Vol.2014-HPC-145 No.8
2014/7/28

# 耐障害性ミドルウェアfalanxへの高可用分散協調スケジューラの実装

竹房 あつ子[1,a)]　中田 秀基[1]　池上 努[1]　戸澤 貴之[1]　田中 良夫[1]

概要：我々は，エクサスケール計算機環境において耐障害性を有した階層型タスク並列アプリケーションプログラムの開発を支援する Falanx ミドルウェアを開発している．Falanx は，データストア機構と資源管理機構からなり，アプリケーションプログラムを単一 MPI ジョブとして実行させるとともに，実行障害発生時には選択的にタスクを再実行／破棄させることができる．しかしながら，Falanx の既存実装では資源管理機構を MPI プロセスの 1 ランクで処理しているため，耐障害性およびスケーラビリティに課題が残っている．本稿では，Falanx システムの耐障害性およびスケーラビリティを高めることを目指し，既発表研究で提案した高可用分散協調スケジューラを ZooKeeper を用いて Falanx ミドルウェアに実装した．予備実験を行った結果，細粒度のタスクを大量に処理する場合には高可用分散協調スケジューラのオーバヘッドが非常に大きくなることが分かり，タスク取得処理の効率化が必要不可欠であることが示された．

キーワード：エクサスケールコンピューティング，耐障害性，MPI，高可用性，分散協調スケジューラ

## 1. はじめに

エクサスケール計算機は，十万プロセッサ，数千万 CPU コア規模で構成されるため，その故障発生間隔 (MTBF) は 1 日未満になると言われている [1], [2]．よって，エクサスケール計算機環境のランタイムシステムには，アプリケーションプログラムの実行中に計算機の一部が故障することを前提として，故障箇所を検知しアプリケーションプログラムに故障発生時の対処を委譲する耐障害性（Fault Resilience）が求められる．また，タスク並列の各タスクに並列処理を内包させた階層型タスク並列処理は，弱スケーリングが比較的容易であること，障害発生時にタスクごとに再計算するなど耐障害性設計が可能であるため，エクサスケール計算機環境における有望なプログラミングモデルの 1 つと考えられている [3], [4]．

我々は，エクサスケール計算機環境において耐障害性を具備した階層型タスク並列アプリケーションプログラムの開発を支援する Falanx ミドルウェアを開発している [4], [5], [6]．Falanx は，テネシー大で開発されている User Level Fault Mitigation (ULFM) MPI[7] を用いて開発されており，実行時にデータストア機構と資源管理機構を動的に構築して，障害発生時においても MPI アプリケーションを継続して実行する仕組みを提供する．データストア機構では冗長管理するオンメモリストレージを提供し，計算の中間データを安全に保管する [6]．一方，資源管理機構では，各計算ノードの健全性を監視しつつ，遊休計算ノード群にデータ並列タスクを割り当て，障害発生時には失敗したタスクを選択的に再実行または破棄する．しかしながら，現在の Falanx の実装では資源管理機構には MPI プロセスの 1 ランクを用いてタスクキューを管理しており，資源管理機構自体の耐障害性およびスケーラビリティが課題となっている．

本稿では，ULFM MPI をベースとした Falanx ミドルウェアの資源管理機構として高可用分散協調スケジューラを実装し，Falanx ミドルウェアの耐障害性およびスケーラビリティを高めることを目指す．我々は，既発表研究 [8], [9] において高可用分散協調スケジューラを提案している．提案するスケジューラでは，耐障害性およびスケーラビリティを高めるために，複数プロセス上に構成されたインメモリファイルシステム上で資源管理情報を冗長管理し，タスクの実行を担う Starter が自律的にタスクキューに投入されたタスクを取得して実行する．Falanx 資源管理機構のプロトタイプとして，Apache ZooKeeper[10]（以降，ZooKeeper と呼ぶ）を用いて Java 言語で開発し，その耐障害性を確認している．本稿においても，ZooKeeper をインメモリファイルシステムとして利用することとし，ZooKeeper の C

[1] 産業技術総合研究所 National Institute of Advanced Industrial Science and Technology (AIST)
[a)] atsuko.takefusa@aist.go.jp

