Vol.2009-HPC-122 No.9
2009/10/9

増分データと Erasure Coding を利用した高速なチェックポイント手法

實本英之[†1]　中村俊介[†1]
遠藤敏夫[†1]　松岡聡[†1,†2]

チェックポイント／リスタート手法は多くの大規模 HPC システムで利用されている耐故障機能である。しかし、近年の大規模 HPC システムのメモリサイズの急速な増大に比べ、並列ファイルシステムの I/O 帯域の増大は相対的に低いため、チェックポイント時間が増加してしまう問題がある。本研究では、チェックポイントのオーバヘッドを低く抑えつつ多重故障に対応することを目的とし、Erasure Coding を採用する。処理のボトルネックを避けるために Erasure Coding の符号演算処理を並列化し、かつ並列ファイルシステムの代わりにノードのローカルストレージにプロセスイメージを格納する。さらにプロセスイメージの大きさを削減するために、Incremental Checkpoint 手法を採用する。この手法はチェックポイント時に前回のプロセスイメージとの差分部分を記憶するものである。並列環境において行列積演算および NPB LU ベンチマークを用いた実験の結果、Incremental Checkpoint を用いたときに 28-84%の性能向上を確認した。

The Efficient Checkpoint based on Erasure Coding with Incremental Method

Hideyuki Jitsumoto ,[†1] Syunsuke Nakamura ,[†1]
Toshio Endo [†1] and Satoshi Matsuoka[†1,†2]

Checkpointing/restarting is a well-known method as a fault tolerance mechanism in large scale HPC systems. However, overhead of this method tends to get larger, since memory size of recent systems is increasing rapidly, while the improvement of I/O bandwidth of file systems is relatively mild. The purpose of this work is to achieve checkpointing that supports multiple faults with low overhead by utilizing erasure coding. To eliminate the bottleneck, we parallelize encoding and store process images into node-local storage instead of shared file systems. Furthermore, to reduce sizes of process images, we adopt incremental checkpointing, which stores only parts of the process image that are modified since the previous checkpointing. Through parallel experiments using matrix multiply computation and NPB LU benchmark, we have observed 28 to 84% performance improvement by introducing incremental checkpointing.

表 1　TOP500[1)] 上位スパコンの性能比較

	ASCI Red (1997)	Jaguar (2008)	増加率
システムメモリサイズ	1.2TB	300TB	250 倍
並列ファイルシステム I/O 帯域	240MB/s	4GB/s	20 倍

1. はじめに

大規模 HPC 環境における要素数の増加に伴い、故障率も大きく増加しており、耐故障機能は必須要件となってきている。このため、耐故障機能として多くの大規模 HPC 環境で利用されているものの一つとしてチェックポイント／リスタートという手法が存在する。これはアプリケーションのスナップショットを定期的に保存（チェックポイント）し、故障時はスナップショットからの再開（リスタート）を行うことにより対応する手法である。しかし、急激な故障率の増加により、チェックポイント／リスタート手法には下記にあげるような対応が必要となっている。

多重故障への効率的な対応　一般的に多重故障に対応するためには、耐故障の対象とする全てのプロセスのイメージを冗長化し、安定なストレージに転送する必要がある。これを、時間的オーバヘッドとストレージ利用量の双方を抑制しつつ実現する必要がある。

チェックポイント時間の削減　格納すべきプロセスイメージの合計サイズは一般にシステムのメモリサイズに比例する。しかし近年の大規模 HPC システムにおけるメモリサイズの急速な増加に比べると、並列ファイルシステムの I/O 帯域の増加は少ないため (表 1)、全イメージを並列ファイルシステムに格納しようとするとそれに必要な時間は増加傾向にある。将来のシステムにおいて、このチェックポイント時間が MTBF(平均故障間隔) を超えてしまうと、チェックポイントは実用不可能になる。

多重故障への耐性を実現する手法として RAID5 技術に代表される Erasure Coding が

†1 東京工業大学
Tokyo Institute of Technology
†2 国立情報学研究所
National Institute of Informatics

