダムに削除
6. 生きている data と coding matrix から decoding matrix を生成
7. 生きている data&coding ブロックと decoding matrix の逆行列のベクトル行列積より元の data&coding ブロックを復元

例として、DM に基づいたサンプルプログラム jerasure_05 を実行例を下の図 4.3 に示す。これは data デバイス数 (k):2,coding デバイス数 (m):3,GF(2^w) の w:8, 各デバイスの容量バイト (size):4 で実行している。

data&coding ブロックの生成時と復元時に表示される data&coding の内容が等しいことから、Erasure Coding による encode/decode がうまく機能していることが分かる。

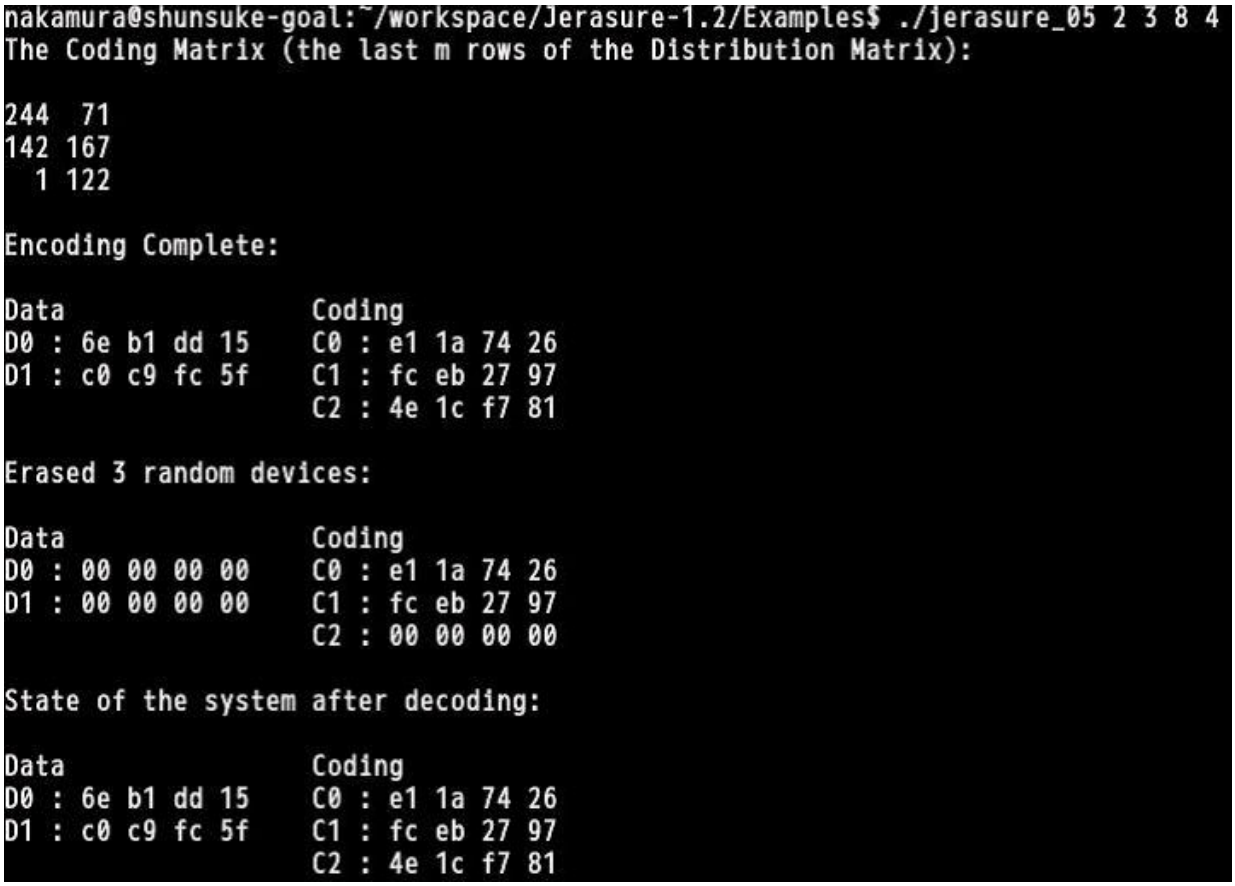


図 4.3: サンプルプログラム jerasure_05 の実行例

第5章　提案と設計

第 3 章関連研究でチェックポイントを行う上で 1)Incremental Checkpointing を用いることでチェックポイントのサイズコストを大幅に下げる研究を紹介した。サイズコストを下げることでその転送時間も削減することが出来る。

また、2)Erasure Coding を用いることで多重故障に対応し、かつ比較的低コストで耐故障性を実現できることを示した。

そこで本研究ではそれらの 2 つの技術を組み合わせることでチェックポイントの時間の削減と低スペースな冗長化を同時に実現するできることを提案する。また近年の大規模システムのノード数増加に対応できるよう、それらを pipeline 転送を用いた並列化デーモンとして提案し、**Incremental Checkpoint with Erasure Coding** を実装した。

この章では **Incremental Checkpoint with Erasure Coding** についてまず簡単な概要を述べてから、それを設計する上で我々が設けた仮定について述べ、最後にその設計方針と通信方式について説明する。

5.1 Incremental Checkpoint with Erasure Coding

Incremental Checkpoint with Erasure Coding は次の 3 つの部分から構成される。

5.1.1 Parallel Erasure Coding

Parallel Erasure Coding はある特定のチェックポイントファイルのエンコードを行うために各そのファイルの担当ノードと coding ファイルを保持するノードに予め立ち上げておくデーモンである。各デーモンを立ち上げるとそれぞれある特定ノード上の別のデーモンとコネクションを張ることができる。そして、あるタイミングでユーザーがそれらの担当するチェックポイントファイルに対してエンコードを行う場合は、また別の特別なデーモンを立ち上げる。この特別なデーモン (今後これを console process と呼ぶ) ではエンコードの最初の担当ノードのデーモンに

コネクションを張ってくれるので、コンソールプロセスに対してユーザーがコマンドを打つことにより先程張ったデーモン間のコネクションを元に並列エンコードが行われ、最終的にエンコード完了した coding ファイルは coding 担当ノードのローカル上に保持されるという仕組みになっている。

5.1.2 並列チェックポイント

エンコードの対象としてチェックポイントファイルを生成するチェックポイントライブラリとしてカーネルレベルチェックポイントの BLCR(Berkeley Lab Checkpoint/Restart) を用いる。(詳細は次章で後述) BLCR はチェックポイントの対象となる動いているプロセスの ID(pid) を元にチェックポイントを取ることができるようになっているので、この pid を予め pid テーブルとしてファイルにまとめておき、上で述べたデーモンと共有しておくことにより指定したプロセスのチェックポイントとエンコーディングが可能となる。

チェックポイントを並列に取るには、GXP タスクスケジューラーを用いて各ノードにログインして並列にチェックポイントのジョブを投げるという手法を用いる。

5.1.3 インクリメンタルチェックポイント

チェックポイントライブラリの選択でインクリメンタルチェックポイントを実装している Libckpt を選ぶこともできたが、しかし、Libckpt のインクリメンタルチェックポイントは実プログラムの実行速度を大幅に下げてしまうという大きな欠点から断念した。そこで本研究は比較的扱いやすい BLCR に LINUX の xdelta プログラムを用いることで、擬似インクリメンタルチェックポイントを実装する。

便宜上、並列チェックポイントとインクリメンタルチェックポイントを 1 つのスクリプトとしてまとめておく。

5.2 Parallel Erasure Coding 設計上の仮定

Parallel Erasure Coding を設計する上で、まず次のような仮定を設けている。

- k,m,w,size,processid といった coding に必要な情報は予め各ノードが設定ファイルから読み込んでいる
- 各 data ノードは自身の data ブロックのみを受け持つ。(i 番目の data ノードはデータ D_i のみを保持する)

