Vol.2013-HPC-140 No.20
2013/8/1

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

竹房 あつ子[1]　中田 秀基[1]　池上 努[1]　田中 良夫[1]

概要：階層型タスク並列処理は，タスクの再実行や冗長実行により耐障害性を備えたプログラムが設計できるため，ポストペタスケール高性能計算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．我々は，耐障害性を備えたアプリケーションプログラムの開発を支援にする耐障害アプリケーションフレームワーク Falanx を提案している．このようなアプリケーションフレームワークは，計算に必要となるデータを障害から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適切に計算を実行する資源管理機構からなる．これらを，ポストペタスケール計算機環境においてスケーラブルでかつ，それら自身が耐故障性を持つように設計・実装する必要がある．本研究では，耐障害アプリケーションフレームワークのポストペタスケール計算機環境での性能特性を検証して技術的課題を明らかにすることを目的とし，試験実装となる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を設計・開発する．本スケジューラは既に実装を進めている資源管理機構と新たに追加したデータストア機構で構成され，Apache ZooKeeper と Apache Cassandra を用いて実装することで耐障害性を実現する．本スケジューラを用いた予備実験から，処理中に計算ノードが落ちてしまった場合も，自動的にタスクが再実行されアプリケーションプログラムが継続実行できることを確認した．

キーワード：高可用性，分散協調スケジューラ，耐障害性，ポストペタスケール高性能計算，パーシステントストレージ

1. はじめに

2020 年前後に実現する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 [1], [2] ため，その故障発生間隔 (MTBF) は 1 日から 5 分になると考えられている．システムの一部が故障した状況が常態化するため，ポストペタスケール計算機環境で長時間にわたって実行されるアプリケーションには，故障が発生しても計算を継続実行することができる耐障害性 (フォルトレジリエンス) が求められる．

タスク並列の各タスクにデータ並列を内包させた階層型タスク並列処理では，比較的容易に耐障害性を持たせるように設計することができるため，ポストペタ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]．具体的には，フラグメント分子軌道法 (FMO)[5] のように障害により失敗した部分について局所的に再計算して実行フローを維持する方法，あるいはブロック櫻井・杉浦法 [6] のように部分的に失敗した部分があっても処理結果に影響を与えないようにアルゴリズムそのものに冗長性を内包させる方法がある．しかしながら，これらを実際に耐障害性を持つように実装するには，障害検知のためにシステムソフトウェアとの連携した処理を行うなど，計算ロジックと無関係な部分で煩雑なコーディングをしなければならない．よって，ポストペタスケール計算機環境で耐障害性を備えつつスケーラブルな階層タスク並列型アプリケーションの開発を容易にする，耐障害アプリケーションフレームワーク Falanx を提案し，開発を進めている [4], [7].

このようなアプリケーションフレームワークでは，アプリケーションの実行を継続させるための仕組みとして，計算に必要となるデータを障害から保全するためのデータストア機構と，計算ノードの健全性を監視しつつ適切な計算ノード上で計算を実行する資源管理機構が必要となる．これらが，ポストペタスケール計算機環境においてスケーラブルでかつ，それら自身が耐故障性を持つように実装する必要があるが，その設計指針が明らかでない．

本研究では，耐障害アプリケーションフレームワークのポストペタスケール計算機環境での性能特性を検証して技

[1] 産業技術総合研究所 National Institute of Advanced Industrial Science and Technology (AIST)
1–1–1 Umezono, Tsukuba, Ibaraki 305–8568, Japan

術的課題を明らかにすることも目的とし，試験実装となる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を設計・開発する．既発表研究 [8] では，資源管理機構を設計し，Java で実装を進めている．資源管理機構では，タスクの計算ノードへの割り当て，タスクの処理状態の管理，計算ノードの死活監視と，故障時のタスクの再実行処理を行う．複数管理プロセスを分散協調させて，資源管理機構そのものの耐故障性，資源管理情報の永続化，スケーラビリティのため，Apache ZooKeeper[9]（以降，ZooKeeper とする）を用いて実装した．

