Vol.2021-HPC-178 No.11
2021/3/15

ULFM を用いた動的プロセス再構成ランタイムの試作

住元 真司[1,a)]　塙 敏博[2]　中島 研吾[2,3]

概要：Urgent Computing に必要な並列プロセス数の動的変更とプロセス故障に対応するランタイムシステム DyProReconf(Dynamic Process Reconfigurable Runtime) の試作について述べる。ULFM を用いた DyProReconf を試作、評価した結果、動的な MPI プログラムの並列数の変更と耐故障性を確保したランタイムシステムが実現可能であることが分かった。一方、ランタイムシステム実現上の課題も判明した。

1. はじめに

エクサスケール以降のシステムでは、より一層のシステム巨大化が進むためシステム MTBF は一段と短くなり、システムだけでなくアプリケーション実行においても耐故障性の導入が必須となると想定される。また、システム運用においても、動的な使用電力調整や実時間性を要するジョブ実行により、動的なシステム構成変更が必要になりつつある。

運用中に動的なシステムの構成変更を実施する場合、現状の並列アプリケーションの多くは実行時の並列数が固定のものが多い。このため、運用中のシステム構成変更でアプリケーションが使用中の計算機資源量以下にするには実行中アプリケーションの中断や停止を伴う。柔軟で強固なシステムを実現するには、動的な運用構成変更に耐えられるシステム運用とそれに対応するアプリケーション構成技術が必要である。

現在我々は、地震、津波、水害等の自然災害発生時における緊急時のシミュレーション実行を実現する Urgent Computing[1], [2] 環境でのアプリケーションとシステムソフトウェア実現にむけ検討を進めている [3]。

本稿では、Urgent Computing に必要な並列プロセス数の動的変更とプロセス故障に対応するランタイムシステムの試作について述べる。試作の目的は、ランタイムシステムの試作を通してアプリケーションプログラムとシステムソフトウェアに必要な機能と実現上の課題を明らかにすることにある。

今回、ULFM を用いた DyProReconf(Dynamic Process Reconfigurable Runtime) を試作、評価した結果、動的な MPI プログラムの並列数の変更と耐故障性を確保したランタイムシステムが実現可能であることが分かった。一方、ランタイムシステム実現上の課題も判明した。本稿では DyProReconf 試作の概要と評価、並びに試作で判明した課題について述べる。

2. Urgent Computing

Urgent Computing は、地震，津波，水害等の自然災害発生時災害時の被害予測や避難計画策定などに大きな役割を果たすものと期待される。

この Urgent Computing について、論文 [2] は、Data Collection, Man-machine interaction そして Data Evaluation の機能を持ち、Reliability, Maintainability, Availability そして Security が必要と述べている。また、論文 [1] では Urgent Computing は “event-driven and deadline-based” であると述べている。

すなわち、Urgent Computing においては、事象発生的にシミュレーションに必要なデータを収集し、シミュレーション計算を安定的に実行し結果を得たうえで実施すべき対策を人間が一定時間内に判断できることが求められる。

これを実現するためには、データ収集とシミュレーション実行における事象発生時の即時実行可用性、実行を継続できるシステム信頼性、そして、システム故障発生時においても最速でシミュレーション実行可能な状態に戻す保守性が求められる。

3. Urgent Computing と計算機センタ運用

第 2 章で述べた Urgent Computing を計算機センターで

[1] 富士通
Fujitsu Ltd.
[2] 東京大学 情報基盤センター
The University of Tokyo
[3] 理化学研究所　計算科学研究センター
RIKEN Center for Computational Science
[a)] sumimoto.shinji@jp.fujitsu.com

実現する場合、緊急実行すべき事象が発生した場合に直ちにデータ収集とシミュレーション実行に必要な計算機リソースを割り当てる必要がある。しかし、緊急実行に必要な計算機リソースを常に確保しておくのは現実的でないため、事象発生時にシステム運用を止めることなく動的に必要な計算リソースを割り当てられるのが望ましい。

事象発生時にシステム運用を止めることなく必要な計算リソースを割り当てるには、事象発生時に必要なだけリソースに空きがあれば問題ないが、空きがない場合は実行中のアプリケーションを停止または中断する必要がある。これは、既存のアプリケーションの多くが実行時に計算リソースを固定して実行するものが多い上、計算機センターで利用される計算リソース管理システム (ジョブスケジューラ) が計算終了まで固定的にリソース割り当てを行うためである。

以上のように、実行中アプリケーションの停止・中断のない Urgent Computing を実現するためには、アプリケーションならびに計算リソース管理システムなどのシステムソフトウェアが動的な計算リソース量変更に耐えられることが求められる。

4. Urgent Computing 向けアプリケーション実行環境試作目的と要件

第 3 章で述べたように Urgent Computing を実現するためには、アプリケーションと実行環境であるシステムソフトウェアが動的な計算リソース変更に対応する必要がある。本論文では、最初のステップとして動的計算リソースに対応する並列アプリケーションとその実行環境であるランタイムシステム試作を考える。本試作では、アプリケーション実現上の課題と実現環境上の課題を明らかにすることを目標とする。

Urgent Computing を実現するためのランタイムシステムの要件を以下に述べる。

事象発生時の計算リソース動的変更： 緊急事象発生時に計算リソースを速やかに変更可能

耐故障性の確保： アプリケーションがプロセス故障に対して継続実行可能であること。プロセス故障時に実行が止まらないだけでなく、故障リカバリのオーバヘッドが最小限であること。