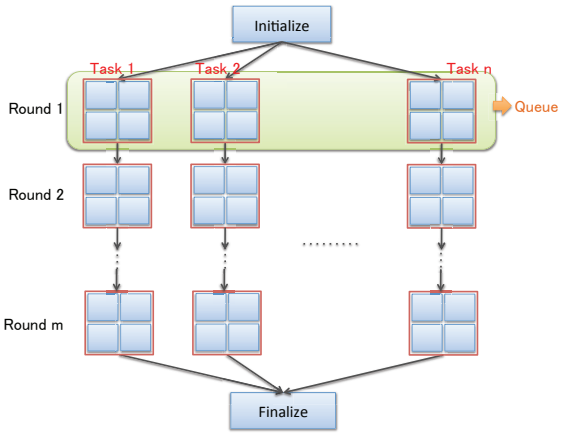


図 1　対象アプリケーションのプログラムロジックイメージ

バインディングを用いて Falanx ミドルウェアのメインロジック側およびワーカ側の処理を実装した．

予備実験では，1 ランクを用いた既存実装と本稿で実装した高可用分散協調スケジューラの実行時間を比較し，ZooKeeper を用いた資源管理情報の冗長管理によるオーバヘッドを調査した．実験から，細粒度のタスクを大量に処理する場合，高可用分散協調スケジューラのオーバヘッドが非常に大きくなってしまい，Starter 間でのタスク取得処理の効率化が必要不可欠であることが示された．

## 2. Falanx の概要

### 2.1 設計方針

Falanx ミドルウェアは，Bag-of-Tasks アプローチによる耐障害性の実現する，単一 MPI ジョブとしてプログラムを実行する，という特徴をもつ．

Falanx ミドルウェアでは，Bag-of-Tasks と呼ばれるアプローチを採用し，図 1 に示すような階層型タスク並列処理アプリケーションに対し，計算ノードの故障時の対処をユーザが容易に指定できるようにすることで耐障害性を実現している．対象アプリケーションは複数の再実行可能なタスクで構成されており，各タスクは数十から数百の並列プロセスからなる．一部のタスクの実行が失敗した場合，Falanx ミドルウェアがアプリケーションプログラマによる指示に従ってそのタスクを再実行または破棄し，プログラムを継続して実行させることができる．

また，Falanx では独自の効率的な通信レイヤを持つスーパーコンピュータ環境におけるプログラムの可搬性を高めるため，アプリケーションプログラムを単一の MPI アプリケーションジョブとして実行する．計算ノード群を小ノード群に分割し，それらに個別の MPI Communicator を設定して各タスクを処理させる．小ノード群に含まれるノードに故障が発見された場合は，実行中のタスクを破棄した後，その小ノード群の再構成を行う．

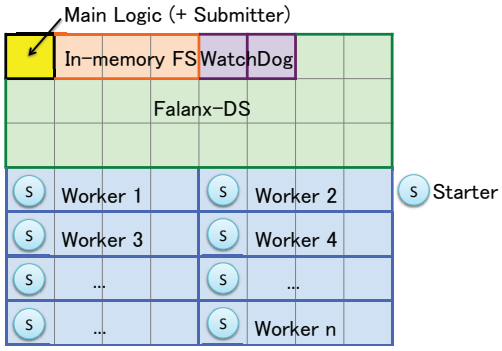


図 2　Falanx の各構成モジュールの計算ノード群への割り当て例

### 2.2 Falanx の構成

Falanx は，計算の中間データを保全するデータストア機構と，計算ノードの健全性を監視しつつ小ノード群にタスクを割り当てる資源管理機構からなる．

データストア機構は，Falanx-DS とよばれる高速な分散オンメモリストレージで構成されている [6]．Falanx-DS は，単純なハッシングにより複数ノード上にデータを分散させるとともに，データの複製を行う．書き込み時には，ハッシュ関数で特定されたプライマリサーバとそれに隣接するセカンダリサーバに 2 つのレプリカを書き込む．読み出し時は，プライマリサーバから読み出し，この読み出しが失敗した場合はセカンダリサーバから読み出す．Falanx-DS を構成するノードが故障した場合は，隣接するノードに失われたデータのレプリカを作成して再構成する．Falanx-DS のプロトタイプシステムではバックエンドに Kyoto Cabinet[11] を用い，永続化機能を無効化してオンメモリストレージとして実装している．