本稿では，資源管理機構の改良を進めると共に，パーシステントストレージを用いたデータストア機構を設計し，高可用分散協調スケジューラを実装する．パーシステントストレージを構成するデータノードはタスクを実行する各計算ノード上に配備し，メインプロセスからタスクプロセスに送信する計算に必要な引数情報，およびタスクプロセスからメインプロセスに返す計算結果情報を格納する．これにより，計算途中の情報がパーシステントストレージに格納されるため，タスク実行時に計算ノードが故障した場合やメインプロセスを実行する計算ノードが故障した場合に，再実行を行うことができる．パーシステントストレージには，データの一貫性を保証可能な分散 KVS（Key Value Store）実装の 1 つである Apache Cassandra[10]（以降，Cassandra とする）を用いた．

予備実験では，実装したスケジューラの耐障害性を調査した．実験結果から，処理中に計算ノードが落ちてしまった場合も，そのノードが担当していた処理を再実行することでアプリケーションプログラムが継続して実行できることを確認した．

2. 耐障害アプリケーションフレームワークの概要

耐障害性を持つアプリケーションの実装を支援するアプリケーションフレームワークが対象とするアプリケーションプログラムについて述べるとともに，耐障害アプリケーションフレームワークの概要について説明する．

2.1 対象アプリケーションプログラム

耐障害アプリケーションフレームワークでは，ポストペタスケール計算機環境で有望なプログラミングモデルの 1 つである階層型タスク並列プログラムを対象とする．図 1 に，対象とするアプリケーションのプログラムロジックイメージを示す．図中の Task は，資源管理機構が資源割り当てを行う単位であるタスクであり，個々のタスクは数十から数百コア並列で処理することを想定している．アプリケーションプログラムの処理 Process 1 が終了すると，4 つのタスクが実行可能となるため，タスクキュー Queue にそれらのタスクが投入される．資源管理機構は，タスク

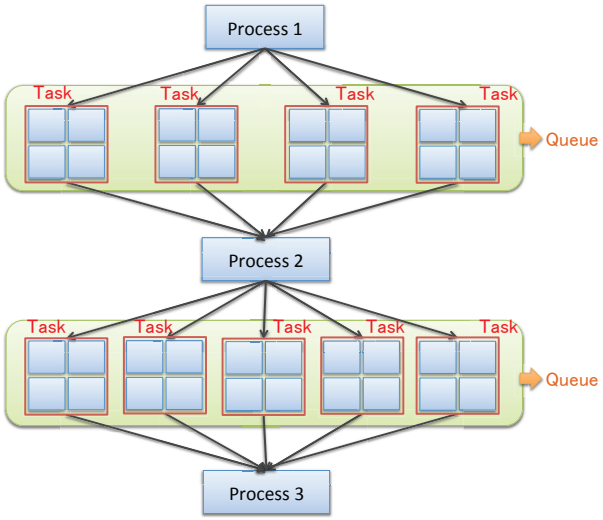

図 1　対象アプリケーションのプログラムロジックイメージ

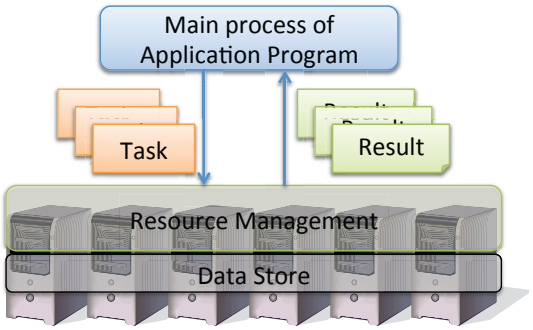

図 2　耐障害アプリケーションフレームワークの概要

キューに投入されたタスクを順次計算ノードに割り当てていく．4 つのタスクの実行がすべて終了すると，Process 2 の実行を開始し，Process 2 の実行が終了すると，同様に 5 つのタスクがタスクキューに投入される．

対象とするアプリケーションのプログラマは，計算ノード等の故障によりタスクの実行が失敗した場合，タスク単位で再実行，または結果の削除を行うことを，明示的に指定することができる．これにより，局所的に再計算するアルゴリズムや冗長性を内包させたアルゴリズムの実装を可能にする．

2.2 耐障害アプリケーションフレームワーク

図 2 に耐障害アプリケーションフレームワークの概要を示す．耐障害アプリケーションフレームワークは，資源管理機構（Resource Management）とデータストア機構（Data Store）からなる．資源管理機構は，アプリケーションプログラムの各タスクを Queue で管理し，利用可能な計算ノードで各タスクを実行する．また，タスクを実行する計算ノードの健全性を監視しておき，計算ノードに故障が検知された場合はその計算ノードが担当していたタスクの再実行または削除を行う．データストア機構では，アプリケーションプログラムのメインプロセスを含む計算ノード