故障模擬機能と外部インターフェイス機能の実現： 故障模擬機構を備え、外部インターフェイスを用いて指定されたプロセス数変更指示と故障模擬機能によるプロセス数に再構成可能なこと。

これ以降では、Urgent Computing を実現するランタイムシステム DyProReconf(Dynamic Process Reconfigurable Runtime) の試作について述べる。

5. DyProReconf の実装方針と設計

第 4 章を要件を実現するランタイムシステム DyProReconf の設計方針として、複数システムで動作するために可能な限り標準仕様ベースを考慮する。

これ以降は、DyProReconf の実装設計として、耐故障性実現方式、プロセス数削減方式、プロセス数増加方式、Checkpoint/Restart 処理との連携、システムソフトウェアとの連携、そして、言語インターフェイスを議論する。

5.1 耐故障性実現方式

アプリケーションの耐故障実現はその実行形式により大きく実現手法が異なる。アプリケーションの実行形式として、1) 入力データが固定で再実行可能な再実行可能型、2) 実時間で入力されるリアルタイム処理型のアプリケーションに分類される。

また、耐故障実現の方式としては、次の 3 つに分類される。

- 計算中断再実行方式：アプリケーション実行中断を伴い、アプリケーションを再実行する。
- 冗長リソース代替計算継続方式：予め代替計算リソースを確保し冗長実行するか、故障時に計算リソースを割り当てアプリケーション実行を継続する。
- 縮退リソース計算継続方式：代替リソースを確保せず故障部を除去してアプリケーション実行を継続する。

DyProReconf の対応するアプリケーションとしては再実行可能型を想定し、耐故障実現の方式としては、計算中断再実行方式、冗長リソース代替計算継続方式、縮退リソース計算継続方式、すべてが実行できることとする。

複数のプロセスから構成されるアプリケーションの再実行性を確保するには、各プロセスの状態の他、通信の状態についても再実行が可能なように、各状態を不足なく引き継げる必要がある。

各プロセスの実行状態を格納する方式として、システムソフトウェアレベルでプロセス状態を格納するシステムレベルでの Checkpoint/Restart 方式とアプリケーションで計算継続に必要なデータ状態を格納するアプリケーションレベルの Checkpoint/Restart 方式があるが、DyProReconf ではシステム毎の制約をうけないアプリケーションレベルのチェックポイントを想定して実装する。

通信状態の格納方式としては、アプリケーションレベルでの Checkpoint/Restart 方式を前提とすると、アプリケーションの再実行時には通信状態を格納する必要がない。アプリケーションを継続実行する場合には、MPI Forum で規格化が進行中の ULFM (User Level Fault Mitigation) ベースでの実装を考える。アプリケーションの実装性と可搬性を考慮すると、標準規格ベース、かつ、オープンソー

スペースが適しているため、Open MPI ベースの ULFM で実装する。

5.2 プロセス数削減方式

MPI プログラムにおけるプロセス数を減らす手法として ULFM の機能を利用し、自プロセスへのシグナル通知を用いることで実現可能である。しかし、ULFM の自プロセスに対してシグナルを送ることによるプロセス数減少の時間を測定したところ、プロセス数に応じて時間がかかることが分かった。

表 1 プロセス減少時の MPI_Comm_Shrink 復帰時間
Open MPI v4.0.x+ulfm

# of decrease	8 (12 to 4)	20 (24 to 4) procs
w/o mpi_ft_detector_thread	151 sec	392 sec
w/ mpi_ft_detector_thread	15 sec	39 sec

表 1 は、ULFM 開発ツリーの v4.0.x+ulfm ブランチを用いての評価結果である。評価システムは Oakbridge-CX システムの 1 ノードを用いた。

表 1 において 12 から 4 プロセスに減らした結果と 24 プロセスから 4 プロセスに減らした時の MPI_Comm_Shrink 復帰時間を調べたところ、8 プロセス減らすのに 151sec、20 プロセス減らすのに 392sec も必要とした。開発者から聞いた mpi_ft_detector_thread オプション付きの場合、それぞれ 15 sec、39 sec とほぼ実行時間が 1/10 に短縮された。

時間が短縮し改善したが、20 プロセス減少するのに 39sec も必要とするのは問題である。これは大規模なプロセス数の減少には問題となりうるために、改善が必要である。

ULFM を使う場合に時間がかかるのは、プロセス故障検知後に生存プロセスの応答を待つためであることがわかった。これを改善するためには ULFM を変更する必要があるが、実装は容易ではない。

このため、ULFM を使わない手法を調べた結果、生存プロセスと削減予定プロセスを MPI_Comm_split で分類し、削減予定プロセスは自プロセスで MPI_Finalize を呼ぶ (MPI_Comm_split+MPI_Finalize 方式) ことで目的の動作を代行できることがわかった。

表 2 に、MPI_Comm_split+MPI_Finalize 方式での経過時間評価の結果を示す。同様に評価システムは Oakbridge-CX システムの 1 ノードを用いた。

表 2 において 12 から 4 プロセスに減らした結果と 24 プロセスから 4 プロセスに減らした時のプロセス削減時間を調べたところ、MPI_Comm_split+MPI_Finalize 方式は 1 sec(0.1-0.2) 以下と必要十分な時間でプロセス数削減できることがわかった。

