Vol.2013-HPC-139 No.10
2013/5/29

適切なチェックポイント周期を与えるアプリケーションレベルチェックポイントライブラリ

實本 英之[1,a)]　鴨志田 良和[1,b)]

概要：多くの科学技術計算において、定期的なチェックポイントファイルの出力は一般的になっている。しかし、この周期に関してはアプリケーションプログラマが経験即を元に決めていることが多く、適切なものにはなっていない。適切なチェックポイント周期は、故障率、チェックポイント時間、I/O 帯域によって決定されるが、これらの情報は実行環境に依存するもので、実行時の情報取得が必要となる。このため、外部から与えられる最適周期に対応可能かつ、同期を極力行わないチェックポイントライブラリを提案、実装した。アプリケーションプログラマは、アプリケーションのループ構造に対し、チェックポイントデータの登録やチェックポイントを行う位置の指定を行う API を呼び出すことにより、適切な頻度でチェックポイントを行うことが可能になる。

1. 背景と目的

大規模 HPC 環境では、要素数の増加や、高密度低電力実装の影響により、システムの故障率は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、耐故障機能が必須要件になっている。耐故障機能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システムレベルの耐故障機能が提案された。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 環境で利用されはじめている。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。

しかしながらエクストリームスケールの HPC 環境を実現するに当たり、従来のシステムによる一元的な耐故障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耐故障機能を維持したまま、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てきた。

しかしながら、耐故障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。チェックポイント／リスタートにおいては以下の様な最適化が考えられる。

チェックポイントデータ量の削減　アプリケーションプログラマはプロセスを復旧するために最低限度必要なデータを判別することが可能である。このため、チェックポイントデータ量の削減、これによる書き込み／読み込み時間を短縮することが可能である。

一貫性保証アルゴリズムの容易化　並列プロセスのチェックポイントでは、それぞれのプロセスが作成するチェックポイントに一貫性が保証されている必要があるが、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。

しかし先にも述べた通り、このようなアプリケーション毎の最適化は、アプリケーションプログラマに耐故障に関する知識を要求しコストが大きい。このため、耐故障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行う必要がある。特に、後者を実現するためのチェックポイントを一貫性を保持したまま行う事が可能な位置の指定については明確な指針を設ける必要がある。また指定した位置を

[1] 東京大学
The University of Tokyo
[a)] jitumoto@cc.u-tokyo.ac.jp
[b)] kamo@cc.u-tokyo.ac.jp

選抜し、最適な量のチェックポイントを行う事について、チェックポイント頻度は故障率やデータ転送時間等の環境によって決まるパラメータであり、システム側からのサポートを受けながら決定する必要がある。

これを解決する手法の一つに WBC-ALC[1] が存在する。この研究では、ユーザはチェックポイントの候補を記述し、並列プロセス間においてメッセージがやりとりされたときのみ、関連するプロセスをチェックポイントするという手法により一貫性を担保している。しかしながら、前述の通り、本来最適なチェックポイントの頻度は、プロセスの実行されるハードウェアの故障率、チェックポイントデータの転送時間から算出されるものであり、アプリケーションの性質によって決まるものではなく、アプリケーションの性質からもたらされる一貫性担保アルゴリズムと、外界の影響からもたらされるチェックポイント頻度算出アルゴリズムを適切に結びつける必要がある。

本研究では、反復計算を主とする科学技術計算を対象として、チェックポイント／リスタートを容易に実装するフレームワークを提案する。また本フレームワークは一貫性保証アルゴリズムと、チェックポイント頻度算出アルゴリズムの両方を、適用可能であることを目標とする。

2. 設計

チェックポイントアルゴリズムを検討するに当たり、対象とするアプリケーションに以下の要件を仮定する。

反復計算 気象シミュレーション等に利用される科学技術計算では代表的なアプリケーションで、単位時間に対して複数のソルバーによる計算を行い、計算結果を並列ジョブ構成プロセス間で遣り取りしデータを更新、これを元に再び次の時間ステップの計算を行うという構造のものとする。