資源管理機構は，Submitter，Starter，In-memory FS，WatchDog からなる高可用分散協調スケジューラで構成される [8], [9]．Submitter は，図 1 に示すようなプログラムにおいて，実行可能なタスクのタスクキューへの投入する．また，アプリケーションプログラマの指示に従って失敗したタスクの再実行または破棄を行う．各 Starter は，小ノード群を管理しており，タスクキューに投入されたタスクをその小ノード群上で実行する．In-memory FS は複数ノードのメモリ上で資源管理情報を高速に冗長管理するデータストアであり，タスクキューやタスクの処理状況，Starter 情報等を保持する．Submitter や Starter は，In-memory FS を介して処理対象のタスクの情報を授受する．また，WatchDog は計算ノードの死活監視を行う．In-memory FS に格納された Starter または Starter が管理する小ノード群に属する計算ノードに障害が発生した場合，その小ノード群が担当していたタスクの状況を失敗状態とし，Submitter が再実行または破棄できるようにする．

図 2 に，高可用分散協調スケジューラを用いた場合における Falanx の各構成モジュールの計算ノード群への割

```
#include <mpi.h>
#include <falanx.h>
#include <datastore.h>

#define N_TASKS        10000
#define N_ROUNDS       2
#define SIZE_CLIENT    1
#define SIZE_DATASTORE 4
#define SIZE_WORKER    20

int main(int argc, char** argv) {
  falanx_context_t* con;
  int n_worker;

  MPI_Init(&argc, &argv);

  //Setup Falanx context and assign a role to each rank
  con = falanx_init_simple(MPI_COMM_WORLD, SIZE_CLIENT,
        SIZE_DATASTORE, SIZE_WORKER, &n_worker);

  // Register task functions
  falanx_register_function(con, "task_A", taskA);

  // Launch Falanx runtime
  if (falanx_start(con) == FLX_SUCCESS) {
    // only the client ranks reach here
    do_main_logic(con);
    falanx_stop(con);
  }

  falanx_finalize(con);
  MPI_Finalize();
  return 0;
}
```

図 3　main プログラム例

り当て例を示す．タスク群をタスクキューに投入するメインロジック（Main Logic）は，MPI プロセスの 1 ランクに割り当てられる．メインロジックは，Submitter としても機能する．In-memory FS および WatchDog は，耐障害性を高めるため複数ランクを割り当てる．WatchDog は本来 1 ランクで動作することができるが，複数ランクに割り当てることで 1 ランク目の WatchDog のノードが故障してしまった場合も 2 ランク目以降の WatchDog が引き続き死活監視処理を行うことができる．Falanx-DS は，Falanx-DS に格納するデータサイズを考慮し，複数ランクに割り当てる．最後に，ワーカ用の小ノード群を割り当て，各小ノード群を代表するランクが Starter の役割を担う．

### 2.3　サンプルコード

図 3，4，5 に Falanx のプログラム例を示す．図 3，4 はメインロジックで実行されるプログラムであり，図 **5** はワーカ用の各小ノード群で実行されるプログラムである．

```
void do_main_logic(falanx_context_t* con) {
  data_store_t* ds = NULL;
  data_in     inp;
  data_out    out;
  falanx_id_t   id;
  char   key[MAX_KEY_LENGTH];
  size_t len;
  int  i, rv, round = 0, priority = 0;

  ds = falanx_context_get_data_store(con);

  // Submit initial tasks
  for (i = 0; i < N_TASKS; i++) {
    prepare_input(&inp, i, round);
    len = sprintf(key, ":in:%04d", i);
    data_store_set(ds, key, len, (char*) &inp,
                   sizeof(inp));
    falanx_task_submit(con, "task_A", key, len,
                       priority, &id);
  }

  // Wait for any tasks to finish
  while (falanx_task_wait_any(con, &id) !=
         FLX_NO_WAIT_TASK) {
    rv = falanx_get_result_value(
           con, id, key, MAX_KEY_LENGTH, &len);
    if (rv == FLX_ERR_PROC_FAILED) {
      // Re-submit a failed task here.
      falanx_task_submit(con, "task_A", key, len,
                         priority + 1, &id);
      continue;
    }
    data_store_get_value(ds, key, len, (char*) &out,
                         sizeof(out));
    round = prepare_next_input(&inp, &out);

    if (round < N_ROUNDS) {
      // submit the subsequent task
      len = sprintf(key, ":in:%04d", inp.task_id);
      data_store_set(ds, key, len, (char*) &inp,
                     sizeof(inp));
      falanx_task_submit(con, "task_A", key, len,
                         priority, &id);
    }
  }
}
```