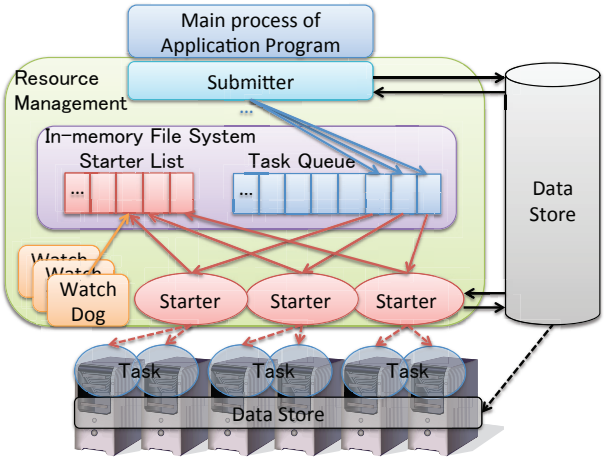

図 3　高可用分散協調スケジューラのアーキテクチャ

間で授受されるデータを格納・複製する．複数計算ノードで複製を共有することでデータを保全するとともに，一部の計算ノードに障害が発生した場合にそのノードが担当していた計算の途中結果を利用した再実行や適切な削除処理が行えるようにする．

3. 高可用分散協調スケジューラ

耐障害アプリケーションフレームワークのポストペタスケール計算機環境での性能特性を検証するため，その試験実装となる高可用分散協調スケジューラを開発する．まず，既発表研究 [8] で設計した資源管理機構の概要と本稿での改良部分について述べた後，今回実装したデータストア機構の設計，実装について述べる．

3.1 資源管理機構の概要とデータストア機構との連携

資源管理機構では，スケーラブルかつ耐障害性を備えるため，以下の機能が必要となる．

(1) タスクキューへのタスク投入
(2) タスクキュー内のタスクの実行
(3) 計算ノードの死活監視
(4) 障害発生時のタスク再実行／削除
(5) 資源管理情報のスケーラブルな管理と永続化

これらを実現するため，資源管理機構のシステムアーキテクチャを図 3 のように設計した．資源管理機構は，Submitter，Starter，WatchDog と，タスクキューや Starter リスト等の資源管理情報を格納するインメモリファイルシステム In-memory FS からなる．これらとデータストア機構が連係動作することで，高可用分散協調スケジューラのスケーラビリティ，可用性を実現する．資源管理機構の各モジュールの機能について，以下で説明する．

3.1.1 Submitter

Submitter は (1) タスクキューへのタスク投入機能と，(4) 障害発生時のタスク再実行／削除の機能を持つ．アプリケーションプログラムで指定されたタスクの実行に必要な情報をデータストア機構に格納した後，タスクを In-memory FS で管理されているタスクキューに投入する．タスクキューではタスクのステータス情報が管理されており，Submitter は定期的に投入したタスクの実行状況を確認する．タスクの実行が終了したら，アプリケーションフレームワークのデータストア機構から結果を取得する．タスクの実行が失敗した場合は，再実行が必要なものは新たなタスクとしてタスクキューに投入し，不要なものはそのタスクの削除処理を行う．投入したタスクの情報はデータストアに格納されているため，Submitter やアプリケーションのメインプロセスを実行している計算ノードが故障した場合も，計算の途中から実行するできる．

3.1.2 Starter

Starter は (2) タスクキュー内のタスクの実行を行う．Starter は各計算ノード上の実行デーモンプロセスであり，計算ノードごとに配備される．各 Starter は自律的にタスクキューの先頭のタスクを取り出し，管理する計算ノードでタスクを実行する．Starter はタスクを実行し終えると，実行結果をデータストア機構に格納する．

Starter はタスクを取得する前に自身の情報を Starter リストに登録しておき，WatchDog で計算ノードの死活監視ができるようにしておく．また，タスクキューではタスクのステータス情報も管理しておく．ステータス情報は，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．これにより，Submitter がタスクの実行終了や失敗を知ることができる．

