容易なアドバイス記述法をもつ Fault Resilience プログラム環境構築にむけて

實 本 英 之[†1] 石 川 裕[†1]

エクサスケールの HPC 環境を実現するに当たり、故障に対処する機能が必須となっている。Fault Tolerance は、システム側で故障に対応するため、故障の局所化が緩く、故障復旧に余計なコストが必要になり、エクサスケール環境の高故障率に耐えられないと考えられている。このため、必要最低限のコストで故障復旧を可能とするために、アプリケーションプログラマが故障対応のコードを記述する Fault Resilience が提案された。しかし、故障検知や利用ライブラリの対処等を考慮した故障対応コードの記述は難度が高く、これを簡易化するフレームワークは必須である。本研究では、故障対応コードのうちアプリケーション特有の手法部分のみのコーディング (アドバイス) を平易に記述する手法について検討を行い、この検討を元に例外処理・条件分岐を利用したアドバイス記述フレームワークを提案した。またベンチマークアプリケーションの 1 つである HPL に提案フレームワークをもちいたチェックポイント／リスタート機構を試験実装し、動作の検証をおこなった。

Toward a Fault Resilient Environment with a Simple Coding for Application Programmers

Hideyuki Jitsumoto[†1] and Yutaka Ishikawa[†1]

Fault handling is a nessesary technique for exa-scale HPC environments. Fault Tolerance, which is the one of fault handling method used on recent HPC environments, localizes faults loosely due to handling faults on system-level. As a result, fault tolerance is not applicable for exa-scale HPC environments because it has needless fault handling cost. On the other hand, Fault Resillience is proposed for exa-scale HPC fault handling. In this method, faults are observed from applications and applications deal with fault handling. However, coding about fault handling is difficult for application programmer because they consider about the out of the application such as fault detection and middleware recovery. For simple coding for fault handling, we investigates how to code the 'advice', which is the part of fault handling related for application and propose the advice description framework based on this investigation. Moreover we apply this framework to checkpoint/restart on HPL for working validation.

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、アプリケーションを実行するに当たり、故障対応が必須要件になっている。故障対応をアプリケーションプログラマやアプリケーションユーザに実装させるのはコストが高く、システム側で自動的に故障に対応し、アプリケーション側に問題が波及するのを防ぐ仕組みとして Fault Tolerance が提案された。この手法の一例として、システムレベルチェックポイント／リスタートが多くの大規模 HPC 環境で利用されている[1),2)]。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。

しかしながらエクサスケール HPC 環境を実現するに当たりシステムによる一元的な故障対応アルゴリズムでは不要なコストが大きく、復旧に必要な総コストが平均故障間隔を上回り、プログラムの実行が破綻してしまう。再度例としてチェックポイント／リスタートをあげると、チェックポイントのコストもしくは、リスタートにかかる時間および、チェックポイントから故障時まで再実行する時間が平均故障間隔を上回ると、プログラムの実行が進まないという現象が起こる。これを防ぐために差分チェックポイントやチェックポイント先の適切な選択、チェックポイント周期の最適化によりシステムによる故障対応を維持したまま、チェックポイントコストを削減するという研究も行われた。

故障対応アルゴリズムのコストを極限まで削減するためには、アプリケーション毎に適したアルゴリズムを適用することが重要である。これを実現するためには、アプリケーションプログラマが故障対応をコーディングする必要がある。このような故障をプログラマから観測可能な状態においた環境のことを Fault Resillience な環境と呼ぶ[3)]。Fault Resillience 環境は Fault Tolerant な環境と比較し、以下のような特徴を持っている。

故障対応コストの削減 故障対応範囲を局所化できるため、復旧コストを押さえることが可能である。チェックポイントの取得においても、プロセスを復旧するために必要なデー

†1 東京大学
The University of Tokyo

© 2011 Information Processing Society of Japan

タのみを保存すれば良く、書き込み／読み込み時間を短縮することが可能であり、また並列プロセスのチェックポイントで必要不可欠な同期処理も、チェックポイントを行う位置を調整することにより最小限とすることが出来る。その他、アプリケーションの利用する外部モジュールの故障なども、該当モジュールを利用している位置で問題が起こることにより、原因を特定しやすく、復旧コードを記述することにより対応可能となる。例として、ファイルシステムの故障が起こった際は、読み込み終了位置を覚えておき、冗長系のファイルシステムから新たにファイルを開き直すといった手法が可能である。