負荷分散の均等性 十分に最適化されたアプリケーションを対象とし、負荷分散が適切に為されており、すべてのプロセスの実行時間がおおよそ同じであるものとする。

小さなデータ量・通信量・計算量の変化率 最適なチェックポイント間隔を算出するためのパラメータの変動が少ないもの。これが大きくても動作に支障はないが、チェックポイント間隔更新が増えるためコストが大きくなる。

このようなアプリケーションは単位時間を制御変数としたループで構成されている。この制御変数の数値のことを以降反復番号と呼ぶことにする。

2.1 動作シナリオの概要

本フレームワークは、チェックポイントされるアプリケーションの他に、チェックポイント間隔最適化サーバを設ける。また、アプリケーションプログラマがチェックポイント候補位置、チェックポイントデータをアノテーションとして指定することで、計算時間・チェックポイント時間を自動的に測定し、最適化サーバに送信する。最適化サーバがチェックポイントすべき反復番号を並列アプリケーションに知らせることで、アプリケーション構成プロセスが非同期にチェックポイントを取得する Asynchronous Coordinated Checkpointing を行う。チェックポイントの取得はアプリケーションプログラマが挿入するチェックポイント候補アノテーションの位置で行われる。詳細は節 2.3 で述べる

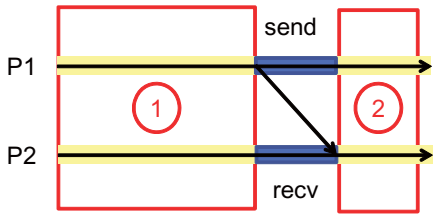

図 1 反復計算における 1 ループ実行のモデル

2.2 一貫性保証

2 プロセスで構成される並列アプリケーションを考え、これが通信 M_n を持っているとする。構成プロセス内において、M_n と M_{n+1} に挟まれた部分をエポック E_n としたとき、双方のプロセスが同じ E_n にあるときに作成されたチェックポイントは一貫性が保証される。図 1 は 1 回のループにおけるプロセスの状態を示したもので、仮定に基づきソルバを実行するタイミングである 1, 2 の部分、計算結果をプロセス間で遣り取りする通信部分が存在する。このとき、一貫性の保証されるチェックポイントの位置は、反復番号 n における 2 の部分および反復番号 $n+1$ における 1 の部分のいずれかですべての構成プロセスの状態が存在する場合である。MPI に代表される SPMD 型のアプリケーションでは通信関数がほぼ同じ場所にまとめられておりわかりやすく、アプリケーションプログラマがこれらの候補位置を指示するアノテーションを記述することは容易である。

今回は最適化計算と実装の単純化のために、チェックポイント対象のループ毎に最大 1 回のチェックポイントを行うこととし、かつチェックポイントアノテーションはすべてのプロセスで同じ位置に記述するとする。つまり、すべてのチェックポイントは同じ反復番号をもち、図 1 の 1 もしくは 2 のどちらかにすべての構成プロセスの状態が存在する場合にチェックポイントされているとする。

なお、本手法では、すべての並列ジョブ構成プロセスの反復番号が同じ状態で生み出されたチェックポイントのみが一貫性を持つため、この条件を満たすチェックポイントを出力するまでは前世代のチェックポイントセットを保持しておく必要がある。

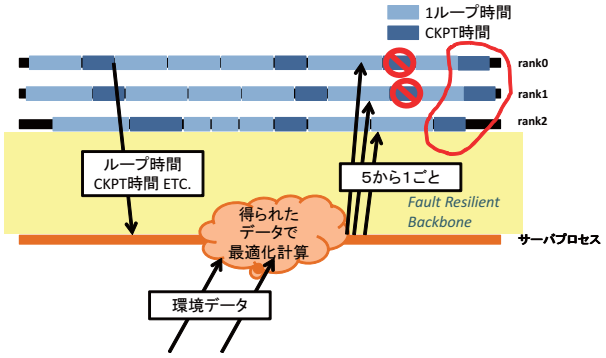

図 2　提案システムの動作シナリオ

2.3　最適なチェックポイント間隔の適用

最適なチェックポイント間隔の調整は、チェックポイントをすべき反復番号をすべての並列ジョブ構成プロセスが共有している必要がある。しかしながら、この反復番号を一致させるためにすべてのプロセスを同期・停止させるのはコストが大きいため、図 2 の様なシステムを提案する。本システムは、チェックポイント間隔最適化サーバ (図下部の赤線) と並列アプリケーションプログラム (図中 P0-2) および、Fault Resilience Backbone により構成される。Fault Resilience Backbone はアプリケーション、ミドルウェア、ハードウェアを横断的に接続し、情報を送りあうバックボーンである。チェックポイントの手順は具体的には以下の様になる。