- 各 data&coding ノードは共通の coding matrix を予め保持している。
- 各 data ノードはどこのノードにアクセスするか、どのファイルを checkpoint,coding するのか予め指定する設定ファイルを各自持っている。

5.3 Parallel Erasure Coding 設計モデルと通信方式について

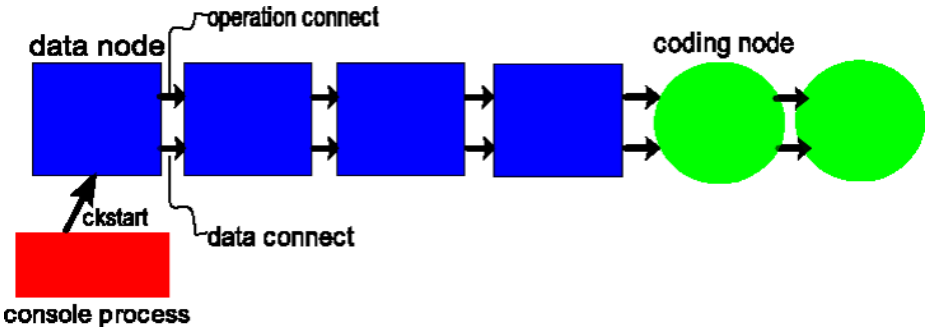


図 5.1: Parallel Erasure Coding モデル

Parallel Erasure Coding の全体モデルは上の図 5.1 のようになっており、通信方法としてはパイプライン転送を用いる。これはあるプロセスを横一列に接続して、ジョブを順に実行しながら隣へ受け渡していく方法であり、逐次で実行した時と比較すると、下図 5.2 のような encode 時間のコスト削減が出来ると予想される。

この図の例では k=4,m=2 としており、各 size バイト単位の処理 (encode,coding のセーブ,codingblock の転送) を size_t としたときの、逐次処理と並列パイプライン処理を抽象的に表したものであり、逐次の場合は data の encode に 4*2*2=16 ステップ (size_t)、coding に 2*2=4 ステップかかり、合計で 20 ステップかかるのに対して、これを並列パイプライン処理にすることで、encode/coding を最小で 9 ステップで行うことができると予想される。(転送遅延などは考慮に入れない)

具体的な encode には以下のようなステップをとる。

1. 全プロセスをブロッキング状態にしておく。
2. ユーザーから最初の data ノードにジョブを発行。(コンソールプロセス)
3. 以下を全プロセスが左から順に行う。
   (a) プロセスをアンブロッキングしてジョブの実行。
   (b) 実行終了後、実行プロセスは隣のプロセスへジョブと必要なデータを渡す。
   (c) 次の命令が来るまで自身を再びブロッキングする。

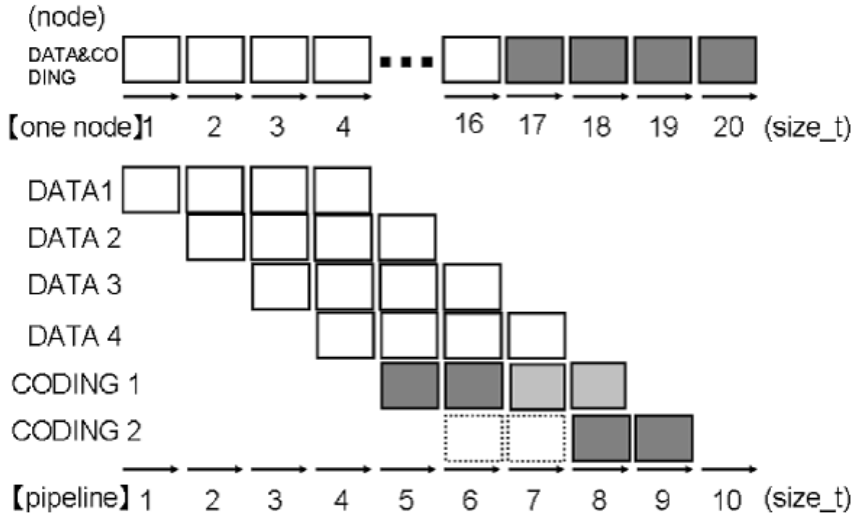


図 5.2: 逐次実行と並列パイプライン実行の違い

data ノード、coding ノードが行うジョブとは以下のようなものになる。

- data ノードのジョブ
  1. left hand ノードから coding ブロックを受信。
  2. 受け取った coding ブロックと自身の data ブロックを coding matrix の要素を元に coding 演算して新たな coding ブロックを生成。
  3. right hand ノードへ新たな coding ブロックを送信。

- coding ノードのジョブ
  - 受信した coding ブロックが自身担当の場合は特定のファイルに書き込む。
  - 受信した coding ブロックが他のノードの担当の場合は right hand の coding ノードへそのまま送信。

全体をまとめると以下のような流れになる。

1. 各ノード、coding に必要な情報と checkpoint,coding の対象となる data ファイルを設定ファイルから読み込む。
2. 各ノード、設定ファイルを見ながら right hand ノードに Operation 用のコネクションを張る。
3. ユーザーが console process を実行。
4. 各ノードは right hand ノードと Data 用のコネクションを張り、Operation コネクションを介して各プロセスに実行内容 (checkpoint,encode,decode) を渡す。

- **checkpoint 命令**
  data ファイルに対してチェックポイントをとる。
- **encode 命令**
  left hand ノードから受け取った coding ブロックと自身の data ブロックから encode して新たな coding ブロックを生成して、right hand ノードに送信。最終的に coding ノードに全体の coding ブロックが置かれる。

今回はこの実装した Parallel Erasure Coding をチェックポイントに適用する。つまり、各 data をそれぞれのノードのチェックポイントファイルにマッピングし、それを size バイト分割してエンコードし、パイプライン転送を行う。

第6章　Incremental Checkpoint with Erasure Codingの実装

並列化の対象となるJerasureライブラリは全てC言語で実装されている為、それに応じてJerasureライブラリの並列化からcodingblockの通信部分に至るまで実装はC言語を用いた。この章ではまず最初に、本論分の核となるそれらをC言語でどのように実装したかを説明し、次にチェックポイントライブラリとして用いたBLCR、並列チェックポイントを行うために用いたgxpタスクスケジューラーについて述べる。最後に次章の実験を行う上で時間測定をどのような方法で行ったかについて説明する。

6.1　Jerasureライブラリの並列実装

まず、Jerasureの逐次エンコードを行うメソッドの詳細について述べ、それをどのように並列化したかについて説明する。以下のコードはJerasureの逐次エンコードに使用するライブラリ関数を表している。(並列化する上で重要な部分のみを抽出)

```
void jerasure_matrix_encode(int k, int m, int w, int *matrix,
                          char **data_ptrs, char **coding_ptrs, int size)
{
  int *init;
  int i, j;
  for (i = 0; i < m; i++) {
    jerasure_matrix_dotprod(k, w, matrix+(i*k),  k+i, data_ptrs, coding_ptrs, size);
  }
}
void jerasure_matrix_dotprod(int k, int w, int *matrix_row,,int dest_id,
                         char **data_ptrs, char **coding_ptrs, int size)
{
  int init = 0;
  char *dptr, *sptr;
  int i;
  dptr = coding_ptrs[dest_id-k];

  /* First copy or xor any data that does not need to be multiplied by a factor */
  for (i = 0; i < k; i++) {
    if (matrix_row[i] == 1) {
      sptr = data_ptrs[i];
      if (init == 0) {
        memcpy(dptr, sptr, size);
        init = 1;
      } else {
        galois_region_xor(sptr, dptr, dptr, size);
      }
    }
  }

  /* Now do the data that needs to be multiplied by a factor */
  for (i = 0; i < k; i++) {
    if (matrix_row[i] != 0 && matrix_row[i] != 1) {
      sptr = data_ptrs[i];
      switch (w) {
        case 8:  galois_w08_region_multiply(sptr, matrix_row[i], size, dptr, init); break;
        case 16: galois_w16_region_multiply(sptr, matrix_row[i], size, dptr, init); break;
        case 32: galois_w32_region_multiply(sptr, matrix_row[i], size, dptr, init); break;
      }
      init = 1;
    }
  }
}
```