Silent Error への対応　メモリ上のデータ不正等の Silent Error に起因する Byzantine Failure への対応が可能となる。システムレベルの故障対応を行う場合、アプリケーションの管理するデータが正常かチェックすることが出来ないため、故障を検知できない。しかし、アプリケーション実装者はデータの異常をチェックするコードを記述可能であり、この種の故障を検知可能である。

コーディング負荷の増大　アプリケーションプログラマが想定される故障に対し適切な故障対応コードを記述する必要がある。故障対応を行うためには、プログラム中の故障隠蔽、故障検出の手法やミドルウェアの構成をはじめとする、専門的な知識が必要になる。

Fault Resillience はエクサスケール HPC 環境を実現する為に重要な手法であるが、Fault Tolerance の持っていたアプリケーションプログラマへのコーディング負荷軽減の機能を犠牲にしている。このため、故障対応手法の全てではなく、アプリケーション特有の手法部分のみをアプリケーションプログラマにコーディング（アドバイス）させることにより、コーディング負荷の軽減を図ると共に、このアドバイスを容易に記述できるフレームワークが必要である。

本研究では、このアドバイスを容易に記述できるフレームワークを検討する。さらに、故障情報を複数のコンポーネント間で転送しあう CiFTS[4] を利用すると共に、故障情報に対して既存プログラミング手法である例外処理と条件分岐を利用したコーディングフレームワークを提案する。本フレームワークを利用することにより、アプリケーションプログラマがコーディングすべき内容をアプリケーションに起因する情報の復旧のみに限定できる。

以後の構成は、2 節でアドバイス記述容易化の検討を行う。次に 3 節でアドバイス記述フレームワークの提案を行い、4 節で今回の評価で用いた試験実装とその動作について検証を行い、5 節で関連研究について述べた後に、6 節でまとめを行う。

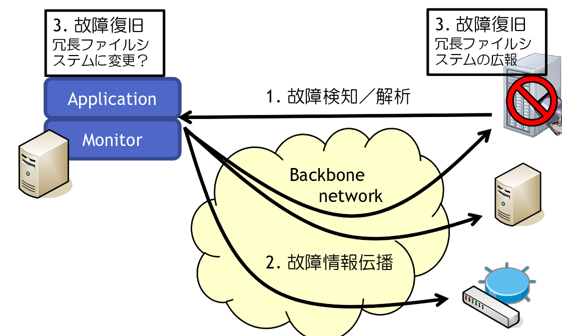

図 1　アドバイス記述の検討に仮定する環境

2. アドバイス記述容易化の検討

2.1 故障シナリオ

アドバイス記述手法を検討するに当たり、対象とする故障シナリオとして以下を考える。

外部コンポーネントに起因する故障　ファイルシステムの故障やネットワークダウン、またこれらの外部コンポーネントの一時的な負荷上昇により起こるエラーが伝播することに起因する故障。外部コンポーネントを利用するシステムコールの失敗、もしくは他の外部コンポーネントからの故障情報を受けることにより検出する。外部コンポーネントの冗長系を利用することにより故障復旧を行う。

Silent error に起因する故障　メモリ上のデータ不正等の Silent Error に起因する故障。中途結果が、アプリケーション内の検証ルーチンに失敗することにより検出する。復旧にはアプリケーションレベルのチェックポイント／リスタートを行う。この手法は、アプリケーションを復旧させる最低限の情報だけをチェックポイントするもので、システムレベルのチェックポイントと比べ低いコストで実現される。

2.2 アドバイス記述の必要要件

故障対応の手法は、大きく分けて 1) 故障の検知／解析、2) 解析情報の伝播、3) 故障復旧に分けられる。これに合わせ、本研究では環境を図 1 の様に仮定する。故障の検知／解析は環境を監視するモニタやファイルシステム、アプリケーションなどの環境を構成するコンポーネント群によって行われる。検知・解析された情報は故障情報を伝播するバックボーンネットワークによって再び各コンポーネントに伝えられ、各コンポーネントは故障情報に応