(1) 並列ジョブ構成プロセスの代表がループ内の計算、チェックポイントの時間を測定する。
(2) 代表プロセスは Fault Resilience Backbone に対し、測定した時間を送信する。
(3) チェックポイント最適化サーバは、代表プロセスからのチェックポイント時間とハードウェアの故障率から最適間隔を計算し、各プロセスに配布する。この際、基準となる反復回数を添える。具体的には反復 n 回目から m 回目毎という様な内容となる。
(4) 全ての並列プロセスは、最適化サーバから送られた情報に基づき、チェックポイント候補地点でチェックポイントを取得する。

この際、最適間隔の受信タイミングは各プロセス毎に異なってしまう。このため、図のように一時的に一貫性を満たさないチェックポイント群がとられてしまい、複数の世代のチェックポイントを保持しておく必要がある。しかしながら、一度、反復回数とチェックポイント位置の一致するチェックポイント群が取得できれば (図右部赤丸) それ以前のチェックポイントは削除することができる。

2.4　構成プロセスのチェックポイント／リスタート

並列アプリケーションの構成プロセスは、チェックポイント候補位置まで実行が進むと、通常起動かリスタート起動かを確認する。通常起動の場合は初回のチェックポイントを行い、経過時間を最適化サーバに送信する。一方、リスタート起動の場合は、最新のチェックポイントを適切な各変数に読み込んだ後にチェックポイントを行い、経過時間を最適化サーバに送信する。

以降、チェックポイント候補位置に置いて、計算時間・チェックポイント時間の測定を行うが、前ループでの測定との差、もしくは比率での変動をトリガとして、最適化サーバに経過時間を送信し、チェックポイント間隔を再計算する。これは、通信量の増加とチェックポイント間隔の頻繁な更新を防ぐためである。

```
int main(){
 int i, data[30];
#pragma MICP init                    *
//フレームワークの初期化
#pragma MICP enable i, data(0-29) *
//以下のループで i, data[30] を
//チェックポイントする
  for(iter i){
    calculate(i, data);
#pragma MICP ckpt                    *
//チェックポイント候補位置
  }
#pragma MICP disable                 *
//チェックポイントループの終了
#pragma MICP fini                    *
//フレームワークの終了
}
```

図 3　アノテーションの利用方法

表 1　アノテーションの種類

init/fini	初期化、終了処理
enable/disable	チェックポイントをする変数群とループの指定
ckpt	チェックポイントを行う位置の指定

2.5　利用方法

本チェックポインタはアプリケーションプログラムにアノテーションを記述するためのディレクティブを挿入することで、利用可能になる。アノテーションの種類は表 1 に示すとおりであり、実際のソースコードは図 3 のように変更する。この図における*で示した行が元コードからの変更箇所となる。

3.　実装

3.1　故障・復旧モデル

故障・復旧モデルは差し替えが可能なようになっているため、必要に応じて対応可能な故障種を強化していくことができるが、本研究では、故障・復旧モデルを以下のよう

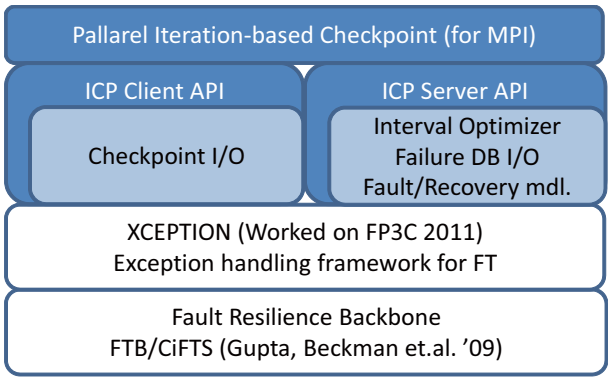

図 4　システムモジュール

に定めた。

故障モデル　ジョブのいずれかに故障が起きたときに、ジョブ全体が確実に停止する

復旧モデル　すべてのプロセスを終了し、あらかじめ確保しておいた冗長ノードを利用して、至近のチェックポイントからジョブ全体を再実行する。

3.2　システムモジュール

実装は図 4 の様なモジュールで構成されている。白色のレイヤは耐故障システムを容易に作成するフレームワークとして以前提案したシステム [2] であり、Fault Resilience Backbone は FTB/CiFTS[3] を利用している。そのほかの本研究で実装したレイヤについては以下のようになっている。