存在する。これをチェックポイントに適用することにより、耐性確保に必要なリソースの増加を抑えることが可能である。しかし、誤り訂正符号の演算（以下エンコード）を行うために、各ノードのチェックポイントを転送する必要があり、結果チェックポイント時間の増大を引き起こしてしまう。メモリ量と I/O 帯域のアンバランスによるチェックポイント時間の増大及びエンコードのためのチェックポイント転送時間を削減するために、チェックポイント自体のサイズを削減する必要がある。本研究では、Erasure Coding をチェックポイントに適用したうえで、チェックポイントサイズを削減することにより、高速かつ多重故障に対応したチェックポイントを実現した。チェックポイントサイズの削減にはチェックポイント間の増分データのみを保存する Incremental Checkpoint を利用している。以下の構成は、2 節にて提案チェックポイント手法の設計について述べ、次の 3 節において、今回の評価で用いた実装について述べる。4 節にて従来手法と提案手法の比較評価を行い、5 節にて関連研究を述べたのちに、6 節でまとめを行う。

2. 設　　計

2.1 Erasure Coding によるデータ復旧

Erasure Coding は誤り訂正符号を利用した誤り制御機能の手法である。本手法をデータ保存に利用した例としては RAID5 等が挙げられ、その手順は大きく分けて、誤り訂正符号を作成するエンコードと喪失したデータを復旧するデコードから構成される。一般に k 個の元データに対して、数学的に直交関係にある誤り訂正符号を m 個作成した場合、$(k+m)$ 個のデータ内で m 個のデータ誤りまでを訂正することが可能である。これを行列ベクトル積を用いた表現で示すと、図 1 となる。左辺の $(k+m) \times k$ 行列は Distribution Matrix と呼ばれ、$k \times k$ の基本行列である Identity Matrix と、$m \times k$ の Coding Matrix から構成されている。Distribution Matrix の各行は数学的に直交関係でなければならない。本研究では誤り訂正符号を求める手法として Cauthy Reed-Solomon Coding(CRS Coding) を用いた。これはバーストな誤りに対して強い訂正能力を持つ手法で、データ保存の分野に適している。

2.2 並列チェックポイントへの Erasure Coding 適用

並列チェックポイントをナイーブな手段で行った場合、全てのチェックポイントを安定ストレージへ転送する必要がある。これにより、I/O ノードと計算ノード間での通信輻輳が起きやすく、チェックポイント時間が大きくなりやすい。そこで、チェックポイントを計算ノードローカルに保存し、故障時のチェックポイント喪失を防ぐために Erasure Coding を適用する。現存する多くの大規模 HPC 環境において、計算ノード群と I/O ノード群の間のバンド幅は、計算ノード数に対して十分ではない傾向にあるため、チェックポイント時間の高速化が期待できる。本研究では、失われる可能性のあるノード毎のチェックポイントファイルを図 1 におけるデータブロックにマッピングした。以降図 1 で示したデータブロック D_n をローカルチェックポイントファイル CP_n と示す。また、チェックポイントデータが作成される計算ノードをデータノード、誤り訂正符号の保存される計算ノードをコーディングノードと呼ぶ。

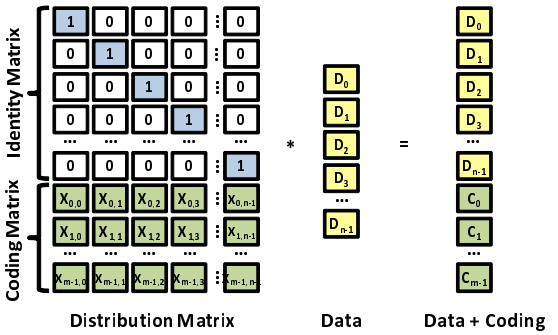

図 1　エンコードの行列ベクトル積表現

並列化 Erasure Coding について図 2 に示す。C_i の算出方法は

$$C_i = \sum_{n=0}^{k-1} X_{i,n} \times CP_n \tag{1}$$

である。つまり、ノード毎にチェックポイントが一つずつ存在し、環境が k ノードで構成されるとする場合、ノード j において以下の $C_{i,j}$ は通信をせずに算出することができる。

$$C_{i,j} = X_{i,j} \times CP_j \tag{2}$$

この $C_{i,j}$ をデータノード全てについて加算演算のリダクション操作をすることにより誤り訂正符号 C_i を算出することができる。

この C_i をコーディングノードに保存する。これによりデータノードが k ノード、コーディングノードが m ノード存在した場合、双方を含めた $k+m$ ノードから、m ノードまでの故障に対応することが可能になる。