Starter は割り当てられたタスクが終了した場合，または故障が発生して再起動した場合は，今までのタスク情報をすべて削除してから新たなタスクを取得，実行する．また，Starter は定期的に Starter リストの状況を確認し，WatchDog が自身のことを故障していると検知していないかどうか調べる．これは，例えば，一時的に Starter と WatchDog との間のネットワークコネクティビティがなくなった場合に，双方の Starter の状態の不整合が生じる．よって，WatchDog が Starter を故障だと判断していた場合は，Starter が今までのタスクの情報をすべて削除して新たなタスクを取得するようにする．

3.1.3 WatchDog

WatchDog は，(3) 計算ノードの死活監視を行う．WatchDog は定期的にタスクキューと Starter リストの状況を監視し，タスクが割り当てられて `RUNNING` 状態になっているにもかかわらず故障が発生している Starter がいるかどうかを調べる．もし，タスクが `RUNNING` でかつ Starter の故障が検知された場合は，タスクの状態を `FAILED` とする．これにより，Submitter がタスクの再実行または削除を行うことができる．**??**節で述べたように，Starter の故障の

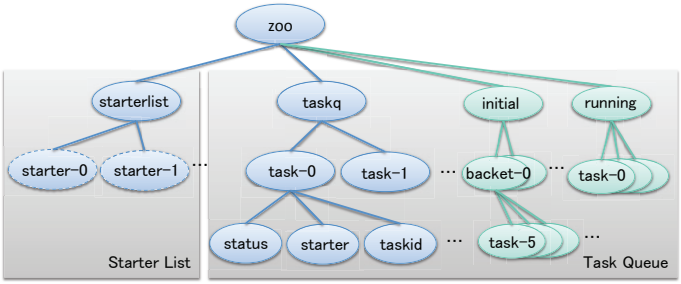

図 4　znode 構成の概要

検知は登録されている Starter に対してネットワークコネクティビティがあるかどうかで判断する．WatchDog の耐障害性を高めるため，予め複数の WatchDog を起動しておき，その中のリーダーが死活監視処理を行うようにする．

3.1.4　In-memory FS

In-memory FS では，タスクキューや Starter リストを含む (5) 資源管理情報のスケーラブルな管理と永続化を実現する．In-memory FS のプロセスは計算機のラック単位で複数立ち上げ，それらを分散協調させて 1 つの簡易なファイルシステムを構成させる．プロセス間でデータの複製を共有しながら，各プロセスでデータの永続化を行う．あるプロセスに故障が発生した場合も，再起動して再度 In-memory FS のプロセスグループに参加することで，再構成が可能になる．In-memory FS では，データ共有の負荷を軽減するため，タスクの ID や Starter のホストアドレスなど，小さいサイズのデータのみを扱う．大きなデータを扱う場合は，In-memory FS ではキー情報のみを扱い，データそのものはアプリケーションフレームワークのデータストア機構に格納して，連係動作させる．

3.2　資源管理機構の実装と改良

既発表研究 [8] において，ZooKeeper を用いて資源管理機構を実装した．ZooKeeper は，分散協調サービスを提供するものであり，分散した複数プロセスがアンサンブルと呼ばれるグループを作り，複製モードで動作させることが可能であるため，単一障害点が回避できる．また，分散するプロセス間のやり取りを支援する機能を持ち，これらの機能を Java や C のライブラリとして提供する．Paxos[11] をベースとした Zab[12] とよばれるアトミックブロードキャストプロトコルを用いて，複数プロセス間でのメッセージ順序を保証する．アンサンブル中にリーダーと呼ばれる特別なメンバを選出し，全ての書き込み要求をリーダーに転送してから行うことで，合意形成するためのプロトコルのアトミック性を保証する．

In-memory FS と計算ノードの死活監視に ZooKeeper を用いている．In-memory FS では，ZooKeeper の提供する階層的な名前空間を利用してタスクキューおよび Starter リストを管理する．図 4 にその詳細を示す．図中の楕円は znode と呼ばれるノードを表す．全ての資源管理情報は，/zoo 以下に格納することにする．右側がタスクキュー，左側が Starter リストを表す．タスクや Starter の投入順の管理と情報の永続化は ZooKeeper が行う．