ⓒ 2011 Information Processing Society of Japan

じた復旧処理を行う。この環境を構築するためには以下の機能をアプリケーションプログラマに提供する必要がある。

故障の検知・隠蔽 ファイルシステムやネットワーク等外部モジュールにより引き起こされる障害を、故障復旧がなされるまで適切に隠蔽する機能。これを実現する為には、外部モジュールとのインタラクションをおこなう関数毎に、故障が起きた際の実行中断機能が必要である。

故障情報の送信 アプリケーションプログラマの検証ルーチンにより発見された障害を含め、アプリケーションにおいて検出された障害を他コンポーネントへ送出する機能。

故障情報の受信・復旧 非同期に発生する障害を適切にハンドリングする機能。また、ハンドリングのコスト、およびアドバイスの記述コストを下げるために、必要な場所で必要な最小限の復旧のみにフォーカスしてアドバイスを記述出来る必要がある。このためには故障情報を選択的に受信できる機能および、受信に合わせて割り込み的にハンドリングを行う機能。また、アプリケーションに起因しないミドルウェア等の状態を復旧する機能が必要となる。

特にアドバイス記述については、アプリケーション状態の復旧アルゴリズムをプログラマが記述する部分であり、容易な記述性が必要になる。割り込み的にアドバイスを実行する手法はシグナルハンドラやスレッドによるもの等、複数存在するが、以下の点に注意する必要がある。

各種関数の利用制限 シグナルハンドラやスレッドを使うことにより、アドバイスに利用可能な関数に制限を受けることがあり、実装難度が増してしまう。例として、シグナルハンドラを利用するのであれば非同期シグナル安全の関数しか利用できず、またスレッドを利用する場合も再入可能関数の利用が必要となる。

元コードのスコープの維持 アドバイスの変数スコープは、元コードのスコープと共有できることが望ましい。これが実現されない場合、アドバイスに必要な多くの変数を大域変数として定義することになり、コードの可読性やパッケージ化を著しく阻害する。

アドバイスへの入出位置 アドバイスから元コードへの復帰位置がわからない場合、プログラマは復旧時にアプリケーションをどのような状態にすれば良いのか規定することができない。また、進入位置がわからない場合、プログラマはアドバイス進入時にアプリケーションがどのような状態であるのかを取得することが出来ない。しかし、進入位置を特定し、故障したコンポーネントを利用するタイミングがアドバイス進入位置より早く実行された場合、アドバイス自体が実行不能になってしまう。

3. アドバイス記述フレームワーク

3.1 フレームワークの構成

2 節での検討に則り、既存プログラミング手法である try/catch-like な例外処理と条件分岐を利用したコーディングフレームワークを提案する。本フレームワークはディレクティブを指定することによりアドバイスを記述するもので、1) 故障隠蔽を実現する関数ラッパの生成、2) 故障情報の送信、3) 条件分岐を利用したアドバイス記法、4) 例外処理によるアドバイス記法により構成される。2 節で仮定した要件にある故障解析情報の伝播には CiFTS[4) を利用する。CiFTS は接続する複数のコンポーネント全てにイベントを送出するバックボーンネットワークを形成するサーバ群を利用するための API を規定している。フレームワークの各要素の詳細は以下のようになっている。

故障隠蔽を行う関数ラッパ 外部モジュールを利用する関数実行時に、利用外部モジュールに故障が起きている場合、関数が正常実行されない。この場合、プログラムの進行を停止し、故障復旧後、関数を再開する必要がある。対象とする関数名とエラーコードを指定することにより、コンパイル時に自動的に故障隠蔽を行う関数ラッパを生成するディレクティブを提供する。関数のエラーコードは関数の返値を利用するものや、引数に格納するもの等があるが、スタンダードなものに関して複数の指示法を用意する。ディレクティブには、故障発生時に送出するイベント名を付記出来る。