Vol.2009-HPC-122 No.9
2009/10/9

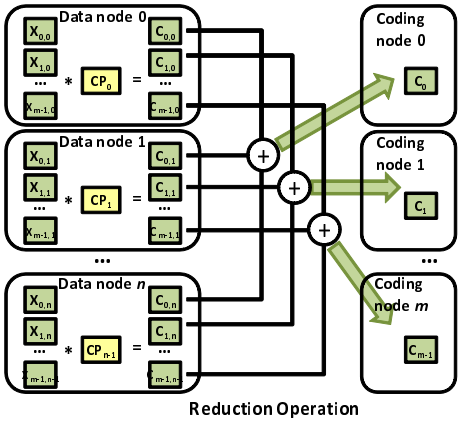

図 2 Parallel Erasure Coding の流れ

2.3 Erasure Coding ベース増分チェックポイント

Incremental Checkpoint はチェックポイントインターバル間の増分のみをチェックポイントする手法である。これにより、プロセスメモリの変更箇所が少ないプログラムにおいてチェックポイントのサイズを削減することができる。出力される増分の量はプロセスごとに違うため、Erasure Coding を適用するに当たり、対象とするプロセス群にある最大増分サイズに合わせてパディングを行うこととした。Incremental Checkpoint を利用した Erasure Coding ベースチェックポイントは以下の流れになる。

(1) あらかじめ、Coding Matrix は各データノードで共有しておく
(2) プログラム開始後、1 度目のチェックポイントではプロセスの全イメージが、2 度目以降では Incremental Checkpoint を利用し前回のチェックポイントとの増分イメージのみがローカルに書き出される。
(3) 各計算ノードにおいて、Coding Matrix とチェックポイントデータの乗算を行う
(4) データノード間で乗算結果に対し加算リダクション操作を行う
(5) 結果をコーディングノード上に配置する

3. 実　　装

実装に当たり、Erasure Coding には Jerasure[2] ライブラリを用いた。さらに、チェックポイントライブラリとしては BLCR[3] を用いている。

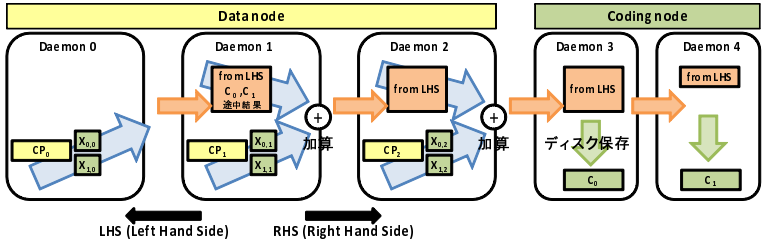

図 3 Pipeline Parallel Erasure Daemon

3.1 Parallel Erasure Daemon

2.3 を行う Parallel Erasure Daemon を図 3 のように実装した。Daemon は、計算ノード毎に 1 プロセス存在し、パイプライン転送を利用してエンコード演算を行う。これは以下の手順により行われる。

(1) 外部プロセスによりエンコード開始を通知される。この際、該当チェックポイントの存在するデータノード名、チェックポイントファイル名、転送ブロックサイズ、およびコーディングノード名も同時に通知される。
(2) 通知された情報を利用しパイプライン転送用の通信路を構築する。並行して以下のエンコードを開始する
(3) 先頭以外のデータノードでは 左隣ノードからエンコード演算結果を転送ブロックサイズ分受信する
(4) ノード内チェックポイントと Coding Matrix に対し式 2 の演算を行い、先頭データ以外のノードでは 左隣ノードからの受け取った結果を加算する。
(5) 前項目で計算したエンコード演算結果を 右隣ノードに転送する。ここまでの演算ステップはダブルバッファリングを利用し、通信と演算の重ね合わせを行っている。
(6) コーディングノードが左隣ノードから演算結果を受け取った場合、そのうち自分が担当する部分をファイルとして保存する。そしてそれ以外の演算結果を右隣のコーディングノードに転送する。
(7) 全ての誤り訂正符号がそろい次第、パイプライン転送用の通信路を閉じる。

本手法では、Parallel Erasure Daemon 自身の耐故障性のため、エンコードごとに通信路をリフレッシュする手法を採用している。加えて、転送形態もパイプラインにより行うため、構築が容易かつノードが故障した場合の再構築計算もほとんど起こらない。本来、Erasure