Iteration-based Checkpoint(ICP)　チェックポイント対象プロセスでアノテーションを利用して挿入する API を実装したレイヤ。大きく分けてチェックポイントを利用するためのクライアント API と、最適化サーバを構築するためのサーバ API がある。

Parallel Iteration-based Checkpoint for MPI(MICP)　並列化ライブラリとして MPI を利用することが前提となった並列ジョブチェックポイントシステムレイヤ。ここでは、最適化サーバや下位レイヤで利用する管理サーバ群の起動など、複数プロセスの連携手法が記述されている。

また、ICP において、以下が差し替え可能であるようにモジュール化されており、実行環境に合わせた柔軟性を備える。本研究においての実装も併記する。

最適化ルーチン　N. Vaidya の 3 状態遷移マルコフモデルを用いた最適化ルーチンを利用 (節 3.4 参照)

チェックポイントの出力手法　libc の高水準ファイル操作 (fopen/fwrite/fread) での出力 (MPI-IO 等で差し替えることを想定)。チェックポイントは共有ディスクにより各ノードで共有されていることを仮定している。

故障・復旧モデル　Fail-stop/re-execute モデル (節 3.1 参照)

故障率データベース　ホスト名・故障率の組をテキストファイルに列挙したものを対象 (SQL 利用等で差し替えることを想定)

3.3　実行環境構築

スーパーコンピュータにおいて、最適化サーバ、そのほかの管理サーバと MPI ジョブを連成して実行するようスケジューラに指定するのは、アプリケーションのユーザにとっては難しい。このため、本実装では前節 Parallel Iteration-based Checkpoint レイヤにて MPI ジョブが起動時に自身で管理サーバ群を起動するように実装されている。この実行環境は図 5 のように構成される。

(1) 各ノードにアサインされる MPI ジョブプロセスのうち 1 プロセスを管理プロセスとする。なお、並列数はアプリケーションで利用するプロセス数＋管理ノード数となる (16 プロセスを 4CPU ノードに配布する場合、MPI アプリケーションの並列数は 22(3 クライアントプロセス+1 管理プロセスが 5 ノード、1 クライアントプロセス+1 管理プロセスが 1 ノード))。

(2) 管理プロセスのうち rank0 が存在するノードにあるものがリーダー管理プロセスとなり、FTB のノード毎サーバ (ftb_agent) の接続先を配布するディレクトリサービスサーバ (ftb_database_server) を起動する。この動作は管理プロセス内でブロッキングする。

(3) 管理プロセスは ftb_agent を起動する。この動作はすべてのプロセスにおいてブロッキングする。

(4) リーダー管理プロセス、および計算プロセスはそれぞれのノードの ftb_agent に接続し送受信イベントの登録を行う。この動作はすべてのプロセスにおいてブロッキングする。

(5) リーダー管理プロセスは以後最適化サーバとしてメッセージをハンドリング、計算プロセスは計算を開始する。

本研究では管理サーバ群による OS ジッタの影響を抑える目的もあり、1 プロセッサコアを管理サーバ群に利用する手法となっている。しかしながら、これによる並列プロセス数指定の煩雑さや、アプリケーションへの影響も考えられ、これは今後改善する予定である。

管理プロセスの作成については東京大学情報基盤センターで作成された spawn tool という MPI_Spawn() で新たなプロセスを作成した際のプロセス配置を制御するライブラリを用いている。このライブラリは動的プロセス生成をプロセスプールのように静的に生成しておき、MPI_Spawn() 発行のタイミングで利用する。加えて MPI アプリケーションから見た場合の MPI_COMM_WORLD を上書きすることが可能になっているため、アプリケーションから見たランクや通信範囲は設計時と変わらず、起動時の並列プロセス数指定を変えるだけで追加のプロセスをノード毎