タスクキューの情報は，/zoo/taskq の他に，/zoo/initial および/zoo/running で表すように改良した．Submitter がタスクを投入する際，`/zoo/taskq/task-*`と`/zoo/initial/backet-*/task-*`のノードを作成する．本来ならば，各 Starter は/zoo/initial 以下に最初に投入されたタスクから順次実行すべきであるが，ZooKeeper では/zoo/initial 以下のノードを待ち行列として管理することができない．すなわち，各 Starter が/zoo/initial の子ノードすべてを取ってきた後，その中で名前に付与された番号が最小のものを実行するという処理を行わなければならない．この場合，全部の Starter が同じタスクを実行しようとして失敗してしまい，結果としてタスクの実行開始が遅くなることが考えられる．よって，/zoo/initial/backet-*というノードを新たに用意してタスクを backet サイズごとに格納する．各 Starter は先頭の `backet` の中からランダムにタスクを選択するように改良した．タスクが Starter に割り当てられると，`/zoo/taskq/task-*`以下の情報が更新されるとともに，/zoo/initial/backet-*/task-*を削除して/zoo/running/task-*を生成する．

死活監視では，Starter が起動時に/zoo/starterlist 以下に `EPHEMERAL` (一時) ノードとして格納した Starter リスト情報を利用する．図 4 では，`EPHEMERAL` ノードは破線の楕円で表している．Starter で障害が発生すると，`EPHEMERAL` ノードとして格納された Starter の znode は ZooKeeper により自動的に削除される．これにより，WatchDog が Starter の障害を検知することができる．また，複数 WatchDog におけるリーダ選出にも ZooKeeper を用いている．

3.3　データストア機構の実装

高可用分散協調スケジューラのデータストア機構では，スケーラブルかつ耐障害性を備えるため，以下の機能を持つパーシステントストレージが必要となる．

(1) タスクの実行に必要な情報および実行結果の格納
(2) 格納された情報の冗長管理
(3) 障害発生時の再構成
(4) データの整合性の保証

本研究では，これらの機能を有する Cassandra をデータストア機構に用いる．Cassandra は分散 KVS 実装の 1 つであり，格納されたデータの複製を生成して複数データノード間で分散協調して冗長管理する．データノードに故障が発生した場合も，残ったデータノード間で値の配置を自動的に再構成することができる．また，読み出し時のレプリカ数を R，書き込み時のレプリカ数を W，レプリケーションファクターを N とすると，$R+W>N$ とな

図 5　Intrigger を用いた実験環境

るようにすれば最新の書き込み結果を読み出すことができるため，結果整合性が保証できる [13]．レプリケーションファクターはデータの冗長度を表し，$N = 3$ の場合は各入力データに対して複製を含めて全部で 3 つのデータが複数データノードで格納される．

Cassandra では，4 次元または 5 次元の連想配列のようなデータ構造をとる．

`[Keyspace] [ColumnFamily] [RawKey] [Column]`
`[Keyspace] [ColumnFamily] [RawKey] [SuperColumn] [Column]`

キースペース `Keyspace` はデータベース名，カラムファミリー `ColumnFamily` は RDB におけるスキーマ名に相当する．スーパーカラム `SuperColumn` はカラム `Column` の集合を表し，必要に応じて利用する．管理対象の値は `Column` 内に名前，値，タイムスタンプの組で格納される．`Column` の集合を持つ各エンティティのことをロウと呼び，ロウキー `RowKey` で `Column` または `SuperColumn` を識別する．

本研究では，レプリケーションファクター N はデフォルト値の 3, 読み出し時のレプリカ数 R は 1 (ONE), 書き込み時のレプリカ数 W は 3 (ALL) とした．また，`Keyspace`, `ColumnFamily` にはあらかじめ Cassandra 上に用意したキースペース，カラムファミリー名を指定し，`RowKey` にはタスク ID を用いることとした．`Column` には，パラメータ名，タスク情報のバイト列，タイムスタンプを入力する．具体的には，Submitter がタスク ID の文字列を `RowKey` に入力し，Cassandra に対してタスク情報を挿入する．Starter は，図 4 に示した ZooKeeper の管理する `/zoo/taskq/task-*/taskid` に格納されたタスク ID を読み出すことで，Cassandra に格納された担当するタスクの情報を取得することができる．

4. 予備実験

予備実験では，高可用分散協調スケジューラの耐故障性を調査するため，計算ノードに障害が発生した場合も継続して実行できるか調査する．

全ての計算ノードが故障することなく正常に実行できる場合と，一部のノードに故障が発生した場合で実行状況を比較する．失敗したタスクは，タスクキューに戻されて再実行するようにした．実験では，InTrigger クラスタの Hongo100-110 の 11 ノードを用いた．全ノードは同質であり，Intel Core2Duo 2.13GHz，メモリ 4GB，HDD 500GB を搭載しており，GiE スイッチに接続されている．