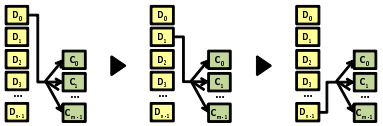

図 4 Broadcast 転送

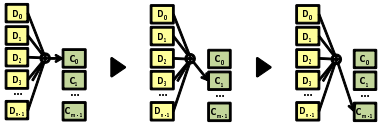

図 5 Fan-in 転送

Coding の並列化通信には以下の例にあげるように様々な手法がある[4),5)]。しかし本研究では 1)1 サイト内の大規模 HPC を対象とし通信遅延が少ない、2) 計算ノードは通信性能を含め均一になっているという仮定を置いた。前述した通信路の再構築および通信・演算の重ね合わせが比較的容易であることを考慮しパイプライン方式を採用した。

Broadcast データノードがそれぞれ、全てのコーディングノードにチェックポイント断片を送信する手法（図 4）。全てのデータがそろってからエンコード演算を行うため、演算自体の最適化が行いやすい。しかし、通信量が多く、データノード、コーディングノード間で輻輳が発生しやすい。

Fan-in データノードを Tree 構造に配置する。親ノードは子ノードから受け取ったデータを全てエンコード演算し、さらに上位の親に転送する手法（図 5）。通信路の最長パスが短くなるため帯域遅延積が小さく、転送時間の無駄がない。しかしネットワークの再構築がパイプラインより複雑になる。

3.2 Incremental Checkpoint のエミュレーション

Incremental Checkpoint を備えたチェックポインタ実装はほとんどない。 Libckpt[6)] という実装がこれに対応しているが、利用できる環境が限られているとともに、メンテナンスが終了しているため対応環境の増加も期待できない。そこで、今後理想的な増分チェックポインタ実装が現れると仮定し、以下の手法で通常のチェックポインタである BLCR によるエミュレーションを行った。

(1) BLCR により RAM ディスク上にチェックポイントを作成する
(2) 同様に事前に RAM ディスク上にとっておいた前世代チェックポイントと Binary Diff を行う (xdelta[7)] を利用)。

表 2 測定環境ノード

CPU	Opteron Processor 242 (1.6GHz) *2
Memory	SDRAM 2048MB
OS	Linux 2.6.18 (Debian base)
NIC	1Gbps (GbE)
HDD I/O (bonnie++)	Read:50MB/s,Write:33MB/s
RAM I/O (bonnie++)	Read:1345MB/s,Write:728MB/s

(3) 相違点をローカルディスク上のファイルに書き出す

4. 評価と考察

Incremental Checkpoint を利用することによる、Erasure Coding ベースチェックポイントへの影響を調査した。以下 Incremental Checkpoint を用いたものを ICP, 用いていないものを CP と記述する。測定には表 2 の性能を持つノードを最大 34 ノード用いた。これら全てのノードは 1 台のスイッチにより相互接続されている。

チェックポイント対象となるプログラムは以下の 2 つを用いた。

MAT 行列積を計算する自作プログラム。メモリの更新範囲が少なく、Incremental Checkpoint の効果が出やすい。1 プロセスのプロセスサイズは 501MB。

NPB LU NAS Parallel Benchmark[8)] の LU アルゴリズム。メモリの更新範囲が大きく、Incremental Checkpoint の効果が出にくい。1 プロセスのプロセスサイズは 526MB。

また、MPI をはじめとする並列分散システムに対し Incremental Checkpoint を実装するコストが大きいため、これらのプログラムの逐次版を各ノードで並列同時実行することにより、並列プログラムをエミュレーションしている。Parallel Erasure Daemon は各ノードに外部プログラムがチェックポイントを取っていることを前提としているので、このエミュレーションによる動作の変更は必要ない。

Parallel Erasure Daemon の利用するエンコードのブロックサイズは、実測から 64KB とした。ただし、Incremental Checkpoint により、データサイズが非常に小さくなってしまった場合 (MAT-ICP) のみ 8KB を利用している。

4.1 エンコード時間、チェックポイントサイズの比較

Incremental Checkpoint によるチェックポイントへの影響およびプログラムのメモリ使用法の基本的な性質を確認するために、データノードを 32 ノードおよびコーディングノードを 2 ノード利用し、チェックポイントのサイズ、誤り訂正符号のサイズ、およびエンコード時間を測定した。エンコード時間とは、エンコード開始を先頭データノードの Parallel

表 3 MAT チェックポイント・誤り訂正符号のサイズおよびエンコード時間

	CP	ICP	削減率
チェックポイントサイズ	15.69GB	15.36MB	99.9%
誤り訂正符号サイズ	501.2MB	557.0 KB	99.9%
エンコード時間	101.6sec	0.46sec	99.5%