図 4　メインロジックプログラム例

図 **3** は，アプリケーションプログラムの main 関数であり，図 1 のにおいてタスク数 10000，ラウンド数 2 の例を表している．Falanx は 1 つの MPI プログラムとして実行されるため，まず最初に `MPI_Init()` で MPI の初期化を行う．次に，`falanx_init_simple()` において Falanx の初期化と図 2 のような各構成モジュールの MPI ランクへの割り当

```
#include <mpi.h>
#include <falanx_worker.h>
#include <datastore.h>

void taskA(falanx_context_t* con, MPI_Comm comm){
  int       rank;
  data_store_t* ds;
  char      key[MAX_KEY_LENGTH];
  size_t    len;
  data_in     inp;
  data_out    out;

  MPI_Comm_rank(comm, &rank);
  ds = falanx_context_get_data_store(con);

  if (rank == 0) {
    // get input data from the data store
    falanx_get_request_key_value(
      con, key, MAX_KEY_LENGTH, &len);
    data_store_get_value(ds, key, len, (char*) &inp,
                         sizeof(data_in));
  }

  do_task_A(&inp, &out, comm);

  if (rank == 0) {
    // put result to the data store
    len = sprintf(key, ":out:%04d", inp.task_id);
    data_store_set(ds, key, len, (char*) &out,
                   sizeof(data_out));
    falanx_set_response_key(con, key, len);
  }
}
```

図 5　サブルーチンプログラム例

てを行う．その後，falanx_register_function() でワーカで実行するタスクの関数 taskA（図 5 で定義）を登録する．falanx_start() で Falanx ルーチンの実行に成功後，図 4 に示したメインロジックを実行する．すべての処理が完了すると，falanx_stop() および falanx_finalize() を実行して，処理を終える．

図 4 では，メインロジックを定義している．falanx_context_get_data_store() で Falanx-DS へのポインタを取得した後，各タスクの実行に必要な情報を data_store_set() で予め格納しておく．その後，falanx_task_submit() でタスクを実行する関数名，タスク実行に必要な情報を取得するためのキー情報を渡してタスクキューにタスクを投入し，falanx_task_wait_any() でいずれかのタスクの実行が終了するまで待つ．タスクが終了すると，falanx_get_result_value() でタスクの正常終了とタスクの実行結果を取得するためのキー情報を資源管理機構から取得し，data_store_get_value() で Falanx-DS から実行結果を取得する．falanx_get_result_value() の戻り値が FLX_ERR_PROC_FAILED の場合，何らかの原因によりそのタスクの実行が失敗したことを意味する．その場合，タスクの再実行または破棄を行うなど，失敗時の処理をアプリケーションプログラマが適宜記述する．また，ラウンド数が 2 以上の場合は，再度タスクに必要な情報の Falanx-DS への格納・タスクのタスクキューへの投入を繰り返し，全てのタスクを実行する．

図 5 はワーカ用のサブルーチンプログラムである．ワーカでは，まずプログラムを実行する MPI Communicator のランクを MPI_Comm_rank() で取得する．ランク 0 のプロセスは Starter の処理を担い，falanx_get_request_key_value() で資源管理機構から計算に必要な情報を取得するためのキー情報を取得し，そのキーを用いて data_store_get_value() で Falanx-DS からタスク情報を得る．その後，ワーカを構成する小ノード群で割り当てられたタスクを実行する．図 5 では，do_task_A() でタスクの処理を行っている．タスクの実行が終了すると，ランク 0 のプロセスが data_store_set() で結果を Falanx-DS に格納し，falanx_set_response_key() でその情報を取得するためのキー情報を資源管理機構に格納する．以上のようにして，Falanx では資源管理機構および Falanx-DS を介して耐障害性のある階層型タスク並列プログラムを記述することができる．

## 3. 高可用分散協調スケジューラの Falanx への実装

### 3.1 資源管理機構の既存実装

既発表研究 [8], [9] では，資源管理機構として高可用分散協調スケジューラを提案し，In-memory FS には分散する複数ノードのメモリ上に木構造のデータストアを構築してデータの永続化および冗長管理を行う ZooKeeper を採用した．また，ZooKeeper の提供する分散協調サービスライブラリを用いて Submitter，Starter，WatchDog を Java で実装した．