図 6.1: Jerasure Encode プログラムコード

この関数は以下のような動作でエンコード演算を行っている。

for(i=0;i < m;i++){ for(j=0;j < k;j++){

data[j](sptr) と coding matrix[i*k+j]、これまでに演算した codingblock(dptr) から以下の条件を元に XOR 演算し、新たな dptr を生成。

if(coding matrix[i*k+j]==1)

galois_region_xor() により XOR 演算 // sptr ⊕ dptr

if(coding matrix[i*k+j]!=0,1)

galois_w**_region_multiply() により XOR 演算

// sptr ⊕ dptr × coding matrix[i*k+j]

} }

このJerasureのm,kのループに着目し、このループをアンロールして、data[j],coding matrix[i*k+j],codingblockを引数にそれらのエンコード演算を行い新たなcoding-blockを生成する関数を作ることでこれを並列化用のライブラリ関数とした。

6.2 並列パイプライン通信の実装

次に各デーモン同士のコネクションを張り、テキストベースの命令やバイナリベースのcodingblockの送受信を行う通信部分の実装について説明する。

6.2.1 selectを用いた複数のTCPソケット通信

通常のsocketを用いたTCP通信では、本実装にも用いたrecv()などはデータが受信できるまでブロッキングする。
確かにソケットを1つしか利用しない場合はブロッキングは有効だが、しかしそのソケットを複数扱うときはこれは難しい。例えば、片方のソケットでブロッキングしたままになっていると、他のソケットにデータが到着してもその受信が出来なくなってしまう。そのため、このように複数のソケットを今回の実装のように扱うためにはどのソケットに対しての受信なのかといった情報が必要である。

そこで本実装にはC言語のselect関数を用いる。
selectを使用すると、複数のファイルディスクリプタを監視し、変化があったファイルディスクリプタについて処理を行うことができるので、本実装の複数ノード間の入出力に対応できることからこの関数を用いることにした。

```
int select(int n,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
           struct timeval *timeout);
```

図 6.2: select関数

selectの引数には、fd_setというファイルディスクリプタの集合がある。そして監視する状態(読み込み, 書き込み, エラー)によって,*readfds,*writefds,*exceptfdsを設定する。nは, これら3つのファイルディスクリプタ集合のうち, 最大の「値」

(個数ではない)+1 を指定する。fd_set は次のような幾つかのマクロで設定、制御を行う。

- FD_CLR(int fd, fd_set *set);
  set から fd を削除
- FD_ISSET(int fd, fd_set *set);
  set に fd がセットされていれば 1 を返す
- FD_SET(int fd, fd_set *set);
  set に fd をセット
- FD_ZERO(fd_set *set);
  set を空に初期化

select が返ってくると、ファイルディスクリプタ集合には変化のあったファイルディスクリプタのみが格納されるので、それらについて何らかの処理を行う。

本実装は data ノードから coding ノードへの一方向な通信であるため、*writefds,*excerptfds は用いず、*readfds についてのみ select で監視を行うようにした。

また、select を通ったファイルディスクリプタを処理の違いで区別して以下のような種類分けを行った。

- opefd(operation file discripter)
  テキスト形式の命令通信用のファイルディスクリプタであり、
  受け取った命令内容からデーモンの状態制御を扱うために用いる。
- datafd(data file discripter)
  バイナリ形式のデータ通信用のファイルディスクリプタであり、
  エンコード途中の codingblock を通信するために用いる。
- consfd(console file discripter)
  コンソールプロセス専用のファイルディスクリプタであり、
  ユーザーがコンソールから直接入力した命令を通信するために用いる。

これら 3 つのファイルディスクリプタの識別は接続が確立した後に、handler() 関数に引数として渡され、更にその中で newconn() 関数に渡される。

newconn() 関数はそのファイルディスクリプタを通って送信側から最初に送られてくる識別命令 (operation,data,console) によってファイルディスクリプタを上のように識別する。

識別されたファイルディスクリプタは更にその handler() 関数内で、その種類ごとに以下のような別の関数に渡される。

- ope(int fds, fd_set *next_rfds)
  - opefd に対しての処理関数
  - 送信側から送られた命令を元に encode を行うかどうかなどを表すデーモンの内部変数を操作する。
- data(int fds, fd_set *next_rfds)
  - datafd に対しての処理関数
  - デーモンの内部変数を見て encode 可能なときに送信側から送られた codingblock とローカルの data から encode 演算して new codingblock を生成。
  - opefd を用いて new codingblock を送る側のデーモンを encode 可能な状態にし、new codingblock を送信。
- console(int fds, fd_set *next_rfds)
  - consfd に対しての処理関数
  - ユーザーコマンドを元にデーモンが立ち上がっている 1 番目のノードにジョブを与える。

6.3 チェックポイントライブラリの選択と並列擬似インクリメンタルチェックポイント

チェックポイントをとるライブラリにはその実装を行うレベルに関して大きく分けて次の 2 種類があり、それぞれにトレードオフがある。

- カーネルレベルチェックポイント (BLCR,CRAK,chpox など)
  - カーネル自体にチェックポイントの機能を埋め込んだもの。

  利点 カーネル内に存在するプロセスの内部状態を自由に取得可能で実装が容易。

  欠点 移植性がない。
- ユーザーレベルチェックポイント (CKPT,Libckpt,Condor など)
  - システムコールなどを利用したチェックポイント。

利点 移植性がある程度保たれる。

欠点 プロセスの内部状態を取得しづらく、実装が難しい。

このように様々なチェックポイントライブラリがある中で、本研究ではカーネルレベルチェックポイントとして実装されている BLCR を用いる。次の小節では BLCR の特徴を説明すると共になぜこのライブラリを選択したのかということについて説明する。

BLCR の選択とその特徴

BLCR(Berkeley Lab Checkpoint/Restart) はカーネルモジュールとしてデザインされたチェックポイントライブラリである。カーネル中に実装することで、ユーザー側で修正不可能なデータやカーネル内のイメージをミラー化するライブラリ呼び出しを容易に行うことができるので、チェックポイントする上で取得するのが難しいデータへのアクセスが可能となる。従って、その実装やユーザーによるチェックポイントの実行が比較的シンプルなものとなっている。

例えば、一秒ごとにカウントを行う count.c プログラムを BLCR でチェックポイントしたい場合は、まず以下のように BLCR ライブラリとリンク付してコンパイルする。

$$gcc\ -o\, count\, count.c\ -LBLCR\_LIBDIR\ -lcr\_run\ -u\ -cr\_run\_link\_m \quad (6.1)$$

そしてコンパイルされて生成された count.o を cr_run というコマンドを用いて通常のプログラムと同じように実行を行う。

$$cr\_run\ ./count.o \quad (6.2)$$

仮にこのときの実行プロセス ID(pid) が 10610 だったとすると、このプロセスに対して BLCR は以下のようにチェックポイントを取ることができる。

$$cr\_checkpoint\ 10610 \quad (6.3)$$

すると、デフォルトでは context.10610 というチェックポイントファイルがカレントディレクトリにダンプされる。
仮にこのプロセスが途中で fail してしまったと仮定して、そのチェックポイントを取った部分からプロセスを再開するには以下のように行う。

cr_restart context.10610 (6.4)