表 2 プロセス減少時の経過時間
Open MPI v4.0.x+ulfm

# of decrease	8 (12 to 4)	20 (24 to 4) procs
ULFM w/o mpi_ft_detector_thread	151 sec	392 sec
ULFM w/ mpi_ft_detector_thread	15 sec	39 sec
MPI_Comm_split + MPI_Finalize	0.1 sec	0.2 sec

5.3 プロセス数増加方式

MPI プログラムでは、MPI プログラム実行中のプロセス生成は、MPI_Comm_spawn を利用することで実現可能である。実際には、MPI_Comm_spawn を実行すると MPI の通信リソースであるコミュニケータが生成したプロセス群間で独立して作られるため、MPI_Comm_merge 関数によりコミュニケータを連結して使用することになる。

MPI_Comm_spawn が実行されると、生成されたプロセス群は呼び出されたプログラムの main 関数から実行されるため、呼び出されたプログラムが呼び出し側のプログラムと同じ場合は、MPI_Comm_spawn を呼びだすプロセスと呼び出されて生成されたプロセス間で、コミュニケータ連結後に同じプログラムの再実行箇所でアプリケーションが再実行可能な状態に戻せるようにする必要がある。

このため、ULFM でプロセス故障時に残存プロセスが再実行個所として利用する setjmp で設定される再実行地点を利用する。具体的には MPI_Comm_spawn を呼びだすプロセスは MPI_Comm_spawn 実行後にこの setjmp で設定された再実行箇所に longjmp し、MPI_Comm_spawn で生成されたプロセス群は main 関数の実行開始後に初期設定の完了後、longjmp で設定された再実行箇所で合流できるようにプログラムする。

このように実装することで、プロセス増加、プロセス故障によるプロセス減少を含むプロセス数変更時のアプリケーションの再実行処理を 1 か所にまとめることができ、プログラムの簡略化が可能である。

5.4 Checkpoint/Restart 処理との連携

DyProReconf ではアプリケーションによる Checkpoint/Restart 処理を前提としている。このために、Checkpoint/Restart 処理との連携においてアプリケーションにとって使いやすいインターフェイスである必要がある。

アプリケーションによる Checkpoint/Restart 処理においては、データの一貫性が確保できる実行箇所と格納データの内容はアプリケーションにより異なる。このため、Checkpoint 処理の記述と処理の呼び出し箇所の指定は柔軟に指定できる必要がある。一方、Resttart 処理については、その処理記述はアプリケーションに依存するが、処理の呼び出し箇所の指定はランタイムシステム側のしかるべ

き箇所で呼び出される必要がある。

このように、DyProReconf における Checkpoint/Restart 処理との連携は、Checkpoint/Restart 処理での格納データ処理はアプリケーションで実装し、それぞれの処理の呼び出しは、Checkpoint 処理についてはアプリケーションで呼び出し、Restart 処理についてはランタイムシステムで呼び出す必要がある。

5.5 システムソフトウェアとの連携

Urgent Computing においては、事象発生時にシステムの計算リソース管理システムなどのシステムソフトウェアと連携してシミュレーションを実行する必要がある。これは、単に事象発生時に実行されるアプリケーションのみならず、事象発生時に既に実行中のアプリケーションとの動的な連携が必要である。

事象発生時にシステムに実行されるアプリケーションの使用計算リソースの空きが足りない場合は、2 つの施策が考えられる。これから実行されるアプリケーションがより少ないリソースで実行できるか、あるいは、既に実行中のアプリケーションから必要な使用計算リソースを確保できるかである。

しかし、どうすべきかは、実行されるアプリケーションにも依存するため、今回の実装では、事象発生時に実行されるアプリケーションは固定の使用計算リソースを確保し、既に実行中のアプリケーションから必要な計算リソースを確保することを前提で連携を考慮する。

そのうえで、計算リソース管理システムから動的にアプリケーションのプロセス数の増減を指示するインターフェイスを試作する。

5.6 言語インターフェイス設計

試作ランタイムで利用する ULFM は、プロセス故障時に例外処理を用い MPI 環境を復帰することで MPI プログラムの継続実行を実現している。例外処理からの復帰は setjmp 関数により予め設定した箇所に longjmp 関数を呼び再実行する。再実行時には、予め格納したデータを用いてデータの復旧後、再実行する。

しかし、setjmp/longjmp は言語により利用できないため、アプリケーションへの適応性を考慮すると C 言語でランタイムを試作し、そのランタイムから各言語処理を呼び出すことでプログラミングスタイルを統一可能である。

5.6.1 C 言語インターフェイス設計

図 1 に、典型的な ULFM を用いた setjmp/longjmp の C 言語によるプログラムを示す。

main_loop の先頭で call_checkpoint 関数によりチェックポイント処理を実行後 setjmp により実行コンテキストを格納し、call_calc() で計算処理を実行する。

プロセス故障が発生すると ULFM が故障を検知し例外ハンドラを呼び出し生存プロセスを用いて MPI 通信コンテキストであるコミュニケータを再構成する。この例外ハンドラにおいて実行環境を再設定した後 longjmp により setjmp 呼び出し地点に復帰する。

```
/* begin sample code */
jmp_buf jmp_buf;

exception_handler(MPI_Comm *comm, context...) {
   ....
   /* ULFM processing.. */
   MPIX_Comm_revoke(comm);
   MPIX_Comm_shrink(comm, &newcom);
   setup process using newcom communicator...
   ....

   longjmp(jmp_buf, 1);
}

main_loop(void) {
   int loop = 1;
   while(loop) {
        call_checkpoint();
        if (setjmp(jmp_buf) != 0 ) {
           call_restore();
        }
        loop = call_calc();
    }
}
/* end sample code */
```