表 4 NPB LU チェックポイント・誤り訂正符号のサイズおよびエンコード時間

	CP	ICP	削減率
チェックポイントサイズ	17.42GB	10.48GB	39.8%
誤り訂正符号サイズ	526.4MB	338.2MB	35.8%
エンコード時間	114.3sec	72.1sec	36.9%

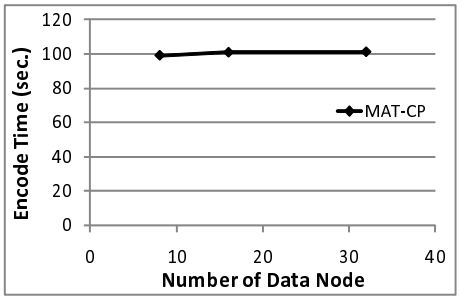

図 6 ノード数増加によるエンコード時間の変化

Erasure Daemon に要求してから、コーディングノードに誤り訂正符号を作成するまでの時間である。結果を表 3、4 に示す。MAT プログラムにおいては、Incremental Checkpoint の効果が大きく、チェックポイントサイズは 99.9%削減した。これに伴い、誤り訂正符号のサイズ、エンコード時間とも同程度に削減されている。NPB LU プログラムにおいても同様に、削減率は小さいがチェックポイントサイズが 39.8%削減し、それに伴い誤り訂正符号サイズ、エンコード時間とも同程度に削減している。

4.2 Parallel Erasure Daemon のスケーラビリティ

本実装ではパイプライン処理を行うことにより、ノード数増加によるチェックポイント時間の増加が最小限になるよう実装を行っている。これを確認するためにデータノードを 8, 16, 32 と変化させながら、コーディングノード 2 つに対し誤り訂正符号を作成する時間を測定した。本測定に当たっては Incremental Checkpoint を適用していない。測定プログラムは MAT を用い、ノード毎に 1 プロセスを配置した。つまり、ノード数の増加に対して、先頭データノードからコーディングノードへのデータ転送量は一定 (501.2MB) になる。結果を図 6 に示す。ノード数の増加に対してエンコード時間はほぼ一定であることが確認できた。詳細にはノード数の増加とともに時間も微増しているが、これに関してはパイプライン長の増加により、パイプラインが転送データで満たされるまでの時間が増加することによる影響と考えられる。

4.3 チェックポイント時間の比較

Incremental Checkpoint では、一般のチェックポイント時間に対し、増分を判別するためのコストがかかる。このため、エンコード時間が短縮されたとしても、チェックポイント時間が増大する可能性がある。これを評価するために、チェックポイントの総時間を CP, ICP において比較した。パラメータとしてデータノード 32 ノード、コーディングノード 2 ノードを用いた。チェックポイント総時間とは、プロセスにチェックポイントを指示してから、各プロセスがチェックポイントを作成、Parallel Erasure Daemon にエンコードを要求し、コーディングノードに誤り訂正符号を作成し終えるまでの時間である。結果を図 7,8 に示す。ckpt が BLCR によるチェックポイント出力時間、diff が増分計算・出力時間、enc がエンコード時間を示す。ICP における ckpt+diff が Incremental Checkpoint 時間であり、CP におけるチェックポイント時間に対応する。MAT において 83.8%, NPB LU において 28.4%の速度向上を確認できた。特に、NPB LU のような、Incremental Checkpoint に対して不利なメモリアクセスを行うようなプログラムにおいても効果を確認できた。この理由として、今回用いた、Incremental Checkpoint の性能が十分高かったことが挙げられる。これに対し性能の悪い Incremental Checkpoint 例として、増分の作成を低速な HDD 上で行った場合、360sec という非常に長いものとなった。このため、本エミュレーション手法で Incremental Checkpoint を実現する場合は、より高速な I/O デバイスを利用する必要がある。

5. 関 連 研 究

5.1 Diskless Checkpoint

ローカルディスクのない環境において、チェックポイントを高速に保存するための技術[9]。メモリ上にチェックポイントを行うため Erasure Coding を用い、少ない冗長データで最大限の耐故障性能を得ている。誤り訂正符号はアプリケーションプロセスの利用していないノードを利用し、故障発生時にはデコード作業によりそのまま計算ノードにリカバリーされ