このようにBLCRは比較的シンプルな操作でチェックポイント/リスタートが可能である。

本研究ではインクリメンタルチェックポイントを対象として、元々それを備えているユーザレベルチェックポイントのLibckptを扱うことも可能ではあったが、Libckptのインクリメンタルチェックポイントはmprotectをシステムコールすることにより対象チェックポイントのファイルのプロテクションをread-onlyとし、そのページにアクセスが起こったときにsigsegvを検出して更新位置を判断するという手法が用いられているので、これはプログラム本体の実行時間に多大な影響を与えるということが分かっているので、次の小節で説明するように、BLCRにLINUXのxdeltaを用いたBLCR-xdeltaによる擬似インクメンタルチェックポイントを用いる。

BLCR-xdeltaによる擬似インクリメンタルチェックポイント

BLCRにはインクリメンタルチェックポイントは実装されていないので、古いチェックポイントファイルと最新のチェックポイントファイルに対して差分を比較を行うプログラムが必要となる。

今回はLINUXのプログラムxdeltaを用いた。これはバイナリファイルの差分をとるbinarry diffプログラムの一種でAndrew Trigellにより開発されたrsyncのアルゴリズムに基づいてる。ただ差分を取るだけでなく、差分したファイルの圧縮なども行うことができる。

このxdeltaを用いることで、２つの新旧チェックポイントファイルの変更部分について差分を取り、それを新たなチェックポイントとして保存することで擬似的なインクリメンタルチェックポイントとした。

これらのdiffプログラムによって差分をとる方法はLibckptのインクリメンタルチェックポイントと比べると、チェックポイント自体は取れているので差分をとるタイミングをずらすことができ、またチェックポイントの一貫性に影響しないのでプログラム本体の実行時間に影響を与えないといった理由から柔軟な差分作成方法といえる。

よって例えば10回普通にチェックポイントを取った後に1回だけdiffを用いて差分を作成するというふうにすれば、チェックポイント10回分の時間をかけて差分作成の計算を行うことができる。

gxp を用いたジョブの並列発行

インクリメンタルチェックポイントの並列化には GXP タスクスケジューラーを用いる。このスケジューラーを実行するノードで gxp を起動し、そこで登録した複数ノードに対して特定ファイルに記載したジョブを一斉に投げることができる。

6.4 測定方法

6.4.1 エンコード時間の測定

Erasure Coding デーモン部分の測定にはその実装に合わせて C 言語の gettimeofday() をエンコード演算+転送を行う前後に挿入し、その差を各ノードのエンコード時間として定めた。また、最初のノードとエンコード開始時間と最後のコーディングノードのエンコードのローカルセーブ終了時間の差を総エンコード時間と定めた。

6.4.2 チェックポイント時間の測定

BLCR もチェックポイントを取る部分は同じく C で実装されているので、gettimeofday を前後に挟みその差分をチェックポイント自体にかかる時間として定めた。また、チェックポイントと diff を取る命令の前後の時間を UNIX の date コマンドにより記録しておき、その差分時間をチェックポイントと diff などを含んだ総チェックポイント時間とし、これと先程のチェックポイント自体の時間の差から diff とその他にかかる時間を導出することができる。

第7章　Incremental Checkpoint with Erasure Codingの評価

7.1　比較項目・実験環境

7.2　予備実験

まず、逐次ノードを対象に予備実験を行う。
提案の実行例で示したjerasure_05を元にk,m,w,sizeといった値を変化させて実行した結果を以下で示す。

7.2.1　データ量を変化

下表7.1はm=2,w=32,size=128に固定してkを変化させたときの結果を表している。データ量はkとsizeから求められるencodeの対象となるデータ容量、dmmake_tはDMを作成するのに要する時間、encoding_tはデータ量のデータに対して2つのcodingデータを生成するのに要した時間を表しており、それぞれプログラム中にCのgettimeofday()を間に挟むことで計測を行った。

また、図7.1は上の表から縦軸k, 横軸encode_tとして描いたグラフである。
この図から分かる通り、kに対して、encode時間は比例的に増えていくことが分かる。

また、k=1000,m=2,w=32に固定にして、sizeを変化させた場合の結果を下表7.2で示しており、下図7.2ではsizeとencode_tのグラフを表している。
kはデータデバイス数を表すのに対して、sizeは各デバイスが保持するデータ量を表している。グラフからsizeに対しても、encode時間は比例的に増えていくことが分かる。

表 7.1: 逐次ノードで k を変化させて Jerasure を実行

k	データ量	dmmake_t	encoding_t
10^2	12.5KB	0.445s	0.002s
10^3	125KB	0.585s	0.055s
10^4	1.2MB	0.717s	0.272s
10^5	12.2MB	3.375s	2.331s
10^6	122MB	31.516s	24.283s
$5 * 10^6$	610MB	2m40.80s	2m24.14s
10^7	1.2GB	5m25.77s	6m38.53s

表 7.2: 逐次ノードで size を変化させて Jerasure を実行

size	データ量	dmmake_t	encoding_t
128	125KB	0.467s	0.024s
256	250KB	0.469s	0.045s
512	500KB	0.470s	0.074s
4096	3.9MB	0.469s	0.472s
16384	15.6MB	0.469s	1.844s
65536	62.5MB	0.470s	7.362s
131072	125MB	0.587s	18.92s
262144	250MB	0.467s	28.85s
524288	500MB	0.470s	57.74s
786432	750MB	0.469s	86.96s
1048576	0.98GB	0.467s	1m55.73s

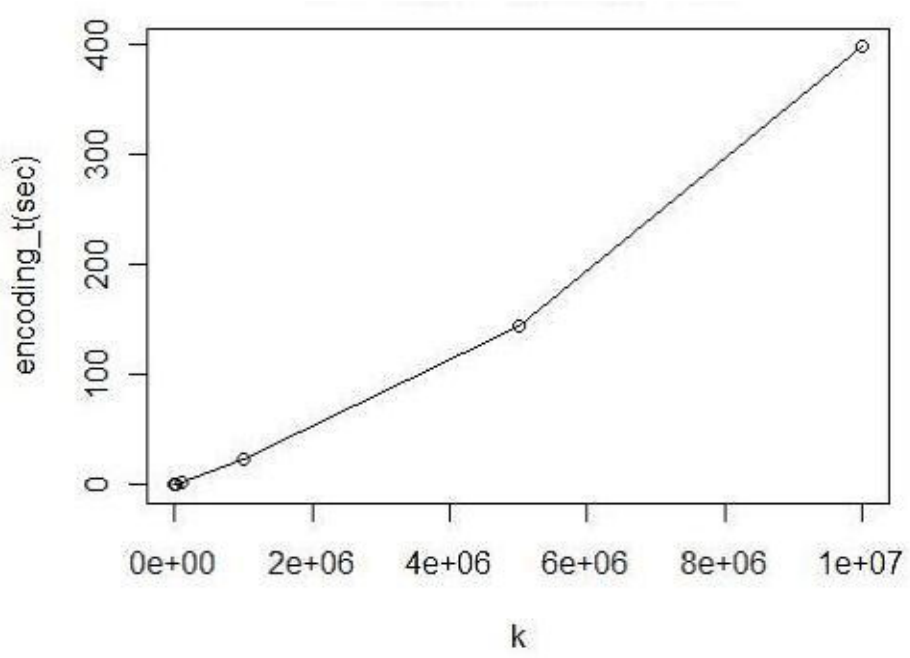


図 7.1: k と encode_t のグラフ

表 7.3: 逐次ノードで m を変化させて Jerasure を実行

m	coding データ量	dmmake_t	encoding_t
10	1.25KB	0.580s	0.121s
10^2	12.5KB	1.763s	1.406s
10^3	125KB	14.129s	11.939s
10^4	1.2MB	2m25.30s	2m21.39s