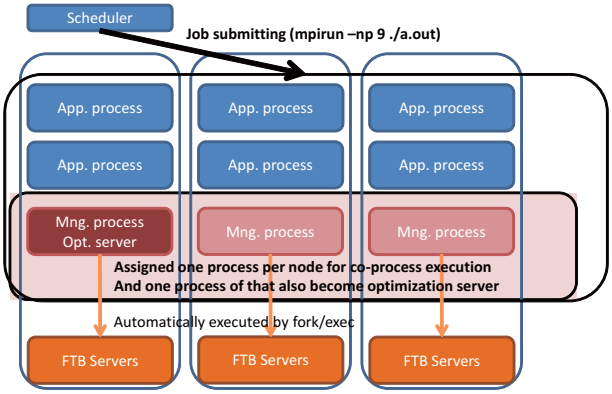

図 5　実行環境

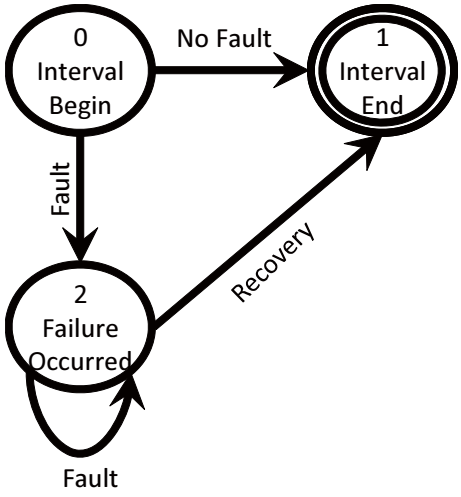

図 6　故障復旧における 3 状態遷移マルコフモデル

に作成することができる。なお、本研究では、MPI アプリケーションが行う他の MPI_Spawn と干渉しない様に、MPI_Spawn() の上書き定義は行わず、専用 API で動くよう改変している。spawn tool ライブラリの動作は以下のように行われる。

(1) MPI_Init() を上書きし、その内部で MPI_Get_processor_name() を用いてホスト名を集計する
(2) 各ノード毎に 1 プロセスをプロセスプールとして登録し、プロセスプール内プロセス、それ以外のアプリケーション用プロセスそれぞれで新たなコミュニケータを作成する
(3) 作成したコミュニケータをそれぞれの MPI_COMM_WORLD に上書きする。

3.4　チェックポイント間隔最適化アルゴリズム

最適化サーバにおいては、チェックポイント／リカバリによるオーバーヘッドを最小化するようにチェックポイント頻度を最適化することを目的とし、本研究では N.Vaidya のモデルに並列アプリケーション全体の故障率に拡張したモデルを利用する。具体的には、並列プロセスを実行するハードウェアを大きな一つの要素と考え、要素を構成するハードウェアのいずれかが故障する確率を、要素が故障する確率としてモデルを適用するものである。故障率の集計は初回のチェックポイントを作成するとき（リスタート時も含む）、および、計算時間・チェックポイント時間が大きく変動する際に行われ、プロセスが実行されるハードウェアの故障率の総和となる。

チェックポイントが終了してから、新たなチェックポイントを作り始めるまでの時間を T とする。このとき、チェックポイントのインターバルは $T+C$ で示される。また故障が発生すると、リスタートのためにチェックポイントの転送が起こり、そのコストは C で示せる。このとき、同期処理などのオーバーヘッドも存在するが、チェックポイントサイズが十分に大きいときは無視できる。N.Vaidya のモデルでは、チェックポイントインターバルを 3 状態のマルコフモデルで示している (図 6)。P_{ij} を i から j へ状態移動する確率、K_{ij} を i から j への状態移動に必要な時間とすると、それぞれの状態変化は以下のように示すことができる。なお、本モデルでは、計算時間、チェックポイント作成、リカバリがオーバーラップしないことを仮定している。