故障情報の送信 CiFTS を用いてイベントを送信するためのディレクティブ。イベント名と付記情報として規定長の文字列を送ることが出来る。

条件分岐を利用したアドバイス記法 ディレクティブ記述位置において指定されたイベントが受信されているかを確認し、受信時に指定されたアドバイスを実行する。この記法はアドバイスへの進入位置が明示的であるため、故障復旧自体を行うアドバイスに利用する場合は故障隠蔽が行われる関数との併用はデッドロックを生み出す可能性がある。このため本ディレクティブは、主としてチェックポイントなどのアプリケーションの状態を取得するために利用する。

例外処理を利用したアドバイス記法 JAVA や C++等に用いられているプログラマによく知れ渡った try/catch-like な手法により、指定されたイベントが届いた際に指定されたアドバイスを実行する。元コードの変数スコープを維持した処理が可能かつ故障情報を受信する範囲も明示的に指示可能であり、元コードのプログラム進行に割り込む形でアドバイスを実行可能である。ディレクティブにはイベント名と復帰位置を示すフラグ名

© 2011 Information Processing Society of Japan

を指定できる。復帰位置の定義も同様にディレクティブで行う。

3.2 故障シナリオに対する復旧例

前節の提案を利用し、2 節で述べた故障シナリオを復旧する例を述べる。

3.2.1 外部コンポーネントに起因する故障

図 2 はファイルシステムに異常が起こった際の復旧例である。このプログラムはファイルから char 型の文字を 1 文字ずつ読みながら表示するもので、∗ 印のついている行がアドバイスによる追加部分である。アドバイスは、実行時間のほとんどを占める fread のループに対して故障復旧を行うよう設計されている。fread において何らかの要因で読み取り失敗が起きた際、アプリケーションからエラーイベント FS_ERR を発行、それを受信した環境内のモニタから、復旧イベント FS_RCV が発行されるというシナリオを想定している。各種アドバイスの内容は以下である。

(1) fread の故障隠蔽を行うためのラッパの自動生成
fread は最終引数のファイルハンドラを ferror 関数にかけることによりエラーを検知可能であり、評価式 $ferror(a) > 0$ である場合はエラーが発生したとしてイベント FS_ERR を発行する。

(2) 本フレームワークに利用されるライブラリ等の初期化処理

(3) FS_RCV を受信可能に変更

(4) 例外処理型のアドバイスの記述
try{, try} で囲まれた区域において、FS_RCV が受信された場合、catch ディレクティブに続くアドバイスを実行してラベル FRETRY の位置に戻る。アドバイスは現在までのファイルの読み込み位置を取得し、冗長ファイルシステムの同ファイルをオープン、読み込み位置をセットし直すというものである。冗長ファイルの位置は復旧イベントの付記情報として書かれるべきものであるが、ここでは単純化のため定数としている。

(5) 復帰位置を指定するためのラベル名の定義

(6) FS_RCV を受信拒否に変更

(7) 本フレームワークに利用されるライブラリ等の終了処理

3.2.2 Silent Error に起因する故障

図 3 はある演算 (calculate 関数) を繰り返し実行するプログラムにおいて、計算結果が何らかの要因で不正であった場合の復旧例で、∗ 印のついている行がアドバイスによる追加部分である。アプリケーションプログラマは中途計算結果が正しいかを検証する ast 関数を用

```
#pragma XPT mask fread(,,,a) ferror(a)>0 FS_ERR *...(1)
int main(){
  FILE *fp;
  char c;
#pragma XPT init                         *...(2)
  fp = fopen("/fs1/user1/xxx", "r");
#pragma XPT subscribe FS_RCV             *...(3)
#pragma XPT try{                         *...(4)
#pragma XPT label FRETRY                 *...(5)
  while(<file do not reach EOF>){
    fread(&c, sizeof(char), 1, fp);
    printf("%c", c);
  }
#pragma XPT catch FS_RCV return FRETRY *...(4)
  long int offset = ftell(*fp);          *
  fclose(*fp);                           *
  *fp = fopen("/fs2/user1/xxx", "r");    *
  fseek(*fp, offset, SEEK_SET);          *
#pragma XPT try}                         *...(4)
#pragma XPT unsubscribe FS_RCV           *...(6)
#pragma XPT fini                         *...(7)
  return EXIT_SUCCESS;
}
```