図 1 典型的な setjmp/longjmp(C 言語)

setjmp 関数は自ら呼び出した場合の復帰値は 0 で longjmp 関数からの復帰時には longjmp 関数の第 2 引数値を返り値として復帰する。setjmp 関数復帰時には予め格納した実行コンテキストからデータを再構築して実行を再開する。

アプリケーションがすべて C 言語で書かれている場合は図 1 に示すように制約なく実現可能である。

5.6.2 FORTRAN 言語インターフェイス設計

FORTRAN は一部の実装を除いて setjmp/longjump をサポートしていないため実行コンテキストの格納は C 言語で、checkpoint/resotre については C 言語並びに計算処理記述言語での実装を考慮する必要がある。

このため、例えば図 2 に示す通り、実際の制御系処理は C 言語で記述し、計算処理は FORTRAN で記述することで実現可能である。ここで、checkpoint/resotre 処理自体は計算記述言語で記述してもよい。

本記述スタイルは FORTRAN 以外の言語に対しても適用可能である。

6. 試作実装概要

本章では、DyProReconf の試作実装の概要について述べる。第 5 章の内容について、それぞれの概要、実装関数、そして制御構造について述べる。

```
/* begin sample code */

jmp_buf jmp_buf;

exception_handler(MPI_Comm *comm, context...) {
   ....
   /* ULFM processing.. */
   MPIX_Comm_revoke(comm);
   MPIX_Comm_shrink(comm, &newcom);
   setup process using newcom communicator...
   ....

   longjmp(jmp_buf, 1);
}

main_loop(void) {
   int restore = 0;
   int loop = 1;
   while(loop) {
        restore = 0;
        if (setjmp(jmp_buf) != 0 ) {
            restore=1;
        }
        loop = call_calc(restore);
    }
}

call_checkpoint() {
   ... do checkpoint
}
call_restore() {
   ... do resotre
}

/*======================*/
/* C 言語以外も記述可能: FORTRAN, C++ etc. */
int call_calc(int restore) {
    if(resotre) {
        call_restore();
    }
    else {
        call_checkpoint();
    }
    ... do calcuration.
    if( calcuration is end ) {
        return 0;
    }
    else {
        return 1;
    }
}
/* end sample code */
```

図 2 C 言語以外の実行を考慮したプログラム実行

6.1 実装設計への対応

本節では、第 5 章で述べた実装設計への対応を述べる。

耐故障性実現方式 耐故障性実現方式の実装として ULFM[4] を採用した。ソースコードは Open MPI v4.x ベースの branch[*1] を採用した。

プロセス数削減方式 MPI_Comm_split+MPI_Finalize 方式で実装、プロセス数変更とプロセス障害復帰時の復帰ポイントを setjmp にて保存、定期的に checkpoint したデータを setjmp 復帰時に Restart 処理して再実行開始

プロセス数増加方式 MPI_Comm_spawn を利用、全プロセスは setjmp 呼出し箇所に合流して各プロセスで Restart 処理して再実行開始

Checkpoint/Restart 処理との連携 まずは、固定的なインターフェイスを定めず、アプリケーションで Checkpoint/Restart 処理を記述し、連携手法を検討する。ランタイムシステムでは、Checkpoint/Restart 処理を実行すべき箇所を指定する。

システムソフトウェアとの連携 ファイル経由のコマンド受け取り関数を設け、一定間隔でアプリケーションから呼び出すことで実行する。まずは、簡易なインターフェイスとしてファイル経由の連携を試作する。仕様は次の通り、

- プロセス数を書く：現状のプロセス数より小さい場合には、 MPI_Comm_split+MPI_Finalize 方式にてプロセス数削減する。大きい場合は MPI_Comm_spawn でプロセス数増加する。
- 故障模擬は k*rank* 番号,*rank* 番号,…: 対象の rank 番号のプロセスは自爆、コマンド実行後の rank 数を書いてコマンド実行完了

言語インターフェイス まずは、ランタイムシステム自体は C 言語を用いて記述し試作する。具体的な言語間のインターフェイスは論文 [3] で開発中のアプリケーションへの適応で検証していく。

6.2 実装関数

図 3 に、DyProReconf の制御構造を示す。不要なインターフェイスは設けず、シンプルな制御構造としている。

今回、追加した 3 つの関数の概要を次に示す。

- ulfm_dyprc_merge_child (MPI_Comm mworld) spawn プロセスと元プロセスをマージする。アプリケーション実行開始時に実行し、返り値は spawn により生成されたプロセスの場合は 1、そうでない場合は 0 を返す。
- ulfm_dyprc_setjmp (MPI_Comm mworld , int newcomer) : setjmp を内部で実装、コマンド実行とプロセス故障時の全プロセス同期を内部実行する。返り