7.2.2 m を変化

次に、k=1000,size=128,w=32 に固定して m を変化させた場合の結果を下表 7.3 で示す。

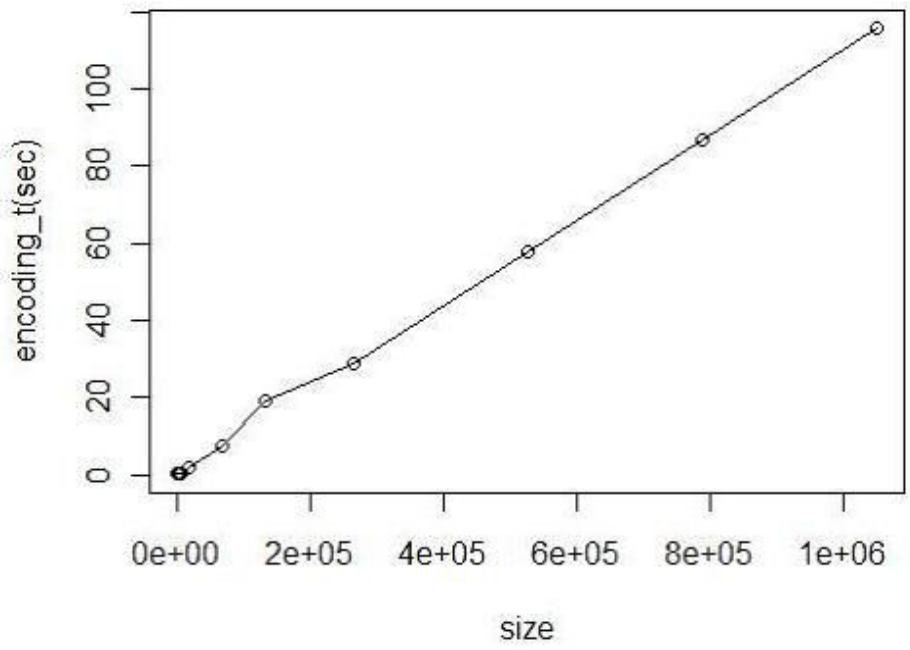


図 7.2: size と encode_t のグラフ

7.3 本実験

7.3.1 実験シナリオと測定内容

Incremental Checkpoint と Erasure Coding 併用すると、どれくらいの性能向上が見られるかを調べるために、「Normal　Checkpoint with Erasure Coding」と「Incremental Checkpoint with Erasure Coding」について性能比較を行う。

測定は Checkpoint 部分と Encode 部分をそれぞれ独立して測定する。

Checkpoint 部分は data32 ノード,coding2 ノードと固定し、まず、それぞれチェックポイントと diff のサイズがどのように増加していくのかを測定し、それを取るのに掛かった時間を同時に測定する。

Encode 部分は上で取った checkpoint と diff に対してデーモンを用いてエンコードを行い、それぞれのエンコード時間を測定する。また、data ノード数を 4,8,16,32 と変化させたときにどのようにその時間が変化するのかを測定する。

7.3.2 チェックポイント対象プログラム

Libckpt[3] の論文を参考に Incremental Checkpoint に対する評価がしやすいよう、以下のような 2 つのプログラムに対してチェックポイントを行った。

- MAT(行列掛け算)
  - N × N 行列 A,B を掛け算して行列 C を求める
  - N=5500 で 1 回あたりのチェックポイントサイズ:502.0MB
  - 単位時間当たりのメモリ変更箇所が少ないプログラム
- NPB LU(流体計算を SOR で解く)
  - 連立一次方程式 Ax=b を解く
  - NPB(Nas Parallel Benchmark) の NPB LU をシングルプロセスで実行。
  - 問題サイズを 146 × 146 × 146 とした。1 回あたりのチェックポイントサイズ:559.5MB
  - 単位時間当たりのメモリ変更箇所が多いプログラム

7.3.3 BLCR+diffにおける問題点の克服

今回は Incremental Checkpoint を実装した適当なチェックポイントライブラリが見つからなかったので、BLCR で取ったチェックポイントを xdelta を用いて差分を取ることで、これを擬似インクリメンタルチェックポイントとした。

ところが、通常の HDD 上で差分を取ろうとすると、既に HDD 上にある前後の２つチェックポイントファイルの読み込みにより一度の書き込みしか行わない通常のチェックポイントやインクリメンタルチェックポイントに比べて大幅に時間を要してしまう。

そこで本実験では、HDD よりも高速に読み書きが出来る RAM ディスク上にまずフルのチェックポイントを保存し、それらのファイルから xdelta により差分ファイルを HDD に書き出すという手法を用いた。

7.4 測定環境

以下の環境で測定を行った。

表 7.4: PrestoIII クラスタ

CPU	Opteron Processor242 1.6GHz × 2
Memory	SDRAM 2GB
OS	Linux 2.6.18-6 64bit(Debian base)
Network Adapter	Ethernet 1Gbps
Compiler	GCC 4.1.2
HDD I/O 性能 (bonnie++を用い測定)	Read:54MB/s,Write:34MB/s
RAM I/O 性能 (bonnie++を用い測定)	Read:1345MB/s,Write:728MB/s

7.5 測定

7.5.1 チェックポイント

MAT

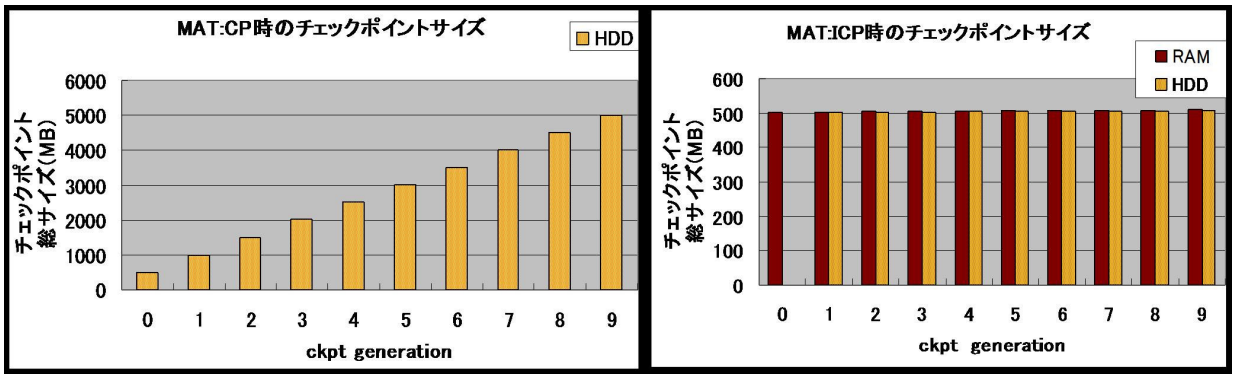


図 7.3: MAT:Normal Checkpoint/Incremental Checkpoint 時のチェックポイントサイズ

図 7.3 は従来手法 (CP) と提案手法 (ICP) を行ったときの、その保存先ディレクトリサイズがどのように増加していったのかを表している。通常のチェックポイントを用いると、チェックポイント保存先のディレクトリが平均 502MB ずつ大きくなっていくのに対して、擬似インクメンタルチェックポイントを用いると、平均で 0.48MB ずつの増加となり、99.9%のデータ削減となっている。

図 7.4, 表 7.5 はチェックポイントサイズとそれを作成する時間をチェックポイント自体とそれ以外の時間に区別して比較している。ckpt time(チェックポイント時間) は CP は HDD 上に ICP は RAM 上にチェックポイントファイルを作成するのにかかった時間を表している。通常チェックポイントは HDD 上に、インクメンタルチェックポイントはより高速に書き込める RAM 上に保存しているため、後者の