図 2 ファイルシステム故障に対応可能なアドバイス

© 2011 Information Processing Society of Japan

```
int main(){
#pragma XPT init                         *
#pragma XPT subscribe CR_RST,CR_CKPT     *
#pragma XPT try{                         *
  for(iter i){
    calculate(i, data);
#pragma XPT assert ast(data)>0 AP_SERR   *...(1)
#pragma XPT chk{ CR_CKPT                 *...(2)
    write i to file;                     *
    write data to file;                  *
#pragma XPT chk}                         *...(2)
#pragma XPT label RST                    *
  }
#pragma XPT catch CR_RST return RST      *
  read i from file;                      *
  read data from file;                   *
#pragma XPT try}                         *
#pragma XPT unsubscribe CR_RST,CR_CKPT   *
#pragma XPT fini                         *
}
```

図 3 Silent Error に対応可能なアドバイス

意しており、これで不正が判明した場合、AP_SERR イベントを発行、これを受けた外部コンポーネントのモニタが復旧イベント CR_RST を発行するというシナリオを想定している。また、本プログラムは外部モニタにより定期的にチェックポイントイベント CR_CKPT が発行されており、復旧時にはこのイベントによりとられたアプリケーションチェックポイントを利用する。各種アドバイスのうち前例で説明しなかったものの内容は以下である。

(1) 条件に合致する場合イベントを発行
中途結果 data を ast(data) により検証し、これが不正であった場合エラーとして AP_SERR イベントを発行する。

(2) 条件分岐型のアドバイス記述
外部から CR_CKPT イベントを取得した際は、iterator と中途計算結果 data を保存する。calcurate 関数は i, data により計算されるため、この 2 つにより最低限のチェックポイントを作成することが可能である。

4. 試験実装

4.1 HPL のチェックポイント／リスタート

提案したフレームワークの実現性を実証するために試験実装を行った。この試験実装は実際のベンチマークアプリケーションである HPL[5] の LU 分解を行うループ (HPL ソースコード内 HPL_pdgesv0.c に存在する) に対してチェックポイント／リスタート機能を提供するものである。HPL において LU 分解の中途結果は、問題行列に上書きされるように格納される。このため、問題行列のデータ領域全てをチェックポイントとして保存した。提案における各要素は以下のように実装されており、今回の試験実装で利用しない項目に関しては実装指針を示した。また CiFTS 規格に準拠したバックボーンネットワークとしては、ANL の開発した CiFTS の参照実装である FTB を利用した。

故障隠蔽を行う関数ラッパ 今回は故障復旧を行わないため、利用していない。この機能は、故障隠蔽を行う関数をラップし、故障が発生した場合は sleep を行うことにより実現する。sleep からの復帰は、以降で述べる例外処理により行われる。

故障情報の送信 CiFTS のイベント送信 API を用いて実現される。

条件分岐を利用したアドバイス記法 CiFTS のイベントポーリング API を利用し、イベントを受信していた場合に if 節による条件分岐を行い指定されたアドバイスを実行する。

例外処理を利用したアドバイス記法 sigsetjmp/siglongjmp、および CiFTS のイベント購読 API を利用した。イベント購読 API は特定のイベントを受信した際に登録しておいた関数をスレッド実行するものである。本実装では、この関数として、メインスレッドに pthread_kill を用いたシグナル送信を行うものを登録している。また、try 開始時には sigsetjmp を発行し、pthread_kill により発行されるシグナルのハンドラとして siglongjmp を発行する。これにより、変数スコープを元コードと共有することができる。

本実装の利用シナリオは、外部にジョブをマネージメントするスケジューラが存在することを仮定する。外部スケジューラはアプリケーションに対して定期的にチェックポイントイベント CR_CKPT を発行している。また、スケジューリングの再配置等を理由に、アプリケーションを強制終了させることがある。強制終了されたアプリケーションはスケジューラのタスクが終了し、資源に余裕ができ次第、チェックポイントから再開を行う。その際スケジューラは、プログラムを最初から再起動し、リスタート可能な状態になったことを示す CR_RDY がアプリケーションから発行されたことを確認してから、CR_RST イベントをア