[*1] https://bitbucket.org/icldistcomp/ulfm2/src/ulfm/

```
/* begin sample code */
jmp_buf jmp_buf;
MPI_Comm mworld;

/* ULFM Process Fault Handler  */
exception_handler(MPI_Comm *comm, context...) {
   ....
   /* ULFM processing.. */
   MPIX_Comm_revoke(comm);
   MPIX_Comm_shrink(comm, &newcom);
   setup process using newcom communicator...
   ....

   longjmp(jmp_buf, 1);
}

/* Application Main routine */
int call_calc(void) {
   int i;
   for(i=0; i < LOOP; i++) {
      do_calc();
      ulfm_dyprc_poll(mworld); /* call polling function */
    }
    return 0;
}

main_loop(int newcomer) {
    int loop = 1;
    while(loop) {
        if (ulfm_dyprc_setjmp(mworld, newcomer) != 0 ) {
           call_restore(); /* application restore */
        }
        else {
           call_checkpoint(); /* application checkpoint */
        }
        loop = call_calc();
    }
}

main(int argc, char *argv[]) {
   MPI_Init(&argc, &argv);

   if((newcomer = ulfm_dyprc_merge_child (&mworld)) != 0) {
      /* if MPI_Comm_spawn ed process return non Zero*/
      call_application_setup_for_spawned_process();
   }
   else {
      call_application_normal_setup();
   }
   main_loop(newcomer);
}
/* end sample code */
```

図 3 DyProReconf の制御構造

値はプロセス故障とプロセス数変更の場合、1 を返して復帰する。また、newcomer が 1 の場合 (spawn されたプロセスを想定) も 1 を返して復帰する。

- ulfm_dyprc_poll (MPI_Comm mworld): 外部プロセスからのコマンド実行のための polling 実行する。

図 3 において、図の下から main, main_loop 関数共にアプリケーション依存の部分は機能ごとに別関数で切り出すことによりファイルを分離記述を可能にしている。calc_calc 関数部分をアプリケーション毎に記述することで記述言語の変更にも対応可能としている。

7. 試作評価

7.1 評価環境

試作ランタイムシステムの評価環境は、表 3 に示すように、東京大学　情報基盤センターの Oakbridge-CX と Oakforest-PACS である。インタコネクトは Intel Omni-Path である。ただし、ULFM は OmniPath 対応の psm2 デバイスに対応していないため、評価は通信リソースとして共有メモリと TCP/IP 利用のデバイスで実施した。

表 3 評価環境

	Oakbridge-CX	Oakforest-PACS
CPU	Intel Xeon Platinum 8280	Intel Xeon Phi 7250
# of cores	56 (28+28)	68
Clock Speed	2.7 GHz	1.4 GHz
# nodes	1368	8208
Interof conect	Intel Omni Path	Intel Omni Path
Topology	Fat Tree	Fat Tree

将来的には、2021 年 5 月に導入予定の Wisteria/BDEC-01(Intel/Arm) での稼働も視野にいれる。

7.2 評価項目

基本機能の評価として、プロセス数増減とプロセス故障機能が動作することを確認する。評価の内容は次の通りである。

プロセス数増減　ノード数可変の sendrecv ベンチマークを作成し評価する。プロセス数増減を外部スクリプトから操作してアプリケーション実行継続を確認する。

任意プロセス故障　ランダムにプロセス故障を発生させてアプリケーション実行継続を確認する。

試作ランタイムの評価においては、コマンドインターフェイスに指定時間にコマンドを書き込む perl スクリプト作成し評価した。利用可能なコマンドは次の通りである。

- *m*: 全体プロセス数を *m* にする。現状プロセス数の差分を MPI_Comm_spawn かシグナルで終了させる。
- *sec*:k*rank*: *sec* 秒後に *rank* 番号をシグナルで終了、
- *sec*:r*maxp*： *sec* 秒後に最大 *maxp* 番号の乱数生成

```
$ ./opt/ulfm2/bin/mpirun -np 32 ./test
Hello MPI World rank/size=0/32, comm=b359b0(56)@obcx01.obcx

MPI_Sendrecv(0/4) comm_size 32, len 1 time 0.020 msec.
MPI_Sendrecv(0/4) comm_size 32, len 2 time 0.021 msec.
MPI_Sendrecv(0/4) comm_size 32, len 4 time 0.020 msec.
....<中略>
MPI_Sendrecv(0/4) comm_size 32, len 131072 time 4.836 msec.
MPI_Sendrecv(0/4) comm_size 32, len 262144 time 9.327 msec.
  ++ ULFM -- Now Restarted rank8/16 procs 2228940 -- ++
MPI_Sendrecv(0/4) comm_size 16, len 524288 time 9.234 msec.
MPI_Sendrecv(1/4) comm_size 16, len 1 time 0.025 msec.
MPI_Sendrecv(1/4) comm_size 16, len 2 time 0.019 msec.
....<中略>
MPI_Sendrecv(1/4) comm_size 16, len 131072 time 2.447 msec.
MPI_Sendrecv(1/4) comm_size 16, len 262144 time 4.746 msec.
MPI_Sendrecv(1/4) comm_size 16, len 524288 time 9.275 msec.
 +- Restarting Proc(13/24), new 24/24
MPI_Sendrecv(2/4) comm_size 24, len 1 time 0.135 msec.
MPI_Sendrecv(2/4) comm_size 24, len 2 time 0.100 msec.
MPI_Sendrecv(2/4) comm_size 24, len 4 time 0.050 msec.
MPI_Sendrecv(2/4) comm_size 24, len 8 time 0.034 msec.
....<中略>
MPI_Sendrecv(2/4) comm_size 24, len 262144 time 4.408 msec.
MPI_Sendrecv(2/4) comm_size 24, len 524288 time 8.626 msec.
  ++ ULFM -- Now Restarted rank2/4 procs 29861f0 -- ++
MPI_Sendrecv(3/4) comm_size 4, len 1 time 0.018 msec.
MPI_Sendrecv(3/4) comm_size 4, len 2 time 0.019 msec.
....<中略>
MPI_Sendrecv(3/4) comm_size 4, len 65536 time 0.228 msec.
MPI_Sendrecv(3/4) comm_size 4, len 131072 time 0.385 msec.
MPI_Sendrecv(3/4) comm_size 4, len 262144 time 0.741 msec.
MPI_Sendrecv(3/4) comm_size 4, len 524288 time 1.819 msec.
ByeBye MPI World rank/size=0/4, comm=b0c6b0
```