$$P_{01} = e^{-\lambda(T+C)} \tag{1}$$
$$K_{01} = T + C \tag{2}$$
$$P_{02} = 1 - e^{-\lambda(T+C)} \tag{3}$$
$$K_{02} = \int_0^{T+C} \frac{\lambda t e^{-\lambda(t)}}{1 - e^{-\lambda(T+C)}} dt \tag{4}$$
$$P_{21} = e^{-\lambda(T+2C)} \tag{5}$$
$$K_{21} = T + 2C \tag{6}$$
$$P_{22} = 1 - e^{-\lambda(T+2C)} \tag{7}$$
$$K_{22} = \int_0^{T+2C} \frac{\lambda t e^{-\lambda(t)}}{1 - e^{-\lambda(T+2C)}} dt \tag{8}$$
$$\tag{9}$$

これを用いて、チェックポイント間隔の期待値 Γ を示すと、

$$\begin{aligned}\Gamma &= P_{01}K_{01} + P_{02}(K_{02} + \frac{P_{22}}{1-P_{22}}K_{22} + K_{21}) \\ &= \lambda^{-1}e^{\lambda C}(e^{\lambda(T+C)} - 1)\end{aligned} \tag{10}$$

となる。これより、チェックポイント／リカバリによるオーバーヘッドの割合 r は

$$\begin{aligned}r &= \frac{\Gamma - T}{T} \\ &= \frac{\lambda^{-1}e^{\lambda C}(e^{\lambda(T+C)} - 1)}{T} - 1\end{aligned} \tag{11}$$

と示すことができ、この最小値 T_{opt} は

$$\begin{aligned}\frac{\partial r}{\partial T} &= 0 \\ e^{\lambda(T+C)}(1 - \lambda T) - 1 &= 0 \; for \; T \neq 0\end{aligned} \tag{12}$$

の解として求めることができる。

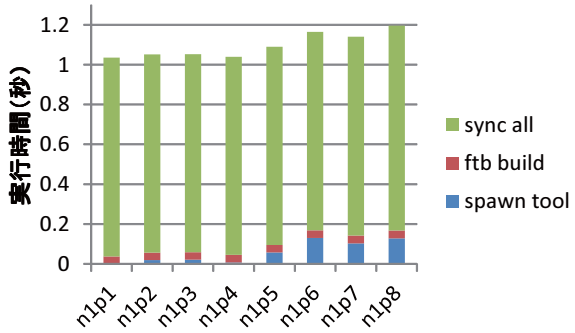

図 7　環境構築オーバーヘッド（ノード内プロセス数変更）

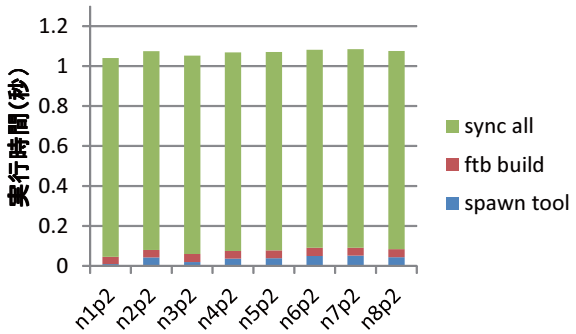

図 8　環境構築オーバーヘッド（ノード数変更）

4. 評価

故障が無い場合、実装したフレームワークのオーバーヘッドは主として環境構築のコストとチェックポイントのコストに分けられる。後者はアプリケーションに対して疑似故障発生を行い、リスタートを含めた総実行時間を取得する必要がある。しかしながら本研究においては疑似故障発生器、リスタート自動化部分の実装が未完成なため、前者の環境構築コストを評価する。すべての測定結果は 5 回の試行の平均値である。

4.1 評価環境

評価環境は、CPU が Intel Xeon X5550 (2.67GHz 4core) が 2 ソケット、Memory が 24GByte、GbE を 1 ポート備えたサーバ最大 8 機を利用した。また、OS は Linux 2.6.39.4、MPI 実装には MPICH-3.0.4 を利用した。ベンチマークアプリケーションとしては、フレームワークの初期化だけを行い、終了するプログラムを用いている。