© 2011 Information Processing Society of Japan

```
pthread_t pid;
sigjmp_buf jbuf;
int main(){
  pid = pthread_self();                               ... init
  FTB Initializing Functions;                         ... init
  FTB_Subscribe(CR_CKPT, handler(&event_status)); ... subscribe
  FTB_Subscribe(CR_RST, handler(&event_status));  ... subscribe
  if(sigsetjmp(jbuf)==0){                             ... try{
  FTB_Publish(CR_RDY);                                ... throw
  for(iter i){                                            (event send)
    calcurate(i, data);
    sigmask(SIGUSR1);                                 ... chk{
    if(FTB_Poll_Event(CR_CKPT)){                      ... chk{
       checkpointing;
    }                                                 ... chk}
    sigunmask(SIGUSR1);                               ... chk}
CR_RST:                                               ... label
  }
  }else if(event_status.name=="CR_RST"){              ... catch
       restarting;
       goto CR_RST:
  }                                                   ... try}
}

void handler(*event_status){
  pthread_kill(pid,SIGUSR1);
}

void sigusr1handler(){
  siglongjmp(jbuf);
}
```

図 4 チェックポイント／リスタート実装の概略

プリケーションに対して発行する。

図 4 は図 3 からチェックポイント／リスタートに関わる部分を実装したものである。実際の HPL では calculate 関数部分に、LU 分解を 1 ステップ分行う関数が入っている。また、リスタートが可能となる時点をモニタに報告するためにイベントの送信文が追加されている。なお、CiFTS API を含む一部の関数の引数は簡略化している。コードの右部に示したのは対応するディレクティブを示している。

表 1 提案フレームワークとシステムレベルチェックポイントのチェックポイントサイズ

OpenMPI+提案手法	OpenMPI+BLCR
33.9MB	44.5MB

4.2 制限事項

本実装では CR_RST イベントを受け取ると、シグナルハンドラ内から sigsetjmp の位置へと強制的にジャンプする。これは非正規脱出にあたるため、シグナルによる中断から元の位置へ戻る必要があるシステムコール (malloc 等) やアプリケーションプログラマの実装したものではないライブラリ API、実装に mutex を利用しているような関数はあらかじめシグナルによる中断を受けない様マスクする必要がある。また、シグナル受信により、無条件でエラーを発行してしまう関数 (OpenMPI の MPI_Bcast 等) も同様なマスクが必要である。

4.3 動作環境と検証

提案フレームワークにより改造した HPL によるアプリケーションレベルチェックポイントと BLCR[6] によるシステムレベルチェックポイントのチェックポイントサイズを比較したものが表 1 である。双方とも実行には OpenMPI を利用している。HPL の問題サイズ (Ns) は 2048、MPI による 4 並列実行を行い、全てのプロセスから作成されたチェックポイントサイズの総和を測定した。アプリケーションにおいて最低限必要なデータをプログラマがアドバイスすることにより、チェックポイントのサイズが抑えられていることがわかる。しかし、本実装ではチェックポイントを問題行列が格納されている領域全てとしてしまったため、メモリ領域の大部分を保存することとなり、大きな削減効果には至らなかった。LU 分解は、ステップが進行するたびにデータの更新領域が減っていく傾向がある。アプリケーション側からのアドバイスにより、確定したデータ領域を取得することは容易であり、データ変更部分に着目したチェックポイントを行えば更にコンパクトなチェックポイントを取得可能である。

5. 関連研究

5.1 CiFTS/FTB

本研究でも利用した CiFTS[4] フレームワークとその実装である FTB は、複数のコンポーネント間で故障情報の共有を可能とするものである。CiFTS API はイベントの送信と、ポーリング型、購読型の二種類のイベント受信をサポートしている。本研究の例外処理を利用し

© 2011 Information Processing Society of Japan

たアドバイス記法に当たる、購読型のイベント受信 API は受信時のコールバックスレッドを設定するもので、元コードとのスコープの共有が出来ないこと、および元コードとスレッドが同時に実行されるために、アプリケーションデータの取り扱いに対しての排他制御が必要になるなどの問題がある。また、アドバイスの実行が侵入位置、復帰位置ともに不明となるため、アプリケーションをどの状態に復旧すればよいかがわかりづらくコーディング難度が高い。