図 4 プロセス数増減評価結果

で 1 プロセスをシグナルで終了

- *sec*:R*maxp*:*n*: *sec* 秒後に 最大 *maxp* 番号の乱数生成で *n* プロセス数をシグナルで終了

7.3 評価結果

プロセス数増減評価の結果、ノードを跨るプロセス増加については動作しなかった。プロセス数減少については動作した。このため図 4 に、Oakbridge-CX の 1 ノードを用いたプロセス数増減評価の結果を示す。図 4 の出力で comm_size がコミュニケータのサイズである。本試験では最初 32rank で実行開始したプログラムが 16rank, 24 rank, 4 rank 実行に変更されて終了するというプログラムである。図 4 中、MPI_Sendrecv(3/4) は全体 4 回繰り返しの 3 回目を示し、rank 数変更によっても実行ループの回数が継続実行されていることを意味する。

図 5 に、Oakbridge-CX の 8 ノード 256 プロセスでの任意プロセス故障評価結果を示す。図 5 では、ランダムに 1 rank づつスクリプトでシグナルを送ってプロセス終了させた結果である。

図 5 の実行ログよりプロセス故障が通知されるごとに

```
Hello MPI World rank/size=0/256, comm=1b0e7c0(256)@cx0017.obcx
<<==== MPI_main40 rank0/256 ====
==== MPI_main40 rank0/256 time 20.075507 msec.====>>
....<中略>
<<==== MPI_main36 rank0/256 ====
==== MPI_main36 rank0/256 time 20.084623 msec.====>>
** Got Signaled and Change size(256->255): Error MPI_ERR_REVOKED\
: Communication Object Revoked, and go recover 76 (pid=213138)
  ++ ULFM -- Now Restarted rank0/255 procs 0x1bdafa0 (22 sec)-- ++
<<==== MPI_main35 rank0/255 ====
==== MPI_main35 rank0/255 time 20.125318 msec.====>>
....<中略>
<<==== MPI_main27 rank0/255 ====
==== MPI_main27 rank0/255 time 20.178514 msec.====>>
<<==== MPI_main26 rank0/255 ====
==== MPI_main26 rank0/255 time 20.051167 msec.====>>
** Got Signaled and Change size(255->254): Error MPI_ERR_PROC_\
FAILED: Process Failure, and go recover 74 (pid=213138)
  ++ ULFM -- Now Restarted rank0/254 procs 0x1c5e810 (62 sec)-- ++
<<==== MPI_main25 rank0/254 ====
==== MPI_main25 rank0/254 time 20.049377 msec.====>>
....<中略>
<<==== MPI_main16 rank0/254 ====
==== MPI_main16 rank0/254 time 20.053693 msec.====>>
** Got Signaled and Change size(254->253): Error MPI_ERR_PROC_\
FAILED: Process Failure, and go recover 74 (pid=213138)
  ++ ULFM -- Now Restarted rank0/253 procs 0x1c00460 (103 sec)-- ++
<<==== MPI_main15 rank0/253 ====
==== MPI_main15 rank0/253 time 20.027136 msec.====>>
....<中略>
<<==== MPI_main7 rank0/253 ====
==== MPI_main7 rank0/253 time 20.177024 msec.====>>
** Got Signaled and Change size(253->252): Error MPI_ERR_PROC_\
FAILED: Process Failure, and go recover 74 (pid=213138)
  ++ ULFM -- Now Restarted rank0/252 procs 0x1cb1c40 (141 sec)-- ++
<<==== MPI_main6 rank0/252 ====
==== MPI_main6 rank0/252 time 20.036606 msec.====>>
....<中略>
<<==== MPI_main2 rank0/252 ====
==== MPI_main2 rank0/252 time 20.072494 msec.====>>
<<==== MPI_main1 rank0/252 ====
ByeBye MPI World rank/size=  2/252, comm=188b6e0
==== MPI_main1 rank0/252 time 20.042927 msec.====>>
```

図 5 任意プロセス故障評価結果

256,255,254,253,252 とプロセス数が減少しており ULFM によりプロセス故障においてもアプリケーション実行が継続している結果である。実行ログの中で
++ ULFM – Now Restarted rank0/252 procs 0x1cb1c40 (141 sec)– ++
の出力が実際のリカバリ後の再スタートのログであるが、この最後の 141 sec はプロセス故障検出から再実行スタートまでの経過時間を示している。100 秒単位で時間が経過しており、再実行開始までの時間短縮が必要である。

以上のように、DyProReconf で実現すべき基本機能については動作を確認した。以下に評価結果をまとめる。

プロセス数増減 プロセス減少については 256 ノードまで試験し正常動作を確認、プロセス数増加については 1 ノード内での増加は正常動作、ノードをまたがる増加は異常終了であった。

任意プロセス故障 256 プロセスまで固定 rank 番号、ランダム rank 番号による模擬プロセス故障試験を実施しともに正常動作を確認。プロセス故障からの復帰時間が課題である。