4.2 ノード内プロセス数変更による負荷変化

まず、1 ノードを用い、アプリケーションの構成プロセス数を 1 から 8 まで変化させながら、環境構築の実行時間を測定した。構成プロセスに加え、管理プロセスが 1 プロセス実行されるため、実際のプロセス数は 2 から 9 となる。図 7 に spawn tool の実行時間 (spawn tool)、FTB の関連サーバ群を起動する時間 (ftb build)、実行開始前の同期時間 (sync proc) の積算を示した。X 軸の nXpY はノード数 X、ノード内プロセス数 Y を示し、Y 軸の単位は秒である。これを見ると、spawn tool の実行時間がプロセス数増大に伴い増加している。spawn tool は MPI_Allreduce, MPI_Gather を使うため、この性能が影響していると思われ、MPICH の詳細な性能測定が必要である。そのほかに関してはおおよそ同じ結果が出ており、最大でも 1.2 秒未満となった。科学技術計算は長時間にわたる場合が多く、ただ一度の初期化にかかる時間として 1.2 秒は十分小さいと言える。

4.3 ノード数変更による負荷変化

同様に、アプリケーションの構成プロセス数を 2（総プロセス数は 3 となる）で固定し、利用ノード数を 1 から 8 まで変化させながら、環境構築の実行時間を計測した。図 8 に結果を示す。凡例、X 軸、Y 軸に関してはノード内プロセス数変更の際と同じである。ノード数の変更に対して、実行時間の有意な変動は見られなかった。本フレームワークにおいて、ノード数の影響を受ける部分としては、FTB の関連サーバ起動時のディレクトリサービス、環境構築終了時の MPI_Barrier であるが、ノード内プロセス変更に比べ、プロセス増加の影響が少ないようである。これに関しても MPICH の詳細な性能測定が必要となる。そのほかに関しては、ノード内プロセス変更の時と同様、おおよそ同じ結果が出ており、最大でも 1.1 秒未満となった。これも十分小さな値と言える。

5. まとめと今後の課題

反復計算を主とする科学技術計算を対象として、チェックポイント／リスタートを容易に実装するフレームワークを提案、実装した。この際最適なチェックポイント頻度は、実行環境といった外的要因から決まるため、アプリケーションから与えられる一貫性保証アルゴリズムと、チェックポイント頻度算出アルゴリズムの両方を適用可能なアルゴリズムに関して提案を行った。フレームワークのオーバーヘッドは 1 秒前後で十分小さいと確認できた。

今後については、並列プロセスを通信パターンに基づきグルーピングする手法 [4] と、テネシー大学で MPI 標準化のために試験実装されている User Level Failure Mitigation[5] を用いることで、Coodinated な手法と Uncoordinated な手法をグループ内、外で使い分ける部分チェックポイントを構成していくことを考えている。これにより、部分リスタートが可能になる。一括リスタートと部分リスタートの性能差は大きいものではないが、すべてのプロセスを再計算する一括リスタートに比べ、一部分のプロセスのみを再計算すればよい部分リスタートは電力消費の面で大きなアドバンテージが存在する。また、現在はループに付き 1 回

のチェックポイントを仮定しているが、ループが十分に長いときには適切なチェックポイント間隔を指定できない可能性がある。このため、ループ内で複数回のチェックポイントが取れる手法を提案していく。

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－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] XU, X., YANG, X. and LIN, Y.: WBC-ALC : A Weak Blocking Coordinated Application-Level Checkpointing for MPI Programs, *IEICE transactions on information and systems*, Vol. 95, No. 3, pp. 786–796 (online), DOI: 10.1587/transinf.E95.D.786 (2012).

[2] 實本英之， 石川裕：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，情報処理学会研究報告. [ハイパフォーマンスコンピューティング]， No. 24, pp. 1–7（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/110008583354/⟩ (2011).

[3] 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).

[4] 轟侑樹，實本英之，佐藤三久， 石川裕：パーシャルメッセージロギングを改善する耐故障性実現フレームワーク，情報処理学会研究報告. 計算機アーキテクチャ研究会報告， No. 17, pp. 1–6（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/110009490628/⟩ (2012).

[5] Bland, W., Bosilca, G., Bouteiller, A., Herault, T. and Dongarra, J.: A Proposal for User-Level Failure Mitigation in the MPI-3 Standard, *University of Tennessee Electrical Engineering and Computer Science Technical Report* (2012).