5.2 OpenMPI/MPICH-V

OpenMPI[2]/MPICH-V[1] はそれぞれ、MPI 実装の 1 つであり、システムレベルのチェックポイント／リスタートを備えている。他にもチェックポイント／リスタートを持つ MPI 実装は存在するが、これらの MPI 実装は、故障対応の並列アルゴリズムをコンポーネント構造により柔軟に変更できるという利点がある。これにより、実行環境に合わせたチェックポイント取得を行うことが出来、コストの削減につながっている。しかしながら、システムレベルの故障対応ではアプリケーションの性質を利用した故障対応機能を利用することが難しい。なお、OpenMPI においては、後述するアプリケーションレベル故障対応をもつ FT-MPI と同等の機能を実装する予定であることが公表されている。

5.3 FT-MPI

FT-MPI[7] は、アプリケーションレベルの故障対応機能を持つ MPI 実装の 1 つである。プログラマは拡張された MPI のエラーコードを利用したり、MPI エラーに対し、例外処理を定義することにより、故障対応をアプリケーションコード内に記述することができる。しかしながら、本実装は MPI 環境をどのように復旧するかといった、ミドルウェア部分の故障復旧も記述する必要があり、アプリケーションプログラマにとっては実装難度が高い。また、故障検知機能は MPI に特化されており、外部コンポーネントによる故障を受信する機能がないため、故障情報の伝播部分もアプリケーションプログラマが実装する必要がある。

6. おわりに

本研究では、Fault Resillience 環境において故障対応を行うアドバイスコードを平易に記述するフレームワークについて検討し、例外処理と条件分岐を模したアドバイス記述手法を提案した。提案フレームワークは既存コードにディレクティブを追加することによりアドバイスを記述可能とするもので、プログラミングに必須である if 節による条件分岐、C++ や JAVA などに利用されている try/catch-like な例外処理といったプログラマにとってなじみ深いフレームワークを用いている。また、アドバイスと既存コードの変数スコープの共有や、アドバイスへの入出位置の明確化により、アプリケーションプログラマが平易にアドバイスを記述できる仕組みを提供している。この提案を利用し、実際のベンチマークアプリケーションである HPL にチェックポイント／リスタート機能を試験実装することによりフレームワークの実現性を検証し結果、アプリケーションからのアドバイスによる故障対応コスト削減が実現できた。今後の課題としては、現在の実装手法で問題になっているシグナルハンドラからの非正規脱出を解消する。また、実際にディレクティブを解釈するプリプロセッサを実装し広く利用されるように公開していくことを考えている。

謝辞 本研究は，科学技術振興機構戦略的国際科学技術協力推進事業（共同研究型）「日本ーフランス共同研究・ポストペタスケールコンピューティングのためのフレームワークとプログラミング」の補助を受けている。

参 考 文 献

1) Aurelien Bouteiller, Thomas Herault, Geraud Krawezik, Pierre Lemarinier and Franck Cappello: MPICH-V: a Multiprotocol Fault Tolerant MPI, *International Journal of High Performance Computing and Applications.* (2005).
2) Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham and Timothy S. Woodall : Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation, *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, pp.97–104 (2004).
3) Franck Cappello, Al Geist, Bill Gropp, Laxmikant V. Kale, Bill Kramer and Marc: Toward Exascale Resilience., *IJHPCA*, Vol.23, No.4, pp.374–388 (2009).
4) R. Gupta, P. Beckman, H. Park, E. Lusk and P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine and J. Dongarra: CIFTS: A Coordinated infrastructure for Fault-Tolerant Systems, *International Conference on Parallel Processing (ICPP)* (2009).
5) A. Petitet, R. C. Whaley, J. Dongarra and A. Cleary: HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/.
6) Duell J. , Hargrove 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).
7) G. Fagg and J. Dongarra: FT-MPI: FaultTolerant MPI,Supporting Dynamic Applications in a Dynamic World, *Euro PVM/MPI User's Group Meeting 2000 ,Springer-Verilag, Berlin, Germany*, pp.346–353 (2000).

© 2011 Information Processing Society of Japan