本節では Oakbridge CX での結果を示したが Oakforest-PACS でも様の動作を確認している。今後、実プログラムでの checkpoint, restore でのプログラム実行で確認していく予定である。

7.4 評価でわかった課題

本節では、試作評価でわかった課題について述べる。

7.4.1 複数ノードでの MPI_Comm_spawn に対応不可

今回の評価では機能動作を確認するために 1 ノードでの評価としたが、複数ノードに跨った MPI_Comm_spawn は動作しなかった。Open MPI 内部を調査したところ PMIx の複数ノード間の spawn プロトコルの実装がないことが判明した。

このため、ULFM と Open MPI の開発コミュニティに確認したところ、PMIx 対応プロセス管理システムである orte の実装不備の問題であることがわかった。

現在開発中の Open MPI v5.x では対応しているとの情報を入手した。しかし、MPI_Comm_spawn の複数ノードでの動作確認ができたが、プロセス故障に対する動作が動作確認できなかった。現在、ULFM, Open MPI, PMIx コミュニティと連携しながら調査を進めている。

7.4.2 OminiPath の通信ライブラリ libpsm2 は ULFM 未対応

ULFM のサイト [4] によれば、ULFM のサポートしている通信デバイスは次の通りである。

- Loopback (send-to-self)
- UCX (beta)
- TCP
- OpenFabrics: InfiniBand, iWARP, and RoCE
- uGNI (Cray Gemini, Aries)
- Shared memory Vader (FT supported w/CMA, XPmem, KNEM untested)
- Tuned, and non-blocking collective communications

Oakbridge CX と Oakforest-PACS のインタコネクトである Intel Omni Path はサポートされていない。実際のシステム運用での利用に TCP プロトコルしか使えないのは問題である。

この中で TCP 以外に Intel Omni Path で動作すると考えられるのは OpenFabrics(OpenIB, libfabric) であり、試行調査をしてわかった結果は次の通りである。

- Verb 対応で ULFM が対応している OpenIB は実行不安定
- 現状のところ TCP,self ノード内 のみ安定動作
- Open MPI は今後 UCX 中心での実装に移行： UCX の OmniPath 対応か libpsm2 の ULFM 対応かの選択肢くらいの対応

今後の進め方として、Intel Omni Path は既に今後の製品化は停止しているため、将来システムでの稼働を考慮する必要がある。選択肢としては UCX の ib verbs 対応と InfiniBand 他のインタコネクト対応がある。このため、まずは、現状稼働リストにある InfiniBand での ULFM の動作確認を進め安定性を検証していく。評価を進めるにあたり、ULFM コミュニティと協力して問題解決にあたる。

7.4.3 プロセス故障に対するリカバリ時間が長い

既に 5.2 節で述べたように、プロセス故障の際のプロセス再構成の場合、8 プロセス減らすのに 151sec、20 プロセス減らすのに 392sec も必要とした。高速化オプションをつけた場合にも 15sec から 39 sec 必要であった。これはプロセス規模が大きくなるにつれて、復帰時間も長くなる傾向が観測されている。故障発生時のプロセス再構成時間の高速化が必要である。

8. 関連研究

ここでは、実際にアプリケーションが取りえる耐故障手法について整理し、DyProReconf との関連性を議論する。

既存の耐故障手法としては先に述べた 計算中断再実行方式、冗長リソース代替計算継続方式、縮退リソース計算継続方式であるが、どの方式においても故障時に故障ノードの持つ情報を何らかの形で引き継ぐ必要がある。このために耐故障機構は、定期的に一貫性が確保された情報を他のノードがアクセスできる形で格納している。この手法として良く用いられるのが Checkpoint/Restart 方式である。

Checkpoint/Restart 方式は、ノード故障が発生した際には、代替ノード、もしくは、残ったノード群で再分配することにより再実行 (リスタート) する。Checkpoint/Restart 方式には、システムレベルとアプリケーションレベルがあるが、システムレベルは OS を含むシステムレベルのサ

ポートが必要なうえ、格納するデータ量が巨大になるため、アプリケーションが再実行に必要な情報のみ格納するアプリケーションレベルのチェックポイント・リスタート方式が多く用いられている。DyProReconf においては、ランタイムシステムの可搬性とオーバヘッドの最小化の観点からアプリケーションレベルでの Checkpoint/Restart の利用を前提とする。

Checkpoint/Restart 方式には、様々な高速化手法が研究され利用されているが、並列計算機においては、複数ノード間の通信で実際にチェックポイントデータが変更されるため、複数ノード間での通信と連携してデータの一貫性のあるチェックポイントを実行する必要がある。　耐故障を実現するライブラリとフレームワークとしていくつかの既存研究がある。その概要を次に示す。