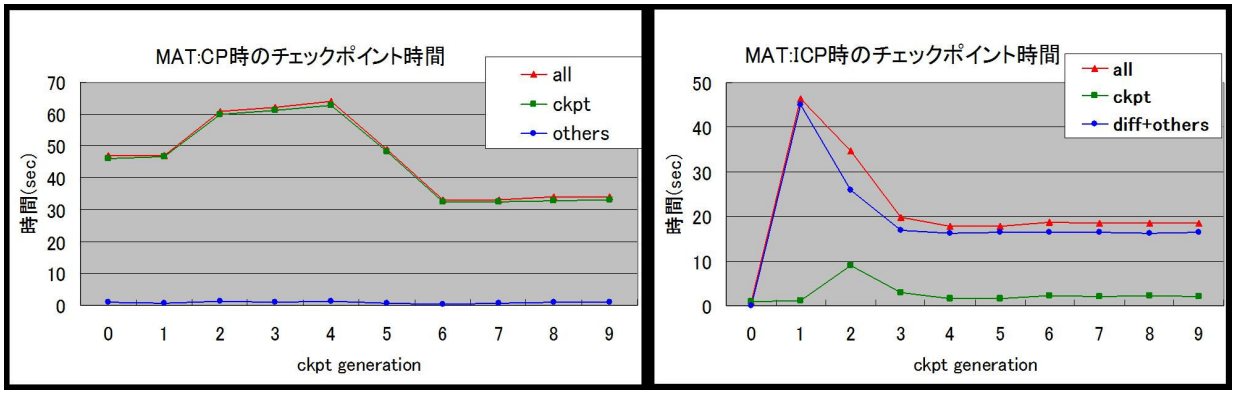


図 7.4: MAT:Normal Checkpoint/Incremental Checkpoint 時のチェックポイント時間

表 7.5: チェックポイントサイズと時間:MAT

	CP	ICP
平均サイズ	502.20MB	0.48MB
ckpt time	45.6sec(on HDD)	2.7sec(on RAM)
diff+others	0.8sec	20.7sec
all time	46.4sec	23.4sec

方がチェックポイントをとる時間自体は 2.5 倍早い。diff time は diff 時間を含めたチェックポイント以外の面でファイル生成にそれぞれ要した時間を表している。CP の 0.1sec はチェックポイント実行スクリプトによるタイムラグである。 これを想定すると、ICP の diff には 73.2sec ほどかかっていることが分かる。

NPB LU

図 7.5 は従来手法 (CP) と提案手法 (ICP) を行ったときの、その保存先ディレクトリサイズがどのように増加していったのかを表している。MAT と比べると NPB LU は単位時間あたりのプロセスイメージの変更箇所が大きいため、擬似インクリメンタルチェックポイントを用いても平均 335.4MB ほどずつ増加していき、データ削減率は 40.0%となり MAT より性能は低い。

図 7.6, 表 7.6 はチェックポイント時間のついて MAT 同様の比較をしている MAT に比べると提案手法の側の diff サイズがかなり大きいが、RAM への読み書きが高速なために、その割には MAT とほぼ変わらない時間で diff ファイルを作成できていることが分かる。

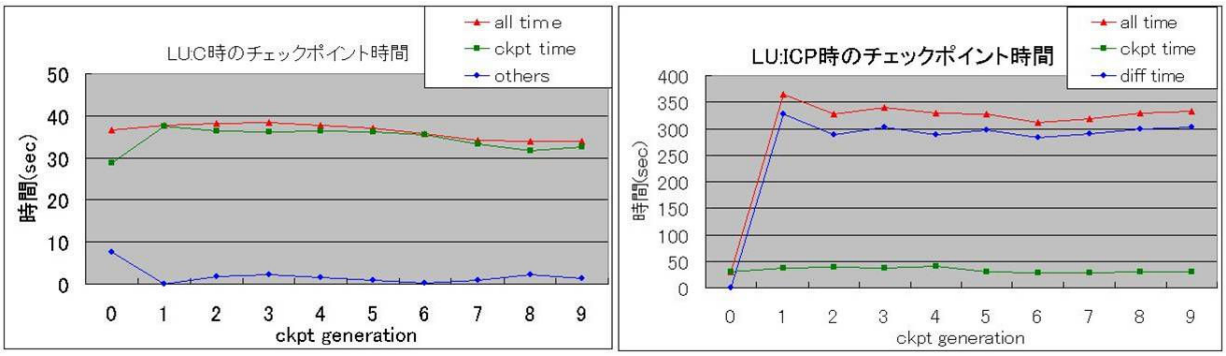


図 7.5: NPB LU:Normal Checkpoint/Incremental Checkpoint 時のチェックポイントサイズ

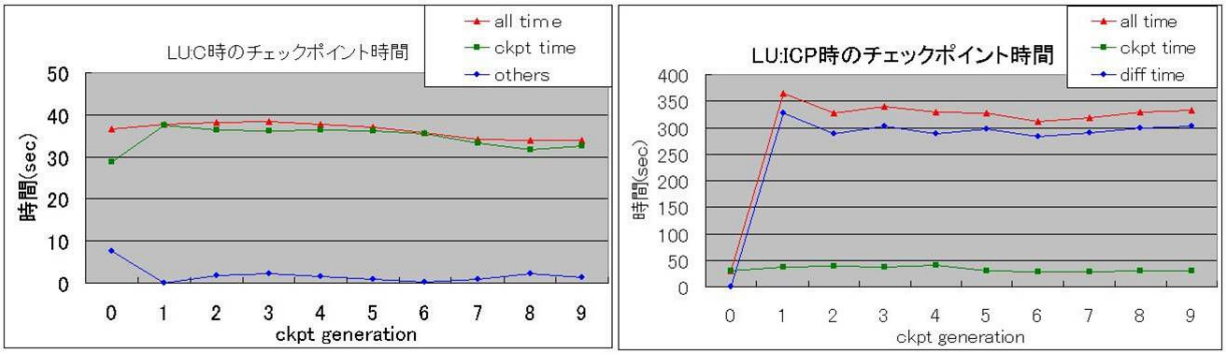


図 7.6: NPB LU:Normal Checkpoint/Incremental Checkpoint 時のチェックポイント時間

7.5.2 エンコード

以下ではフルのチェックポイントを ckpt original、チェックポイント差分ファイルを diff と呼ぶ。

ckpt original,diff についてデーモンを用いてエンコードを行い、その coding ノードまでの転送と保存を含めた時間を測定した。

コーディングノード数 m は 2 に固定し、size(一度に encode/転送行う単位サイズ)は MB 単位のデータに関しては、は 64KB,KB 単位のデータ (MAT の diff) は 8KB に固定した。

以下ではそれぞれ保持するチェックポイントサイズは固定のまま、data ノード数を 4,8,16,32 というふうに変化させたときの比較を行っている (weak scaling)。

表 7.6: チェックポイントサイズと時間:NPB LU

	CP	ICP
平均サイズ	559.5MB	335.4MB
ckpt time(gene1)	48.3sec(on HDD)	2.6sec(on RAM)
diff(+others)	0.1sec	41.8sec
all time	48.4sec	44.4sec

MAT

MAT に関しては図 7.7, 表 7.7 のような測定結果になった。
data ノード数 32 の場合に注目すると、1 つあたりの平均 502.0MB の ckpt original の encode に 101.1sec、データ削減率 99.9%の平均 0.48MB の diff の encode に 0.46 秒かかっており、99.5%の Encode 高速化を実現した。

また、ノード数別に比較を行うと、どちらの場合も 4 ノードと少ないときにより高速にエンコードが行えて、それ以上にノード数を増やしたときはエンコード時間は少しずつ大きくなっていくという結果が得られた。これは並列パイプライン通信で実装した Parallel Erasure Coding デーモンの特徴と言える部分である。また、CP,ICP いずれの場合も各ノードの保持するチェックポイントファイルサイズ一定なので、ノード数に関わらず冗長化にかかるコスト (encode サイズ) は、それぞれほぼ一定の値となる。