一方，既存の Falanx ミドルウェアプロトタイプ実装では，MPI プロセスの 1 ランクを用いて In-memory FS，WatchDog に相当する処理とタスク割り当て処理を行っている．この処理を担うプロセスを，“タスクキューサーバ”と呼ぶ．資源管理情報はこの 1 ランクで管理しているのみで，Java 実装における In-memory FS での冗長管理・永続化機能は提供されていない．また，Submitter 処理はメインロジックプロセスで，Starter に相当する処理は各ワーカの小ノード群のランク 0 プロセスで行っており，タスクキューサーバと通信してタスクキューへのアクセスや Starter 情報の登録・取得を行う．

既存実装では，1 ランクで資源管理処理を行っているため，タスク数が増大したときの資源管理機構のスケーラビ

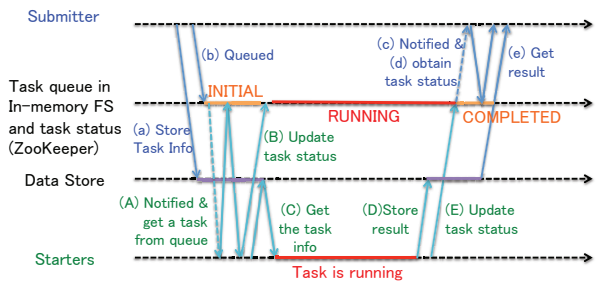


図 6　高可用分散協調スケジューラの処理の流れ

リティや，資源管理プロセスの計算ノードに故障があった場合の資源管理情報の損失などの耐障害性に課題が残る．本稿では，Falanx ミドルウェアのスケーラビリティおよび耐障害性を高めるため，高可用分散協調スケジューラを Falanx へ実装する．なお，障害を検知して，障害ノードが担当していたタスクが失敗したことを知らせる WatchDog 機能は今回は未実装である．

### 3.2　実装

本稿では，高可用分散協調スケジューラの Falanx への実装の第一段階として，Java 実装同様に In-memory FS に ZooKeeper を用いる．また，Submitter，Starter 機能は ZooKeeper の C バインディングを用いて実装した．

図 **6** に，高可用分散協調スケジューラの処理の流れを示す．上から，Submitter，In-memory FS，Falanx-DS，および Starter 群の処理シーケンスを表し，青字の (a)–(e) は Submitter の処理，緑字の (A)–(E) は Starter の処理を示している．表 **1** に示すように，各処理は 2.3 節 で述べた主な Falanx 関数に対応している．

実装する高可用分散協調スケジューラでは，メインロジックで実行される Submitter ライブラリとして (b)，(c)，(d) を，ワーカ側で実行される Starter ライブラリとして (A)，(B)，(E) を，ZooKeeper 分散協調ライブラリの C バインディングを用いて実装した．図 **7** に，ZooKeeper が提供する In-memory FS での資源管理情報のデータ構造を示す．/zoo 以下の task_*にタスクキュー情報を，starterlist に Starter のホスト情報を管理する．Submitter ライブラリおよび Starter ライブラリの処理と In-memory FS 情報との対応について以下に説明する．

#### 3.2.1　Submitter ライブラリの実装

(b) のタスクキューへのタスク投入では，task_*以下にタスク情報を格納していく．この際，タスク ID に相当するノード名は，task_< 優先度 >_< 投入順 > というフォーマットでアプリケーションプログラマがつける．このノード名を元に，Starter はどのタスクから処理するか，判断する．タスク情報として，ステータス（status），タスクを実行するための情報を取得するキー情報（inputkey），実行する関数名（func）を格納する．タスクのステータスは，初期状態では INITIAL，Starter がタスクを取得した時点で RUNNING，タスクの実行が終了した時点で COMPLETED，何らかの障害が発生して正常に終了しなかった場合には FAILED となる．

また，task_*以下の initial ノードにもタスク ID を格納する．これにより，Starter は initial ノード以下のタスクを実行可能なタスクとして取得することができる．すべてのタスクの投入が終わると，task_*/done を監視するウォッチャーを設定する．ウォッチャーは ZooKeeper が提供する機能であり，設定したノードに変更があった場合は，呼び出し側のコールバック関数が呼ばれる仕組みとなっている．これにより，Submitter と In-memory FS 間の無駄な通信を削減することができる．

(c) では，ZooKeeper のウォッチャーからの通知を受け，あるタスクが終了したことをアプリケーションプログラムに通知する．また (d) では，done ノードに格納されているタスクのステータスを確認し．正常終了時には出力結果のキー情報をアプリケーションプログラムに返す．WatchDog により失敗タスクが検出された場合も，タスク ID が done ノード内に格納されるため，この段階でタスク実行の失敗を通知することができる．(d) の結果を受けて，アプリケーションプログラマがその後の処理を行う．