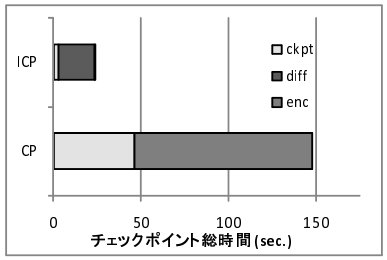

図 7 MAT:チェックポイント総時間の構成

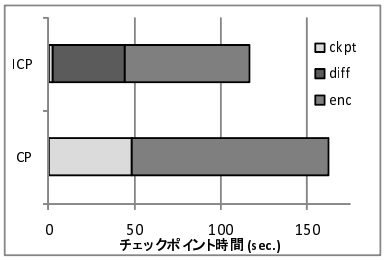

図 8 NPB LU:チェックポイント総時間の構成

る。我々の研究との差異は、ディスクを使用しないため、小さなプロセスイメージのみにしか対応できない、増分に注目したチェックポイントサイズの削減を行っていないことが挙げられる。また、Diskless Checkpoint では、エンコード演算のために Fan-in 方式を用いている。

5.2 Using Erasure Codes Efficiently for Storage in a Distributed System

分散環境における、ノード固有データの喪失を防ぐために Erasure Coding を利用した研究[10]。この研究はストレージノード全体のデータに対して Erasure Coding を行いデータ損失を防ぐ研究である。しかし、チェックポイントという目的のためには、耐故障性ストレージへの集中書き込みはデータ送信の輻輳が起きる可能性が大きい。また計算ノードとストレージノードの共有も提案されているが、データ書き込みが起こるたびにエンコード処理を行うため、アプリケーションプログラムの性能に影響を与えてしまう。またチェックポイント取得では同じファイルへの同時書き込みやファイルの部分変更処理等を考える必要がなく、より単純なアルゴリズムが採用できるため、我々の研究のようなチェックポイントに特化した対処を行った方が高速にイメージの書き込みが可能となる。

6. おわりに

本研究では、Erasure Coding ベースのチェックポイントに Incremental Checkpoint を組み合わせることにより、高速かつ多重故障に対応したチェックポイントを実現した。この実装に際し、Erasure Coding をパイプラインを利用して並列化し、ノード数増加によるチェックポイント時間の増大を抑えた。また、Incremental Checkpoint の効果が出やすいプログラム、および出にくいプログラムを対象として本チェックポイントを実行し、従来の Erasure Coding ベースチェックポイントと比較し、28-84%のチェックポイント時間の削減を実現した。今後の課題としては、大規模環境を利用し、詳細なスケーラビリティを調べることが挙げられる。また、実際の実用的な Incremental Checkpoint 実装を利用するとともにチェックポイントイメージの増分量と Incremental Checkpoint による負荷をモデル化する。加えて、今回は Parallel Erasure Daemon 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、他の転送方式である、broadcast や fan-in といった方法との詳細な比較を行う。以上を統合し、実行環境に合わせたアルゴリズムを適切に選択可能なチェックポインタの実装を目指す。

参 考 文 献

1) Top500 Supercomputer Sites: http://www.top500.org.
2) Plank, J.S.: Jerasure: A library in C/C++ facilitating erasure coding for storage applications, Technical report, University of Tennessee Technical Report (publication UT-CS-07-603) (2007).
3) Duell, J., H.P. and Roman., E.: The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart., Technical report, Berkeley Lab Technical Report (publication LBNL-54941) (2003).
4) Plank, J.S.: A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-Like Systems, *Software-Practice and Experience vol. 27, no. 9*, pp.995–1012 (Sept. 1997).
5) Chen, Z., Fagg, G.E., Gabriel, E., Langou, J., Angskun, T., Bosilca, G. and Dongarra, J.: Fault Tolerant High Performance Computing by a Coding Approach, *10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, pp.213–223 (2005).
6) Plank, J.S., Beck, M. and Kingsley, G.: Libckpt: Transparent Checkpointing under Unix, *Conference Proceedings, Usenix Winter 1995 Technical Conference, New Orleans, LA, January*, pp.213–223 (1995).
7) xdelta Official Site: http://xdelta.org/.
8) Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, D., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V. and Weeratunga, S.K.: The NAS Parallel Benchmarks, *The International Journal of Supercomputer Applications*, Vol.5, No.3, pp.63–73 (1991).
9) Plank, J.S., Li, K. and Puening, M.A.: Diskless checkpointing, *IEEE Transactions on Parallel and Distributed Systems* (1998).
10) Aguilera, M.K., Janakiraman, R. and Xu, L.: Using Erasure Codes Efficiently for Storage in a Distributed System, *Dependable Systems and Networks, 2005. DSN 2005. Proceedings*, pp.336–345 (2005).