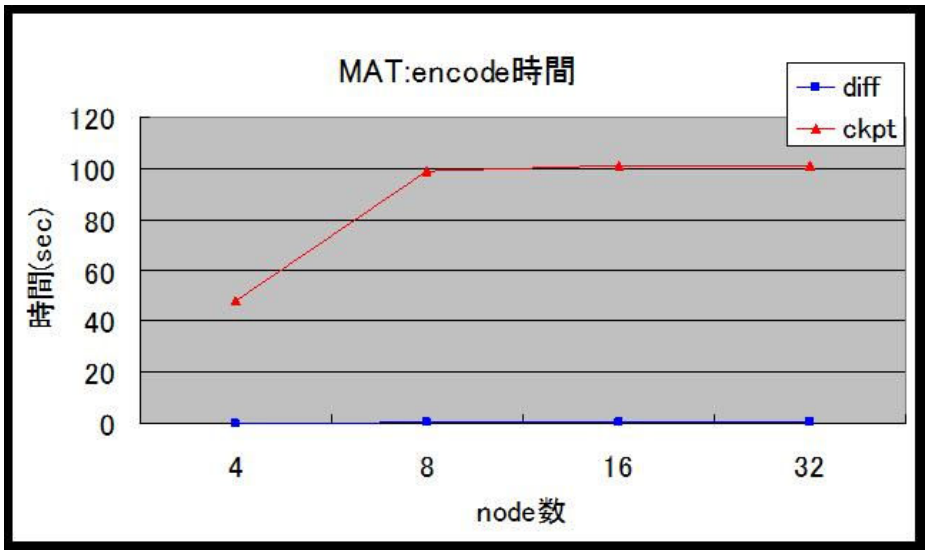


図 7.7: MAT:Normal Checkpoint v.s. Incremental Checkpoint の encode 時間

表 7.7: data ノード数別 encode 時間:MAT

	ckpt original	diff
4 ノード		
ckpt 総サイズ	1.96GB	3.43MB
encode サイズ	501.2MB	552.0KB
encode 総時間	47.9s	0.05s
8 ノード		
ckpt 総サイズ	3.92GB	6.46MB
encode サイズ	501.2MB	552.0KB
encode 総時間	1m39.0s	0.36s
16 ノード		
ckpt 総サイズ	7.84GB	13.12MB
encode サイズ	501.2MB	557.0KB
encode 総時間	1m40.8s	0.46s
32 ノード		
ckpt 総サイズ	15.69GB	26.56MB
encode サイズ	501.2MB	557.0KB
encode 総時間	1m41.1s	0.46s

NPB LU

一方、NPB LU に関しては図 7.8, 表 7.8 のような測定結果になった。
同じく data ノード数 32 の場合に注目すると、1 つあたりの平均 559.5MB の ckpt original の encode に 114.3sec、データ削減率 40%の平均 335.4MB の diff の encode に 72.1sec かかっており、プログラムの特徴から diff サイズの関係で MAT には大幅に劣るが、36.9%の高速化を実現した。

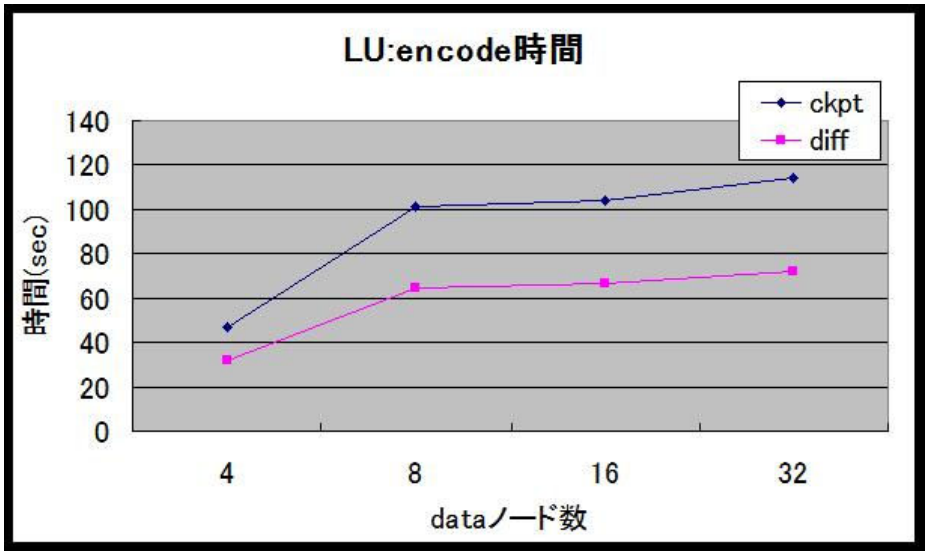


図 7.8: NPB LU:Normal Checkpoint/Incremental Checkpoint の encode 時間

7.5.3 総合評価

最後にチェックポイントからエンコードまでの時間比較し、これを総合評価とする。
今回の実験ではそれぞれの実行を独立して行い、その時間を測定しているので、実際にそれらを併用して用いる場合は、CPU や MEM の負荷などの理由から幾分か以下の結果より時間がかかると予想される。しかし、通常チェックポイントとインクリメンタルチェックポイントの比較評価を行う上では条件は等しいので以下の結果で十分な評価ができていると考えられる。

MAT

MAT の場合の総合評価を図 7.9, 表 7.9 に表す。提案手法は diff に時間がかかる為、チェックポイントを行った地点では、平均で従来手法 (CP) は 46.4sec, 提案手法 (ICP) は 23.4sec といった時間を要するが、diff によりデータを 99.9%も収縮できる為に、これらを encode すると、CP の 101.1sec に対して ICP は 0.46sec といった時間しかかからない。従って、総合時間は CP:147.5sec,ICP:23.9sec となり、提案手法 (ICP) の方が全体で 83.8%の高速化を実現した。

表 7.8: data ノード数別 encode 時間:NPB LU

	ckpt original	diff
4 ノード		
ckpt 総サイズ	2.06GB	1.30GB
encode サイズ	526.4MB	338.1MB
encode 総時間	47.1s	31.7s
8 ノード		
ckpt 総サイズ	4.12GB	2.62GB
encode サイズ	526.4MB	338.2MB
encode 総時間	1m41.5s	1m4.8s
16 ノード		
ckpt 総サイズ	8.23GB	5.23GB
encode サイズ	526.4MB	338.2MB
encode 総時間	1m43.8s	1m6.6s
32 ノード		
ckpt 総サイズ	16.45GB	10.48GB
encode サイズ	526.4MB	338.2MB
encode 総時間	1m54.3s	1m12.1s

NPB LU

NPB LU の場合の総合時間の評価を図 7.10, 表 7.10 に示す。NPB LU は MAT に比べるとチェックポイント間の変更箇所が多いので、擬似インクリメンタルチェックポイントを適用しても 40%ほどしかサイズを削減できない。しかし総合時間を見てみると、従来手法よりも全体で 28.4%の高速化を実現した。