#### 3.2.2　Starter ライブラリの実装

Starter は，task_*/initial ノードに予めウォッチャーを設定しておき，タスクキューにタスクが投入されるまで処理待ち状態とする．Submitter 同様，ウォッチャー機能を用いて Starter と In-memory FS 間の無駄な通信を行わないようにする．タスクキューにタスクが投入されると，(A) で示したように In-memory FS から通知される．通知があると，(B) においてまず initial ノード内の全タスク ID を取得し，そのうち最初に処理すべきタスクを特定する．次に，特定したタスク ID の task_*/< タスク ID> ノードにアクセスし，タスクのステータスが INITIAL であることを確認した後，タスク ID 名を initial ノードから削除して running ノードへ書き込む．ZooKeeper がタスクキューに相当する機能を有していないため，自律的に動作する Starter 群が順次別々のタスクを実行するために，このような処理を行う．競合を緩和するため，initial ノード以下に bucket と呼ぶ階層を作り，bucket 間の順序は保証しつつ，bucket 内のアクセスはランダムに行う手法も提案している [9]．running ノードへの書き込みが無事成功すると，取得したタスクを実行する．

上記のいずれかの処理で失敗した場合，すなわち他の Starter がタスクキューの先頭タスクを取得した場合は，失敗した Starter は initial ノードにタスクが残っている間，上記の処理を繰り返す．タスクが空になったら，再度ウォッチャーを設定して次のタスクが投入されるまで待つ．

(E) では，(D) でタスクの実行結果を DS に格納した際に

表 1　Falanx の主な関数

| | 関数名 | 意味 |
|---|---|---|
| | （Submitter 関数） | |
| (a) | `data_store_set()` | DS へのタスク情報の格納 |
| (b) | `falanx_task_submit()` | タスクキューへのタスクの投入 |
| (c) | `falanx_task_wait_any()` | タスク実行終了通知 |
| (d) | `falanx_get_result_value()` | タスクステータスの取得 |
| (e) | `data_store_get_value()` | DS からのタスク実行結果の取得 |
| | （Starter 関数） | |
| (A)，(B) | `falanx_get_request_key_value()` | タスクのキー情報を取得およびタスクステータスの更新 |
| (C) | `data_store_get_value()` | DS からのタスク情報の取得 |
| (D) | `data_store_set()` | DS へのタスク情報の格納 |
| (E) | `falanx_set_response_key()` | DS のキー情報の格納およびタスクステータスの更新 |

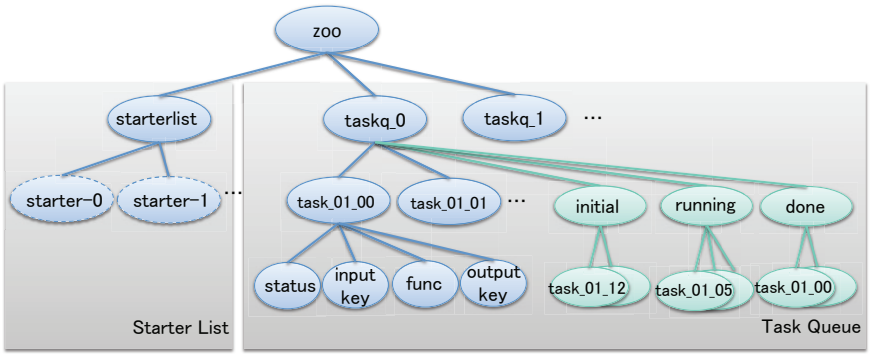


図 7　In-memory FS における znode 構成の概要

利用した出力キー情報を task_*/< タスク ID>/outputkey に格納する．その後，タスク ID 名を running ノードから削除し，done ノードへ書き込む．

## 4. 予備実験

高可用分散協調スケジューラでは，複数プロセスにより資源管理情報を分散冗長管理するとともに，各 Starter のタスク取得処理は独立して行われるため，1 ランクで実装されたタスクキューサーバより実行オーバヘッドが大きいと考えられる．よって，予備実験では既存実装のタスクキューサーバ（1 ランク版）と比較することにより，Falanx の高可用分散協調スケジューラ実装（高可用版）の実行オーバヘッドを調査する．