- ULFM (User Level Fault Mitigation)[4]:
  FT-MPI(Fault Tolerant MPI) などの耐故障通信ライブラリの研究が発展して標準の MPI 通信ライブラリ規格化を進めており、一部の機能は MPI 規格として規定されている。並列計算機上の通信ライブラリとして、上述した冗長リソース代替計算継続方式と縮退リソース計算継続方式の 2 つの耐故障機構を実現可能な MPI 上での通信ライブラリ仕様であり実装である。[5]
- Fenix[*2]:
  ULFM を用いて MPI プログラムの耐故障処理をサポートするアプリケーションフレームワークである。既存の MPI プログラムに幾つかの処理を加えることで耐故障性を実現する。機能としてはプロセス故障により通信コミュニケータの耐故障処理とデータリカバリ処理を提供する。プロセス故障への対応は故障時に新たなプロセスを割り当てて冗長リソース代替計算継続方式と残りのプロセス群で処理を継続する縮退リソース計算継続方式をユーザ選択で実現可能できる。また、データのリカバリにおいても Non-local Data ベースと、Local Data ベースのデータリカバリが選択可能である。[6]
- Resilient Kokkos[*3]：
  kokkos は C++ベースで性能可搬性を実現するプログラミングモデルであるが、Kokkos のモデルに Checkpoint/Restart 機能の実装が ECP プロジェクトの一つとして進行中である。[7], [8]
- CRAFT(Checkpoint-Restart and Automatic Fault Tolerance) ライブラリ〔Shahzad et al. 2018〕:
  FAU Erlangen/Nuremberg で開発された数値ライブラリで，ULFM-MPI（User Level Failure Mitigation）を使用することによって，MPI プログラム実行中にノード故障が発生してもアプリケーションの実行を継続可能であり，スペアノードの有無の両者に対応している。CRAFT は並列固有値ソルバー〔Shahzad et al. 2018〕，並列有限要素法〔Fukasawa et al. 2018〕など様々なアプリケーションに適用されている。大規模並列環境では，ULFM-MPI のような耐故障性を有する通信ライブラリの存在が不可欠であり，システムソフトウェアグループとの密接な協力が必要である。[9], [10]
- 故障フリー計算手法他 (Approximate Computing):
  一つのアプローチとして、計算手法自体を耐故障化容易な手法を用いるものがある。例えば SPMD に比べ Master-Worker 方式や MapReduce 方式などは故障発生時の耐故障実装が容易である。　また、厳密解を必要とせずある程度の計算誤差を許す問題の場合は故障発生時に故障プロセスを切り離すだけで処理を続けても精度に影響するが許容範囲な場合があり得る。あるいは計算原理として積極的な近似計算を実行する場合はプロセス故障が発生しても、そのプロセス故障を近似誤差として取り込める場合があり得る。[11]

以上述べた耐故障を実現するライブラリとフレームワークと DyProReconf との関連性ついては、DyProReconf はアプリケーションレベル Checkpoint/Restart 方式を用いて ULFM ベースでの実装している点で Resilient Kokkos, CRAFT と類似点がある。しかし、耐故障性という観点だけでなく Urgent Computing を目的とした動的な計算リソース変更に対応したシステム運用を目指している点で異なる。また、故障フリー計算計算手法との違いは、計算手法により耐故障性を確保するのではない点が異なる。

9. まとめ

本稿では、Urgent Computing に必要な並列プロセス数の動的変更とプロセス故障に対応するランタイムシステム DyProReconf の試作について述べた。

ULFM を用いた DyProReconf を試作、評価した結果、動的な MPI プログラムの並列数の変更と耐故障性を確保したランタイムシステムが実現可能であることが分かった。一方、ランタイムシステム実現上の課題も判明した。

今後の予定としては、論文 [3] で開発中のアプリケーションへの適用を進めると共に、今回明らかになった課題の解決と Urgent Computing 実現を進めていく。

謝辞　本研究の一部は科学研究費補助金（19H05662，代表：中島研吾）の助成を受けたものである。

参考文献

[1] Pete Beckman, Ivan Beschastnikh, Suman Nadella, and Nick Trebon C. Building an infrastructure for urgent computing. high performance computing and grids in ac-

*2 https://github.com/epizon-project/Fenix
*3 https://github.com/kokkos/kokkos

tion, 2007.
[2] Siew Hoon Leong and Dieter Kranzlmüller. Towards a general definition of urgent computing. *Procedia Computer Science*, Vol. 51, pp. 2337 – 2346, 2015. International Conference On Computational Science, ICCS 2015.
[3] 中島研吾, 住元真司, 塙敏博. Urgent computing に向けたアプリケーション. 情報処理学会研究報告 21-HPC-178. 情報処理学会, Mar 2021.
[4] ULFM: https://fault tolerance.org/.
[5] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J. Dongarra. An evaluation of user-level failure mitigation support in mpi. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, pp. 193–203, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
[6] M. Gamell, D. S. Katz, K. Teranishi, M. A. Heroux, R. F. Van der Wijngaart, T. G. Mattson, and M. Parashar. Evaluating online global recovery with fenix using application-aware in-memory checkpointing techniques. In *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, pp. 346–355, Los Alamitos, CA, USA, aug 2016. IEEE Computer Society.
[7] kokkos: https://sc19.supercomputing.org/proceedings/tech_poster/poster_files/rpost188s2-file3.pdf.
[8] resilient kokkos: https://ecpannualmeeting.com/assets/overview/sessions/ECP-Breakout-Teranishi.pdf.
[9] F. Shahzad, J. Thies, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein. Craft: A library for easier application-level checkpoint/restart and automatic fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 30, No. 3, pp. 501–514, 2019.
[10] T. Fukasawa, F. Shazad, K. Nakajima, and G. Wellein. pfem-craft: A library for application-level fault-resilience based on the craft framework. *Poster at the 2018 SIAM Conference on Parallel Processing for Scientific Computing (SIAM PP18)*, 2018.
[11] J. Han. Introduction to approximate computing. In *2016 IEEE 34th VLSI Test Symposium (VTS)*, pp. 1–1, 2016.