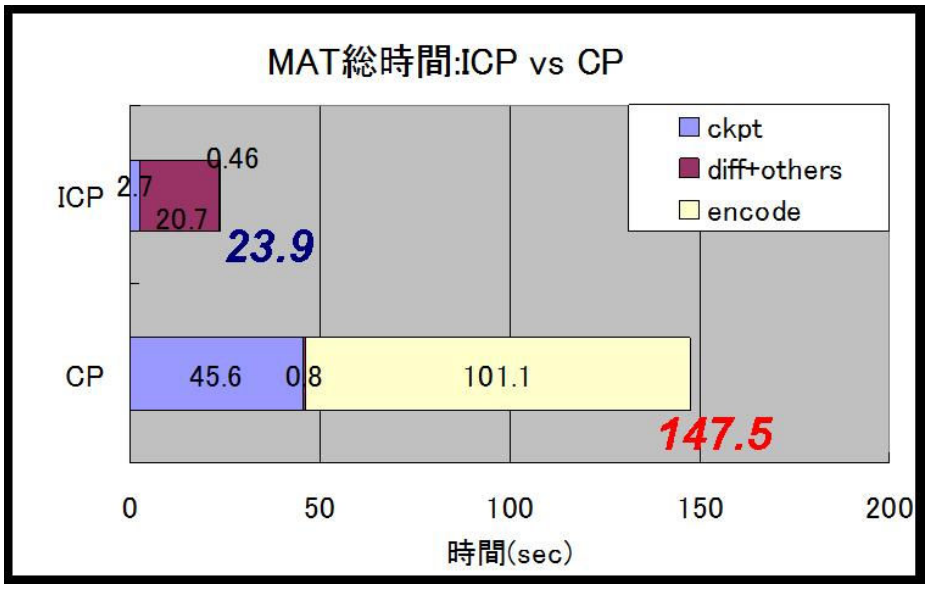


図 7.9: MAT:Normal Checkpoint/Incremental Checkpoint の ckpt+encode 時間

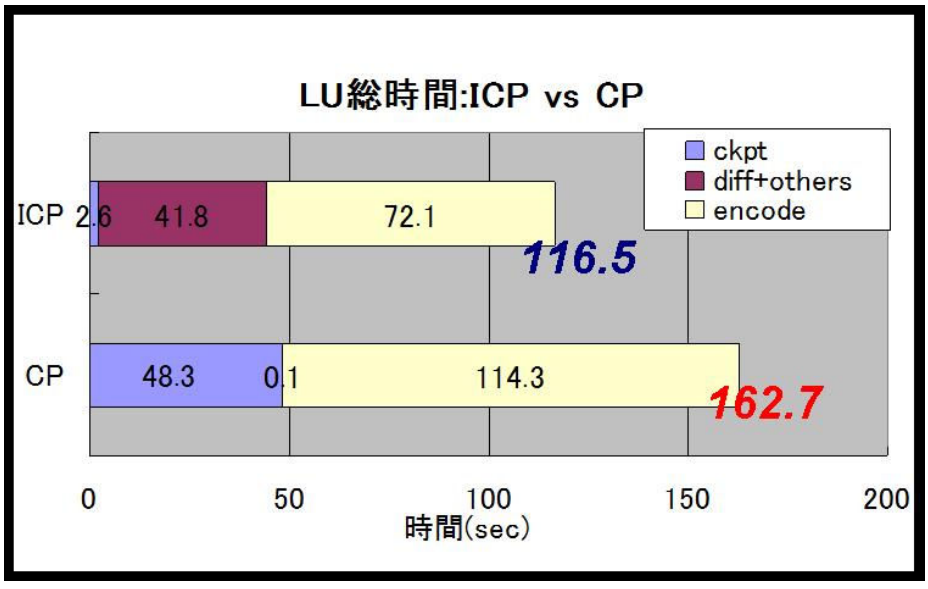


図 7.10: NPB LU:Normal Checkpoint/Incremental Checkpoint の ckpt+encode 時間

表 7.9: ckpt+encode:MAT

(sec)	CP	ICP
ckpt time	45.6(on HDD)	2.7(on RAM)
diff+others	0.8	20.7
ckpt all	46.4	23.4
encode	101.1	0.46
all	147.5	23.9

表 7.10: ckpt+encode:NPB LU

(sec)	CP	ICP
ckpt time	48.3(on HDD)	2.6(on RAM)
diff+others	0.1	41.8
ckpt all	48.4	44.4
encode	114.3	72.1
all	162.7	116.5

第8章　結論

8.1　まとめ

本研究では近年の大規模分散システムの耐故障性へのチェックポイント時間の削減、冗長化の必要性といった需要に対して、従来からあるIncremental CheckpointとErasure Codingという技術を併用して用いることを提案し、更にそれをノード増加に対応したパイプライン並列化によって実装することで冗長化とチェックポイント時間の削減を試みた。

結果、それらの技術は問題無く組み合わせることができ、チェックポイント対象プログラムがある区間におけるプロセスイメージの変更点が少ないプログラムから変更点が多いプログラムであっても、従来のチェックポイントとその転送にかかる時間を削減し、うまく高速化を図ることができた。

8.2　今後の課題

今後の課題としては、まず大規模環境での実験によりスケーラビリティの評価を行う。また、チェックポイント対象プロセスの変更点がどれくらいまでのものに対して本技術が有効なのかということは本研究で示すことができなかったので今後比較していかなければならない。また、今回は並列Erasure Codingをパイプライン通信を用いたデーモンとして実装したが、他の転送方式、broadcast/allgatherやfan-inといった方法を用いることで、スケーリングの観点からどのような違いが出るのかということを調査する必要がある。最後に今回はエンコード部分についてのみ、実装と性能比較を行ったが、デコード部分についても実装とその性能を比較する必要がある。デコード部分はエンコード部分と同様に従来の逐次ライブラリを細粒化・並列化し、生存するノード内からk個のデコード用ノードを選択し、それらに対してErasure Codingデーモンをコネクションを張り、デコードブロックの演算と転送を行い、最終的に新しいデータノード上に転送するといった方式で実装が可能だと考えられる。また、多重故障からの復元といった点についても言及が必要とされる。これらが今後の課題といえる。

謝辞

本論文を執筆するにあたり多くの助言と指針を示していただいた指導教員の松岡聡教授に感謝いたします。

お忙しい中、ゼミなど広くに渡り指導していただいた遠藤敏夫准教授に感謝いたします。

また研究室での指導に加え、スライド作りや発表練習にお付き合いいただいた實本英之氏をはじめ松岡研究室の皆様に感謝いたします。

最後に共に学士論文に取り組んだ松岡研究室の仲間や友人や家族に感謝いたします。

参考文献

[1] Marcos K. Aguilera, Ramaprabhu Janakiraman, and Lihao Xu. Using erasure codes efficiently for storage in a distributed system. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings*, pp. 336–345, 2005.

[2] Condor project homepage. http://www.cs.wisc.edu/condor.

[3] James S. Plank, Micah Beck, and Gerry Kingsley. Libckpt: Transparent checkpointing under unix. In *Conference Proceedings, Usenix Winter 1995 Technical Conference, New Orleans, LA, January*, pp. 213–223, 1995.

[4] D.B.Johnson, E.N.Elnozahy, and W.Zwaenepoel. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems*, pp. 39–47, 1992.

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

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

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

[8] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. In *IEEE Transactions on Parallel and Distributed Systems*, 1998.

[9] Luis M. Silva and Joao Gabriel Silva. An experimental study about diskless checkpointing. In *Proceedings of the 24th Conference on EUROMICRO-Volume 1*, 1998.

[10] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson. Raid: High-performance, reliable secondary storage. In *ACM Computing Surveys, vol. 26, no. 2*, pp. 145–185, June 1994.

[11] James S. Plank. Improving the performance of coordinated checkpointers on networks of workstations using raid techniques. In *Proc. 15th Symp. Reliable Distributed Systems*, pp. 76–85, Oct. 1996.

[12] M. Blaum, J. Brady, J. Bruck, and J. Menon. Evenodd: An optimal scheme for tolerating double disk failures in raid architectures. In *Proc. 21st Ann. Int' l Symp. Computer Architecture*, pp. 245–254, Apr. 1994.

[13] James S. Plank. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. In *Software-Practice and Experience vol. 27, no. 9*, pp. 995–1012, Sept. 1997.

[14] Vera Pless. Introduction to the theory of error-correcting codes. In *Wiley Interscience*, 1998.

[15] S. Frolund et. al. A decentralized algorithm for erasure-coded virtual disks. In *Proceedings of DSN*, pp. 125–134, 2004.

[16] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Proceedings of DSN*, Dec. 2003.

[17] P.Corbett et. al. Row-diagonal parity for double disk failure correction. In *Proceedings of FAST*, 2004.

[18] James S. Plank. Jerasure: A library in c/c++ facilitating erasure coding for storage applications. In *Technical Report CS-07-603*, 2007.