### 4.1 実験環境

実験は，AIST Super Cloud を用い，仮想環境で行った．OS はホスト，ゲストとも CentOS release 6.5（Final）を用いた．ホストマシンの CPU は Intel Xeon E5620（2.4 GHz，4 コア，2 ソケット），メモリは 20G バイトであり，実験では 4 台 32 コアを用いた．各ホスト間は 10Gbit Ether スイッチで接続されている．また，仮想化には KVM（quem-kvm-0.12.1.2-2.355.el6_4.6）を用い，ゲストマシンの CPU は QEMU Virtual CPU version (cpu64-rhel6)，メモリは 20Gb バイト，ネットワークインタフェースには virtio-net を用いた．

表 2　1 ランク版と高可用版の処理時間の比較

| | Setup | Main Logic | Baseline | Total [sec] |
|---|---|---|---|---|
| 1 ランク版/1 秒 | 0.806 | 2.853 | 500 | 503.658 |
| 高可用版/1 秒 | 1.182 | 157.253 | 500 | 658.435 |
| 1 ランク版/2 秒 | 0.807 | 2.611 | 1000 | 1003.417 |
| 高可用版/2 秒 | 0.858 | 77.080 | 1000 | 1077.938 |

全 32 コアに対して，1 ランク版ではメインロジック ×1，タスクキューサーバ ×1，Falanx-DS ノード ×4，ワーカ ×20 を 1 コアずつラウンドロビンで割り当てた．また，高可用版ではメインロジック ×1，Falanx-DS ノード ×4，ワーカ ×20，ZooKeeper プロセス ×3 を同様にラウンドロビンで 1 コアずつ割り当てた．ZooKeeper は v. 3.4.6 を用いた．

アプリケーションプログラムとしては，ZooKeeper のオーバヘッドを調査を目的とし，1 秒または 2 秒スリープするタスクを 10000 個実行するものを用いた．20 個のワーカで処理する場合，理想的な処理時間はそれぞれ 500 秒，1000 秒となる．

### 4.2 実験結果

表 2 に 1 ランク版と高可用版の処理時間の結果を示す．各版に対し，/（スラッシュ）以降に書かれた秒数はタスクのスリープ時間を秒で表す．また，Setup，Main Logic，Baseline，Total は，それぞれ `MPI_Init()` 等のメインロ

ジック以外に要した時間，メインロジックからタスク実行時間の理想値を差し引いた時間，タスク実行時間の理想値，総処理時間を表す．Main Logic 部分が資源管理処理のオーバヘッドに相当する．

1 ランク版と高可用版のオーバヘッドを比較すると，1 ランク版は資源管理処理を 3 秒以内に行っているのに対し，高可用版ではスリープ時間が 1 秒と小さくなると高可用版のオーバヘッドが非常に大きくなることが分かる．また，高可用版ではスリープ時間が 2 秒の場合でも 74.5 秒近く資源管理処理に要しており，今回の実験のようにタスクの粒度が非常に小さい状況では，ZooKeeper によるオーバヘッドが非常に大きくなってしまうことが明らかとなった．よって，Starter 群がタスクキューからタスクを取得する際，既発表研究 [9] で提案した bucket を用いる手法や，Starter 同士で処理順を制御する手法などを適用する必要があることが示された．

## 5. 関連研究

耐障害性を考慮したタスク並列アプリケーション向けのスケジューラの研究としては，以下のものがある．

Condor Master Worker (CondorMW)[12], [13] は，マスターワーカ型アプリケーションの記述を助けるフレームワークであり，Condor バッチスケジューラを用いてワーカプログラムを複数の計算機に自動的に割り当てる．チェックポインティングによるデータ保全は行うものの，タスクの再実行や削除による継続実行は想定されていない．また，CondorMW では個々のタスクをバッチスケジューラに投入する点で Falanx とは異なる．

辻らは，並列プログラミング言語 XcarableMP で記述された並列タスクを対象とした階層型プログラミングツール FP2C を開発している [14]．FP2C では，上位のタスク並列プログラムをワークフローとして記述し，OmniRPC を用いて各タスクの並列実行を行う．一方，Falanx は階層型タスク並列プログラムを 1 つの MPI プログラムとして実行する．

梅田らは，グリッド環境において耐故障性と資源数の増加に対しスケーラブルな分散ジョブスケジューリングシステムを提案している [15]．スケジューリングに必要な資源収集や実行ジョブと資源のマッチングを少数の実機で行うと，単一障害点の存在とスケーラビリティの欠如という問題が起こる．よって，マッチングを行うスケジューリングノード間でゴシッププロトコルを用いて資源情報を共有し，スケジューリングノードを複数分散させてこれらの問題を解決する．この分散ジョブスケジューリングシステムでは，ノード間が疎結合な環境で複数ジョブの処理スループットの向上を目的としているのに対し，我々の研究ではエクサスケール計算機環境で 1 つの階層型タスク並列アプリケーションプログラムの実行を高速かつ継続して実行させることを目的としている点で異なる．

## 6. まとめと今後の課題

我々は，エクサスケール計算機環境において耐障害性を有した階層型タスク並列アプリケーションプログラムの開発を支援する Falanx ミドルウェアを開発している．Falanx は，データストア機構と資源管理機構からなり，アプリケーションプログラムを単一 MPI ジョブとして実行させ，実行障害発生時には選択的にタスク単位で再実行／破棄を行うことができる．既存実装では，資源管理機構は 1 ランクにより処理していたため，本研究では Falanx システムの耐障害性およびスケーラビリティを高めることを目指して，高可用分散協調スケジューラを ZooKeeper を用いて実装した．予備実験を行った結果，細粒度のタスクを大量に処理する場合には高可用分散協調スケジューラのオーバヘッドが非常に大きくなってしまい，Starter 間でのタスク取得処理の効率化が必要不可欠であることが示された．

今後は，タスク取得処理の効率化を行うとともに，Watch-Dog 機能の実装，エクサスケール計算環境への移植を行う．

謝辞　本研究の一部は，JSPS 科研費 25871199 の助成を受けたものである．

### 参考文献

[1] Jack Dongarra et al.: International EXASCALE SOFTWARE PROJECT ROADMAP 1.1, http://www.exascale.org/mediawiki/images/a/a8/IESP-roadmap-1.1.pdf.
[2] 石川裕，丸山直也ほか：HPCI 技術ロードマップ白書，http://open-supercomputer.org/wp-content/uploads/2012/03/hpci-roadmap.pdf.
[3] FOX Project (A Fault-oblivious Extreme-scale Execution Environment): http://fox.xstack.org/.
[4] 池上　努，田中良夫，中田秀基，高野了成，関口智嗣：ポストペタスケール高性能計算に向けた階層的プログラミングモデルの提案，情報処理学会研究報告 2012-HPC-133(10), pp. 1–6 (2012).
[5] The Falanx project: https://sites.google.com/site/spfalanx/.
[6] 中田秀基，池上　努，竹房あつ子，高野了成，田中良夫：ポストペタスケール高性能計算のためのオンメモリストレージの設計，情報処理学会研究報告 2013-HPC-140(28), pp. 1–7 (2013).
[7] Bland, W., Bosilca, G., Bouteiller, A., Herault, T. and Dongarra, J.: A proposal for User-Level Failure Mitigation in the MPI-3 Standard, Technical report, Department of Electrical Engineering and Computer Science, University of Tennessee (2012).
[8] 竹房あつ子，中田秀基，池上　努，田中良夫：ポストペタスケール計算機環境に向け高可用分散協調セルフスケジューリング機構の提案，情報処理学会研究報告 2012-HPC-133(26)，pp. 1–6 (2012).
[9] 竹房あつ子，中田秀基，池上　努，田中良夫：パーシステントストレージを利用した高可用分散協調スケジューラの実装，情報処理学会研究報告 2013-HPC-140(20)，pp. 1–6 (2013).
[10] Apache ZooKeeper: http://zookeeper.apache.org/.

[11] FAL Labs: Kyoto Cabinet: a straightforward implementation of DBM, http://fallabs.com/kyotocabinet/.
[12] The MW Homepage: http://research.cs.wisc.edu/condor/mw/.
[13] Goux, J.-P., Kulkarni, S., Linderoth, J. and Yorde, M.: An Enabling Framework for Master-Worker Applications on the Computational Grid, *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, Pittsuburgh, Pennsylvania, pp. 43–50 (2000).
[14] 辻美和子，佐藤三久，Hugues, M.，Petiton, S.：並列コンポーネントを統合する階層的並列プログラミングモデル，情報処理学会研究報告 2012-HPC-135，pp. 1–10 (2012).
[15] 梅田典宏，中田秀基，　松岡聡：大規模環境向け情報共有手法を用いた分散ジョブスケジューリングシステム，情報処理学会研究報告 2006-HPC-105，pp. 223–228 (2006).