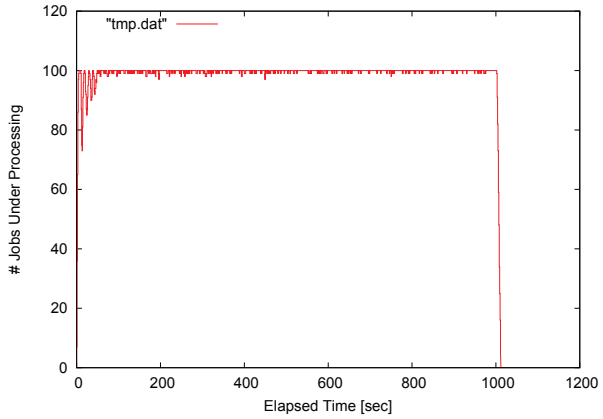

図 6　正常時の実行結果

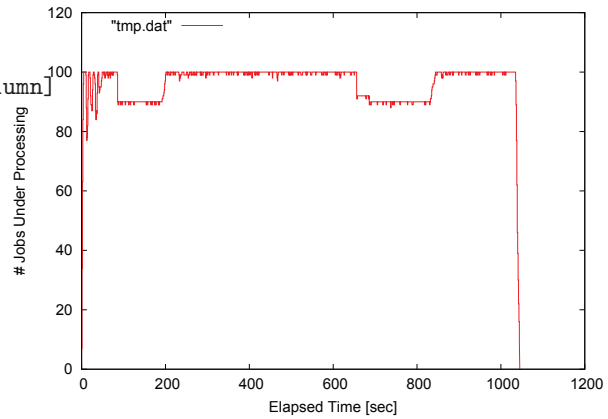

図 7　障害発生時の実行結果

ZooKeeper は ver. 3.4.3 を用いた．

図 5 に実験中の高可用分散協調スケジューラの各プロセスの構成を示す．1 台のノードに Submitter を用意し，ユーザプログラムをここで立ち上げる．残りの 10 台のノードでは，Starter を各ノードに 10 個ずつ，計 100 個用意する．In-memory FS の役割を果たす ZooKeeper と，WatchDog は，Starter が配備されているノードのうちの 3 台で実行する．故障発生時の実験では，2 つのノードでそれぞれ 120 秒間と 180 秒間，故障状態になり，その後復旧するようにした．

高可用分散協調スケジューラの機能の評価を目的としているため，実行するタスクはすべて 10 秒間スリープするものとし，1 万タスクを実行するようにした．また，本実験では改良前の資源管理機構を用い，今回実装したデータストア機構は用いていない．

図 6 と図 7 に正常時と一部のノードの故障発生時の実行結果を示す．横軸は経過時間，縦軸は実行中のプロセス数を表している．Starter は 100 個あるため，最大で 100 となっている．いずれの場合も，開始後 1 分程度プロセス数が不安定な状況が続き，立ち上がりが緩やかになっている．これは，Starter がタスクキューからタスクを取得する処理が集中しているためである．本実験は改良前の実装を用い

ており，各 Starter がそれぞれタスクキューの先頭のタスクを実行しようとして過度に集中し，失敗しているためと考えられる．図 7 では，実行中に 2 回計算ノードの故障が発生している．故障が発生した場合は全タスクの実行終了時間が長くなるものの，故障ノードに割り当てられたタスクが健全なノードで再実行され，最終的に全タスクの実行が完了していることが確認できた．

5. 関連研究

耐故障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[14], [15] 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，ワーカ数は数百程度を想定しており，ペタスケール計算機でのスケーラビリティは明らかでない．

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 [16]．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，我々の研究ではペタ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

本研究では，ポストペタスケール計算機環境における耐障害アプリケーションフレームワークの性能特性を調査することを目的とし，試験実装となる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を設計・実装した．本スケジューラを構成する資源管理機構とデータストア機構には，Apache ZooKeeper と Apache Cassandra を用いた．備実験から，処理中に計算ノードが落ちてしまった場合も，自動的にタスクが再実行されアプリケーションプログラムが継続実行できることを確認し，本スケジューラ自体の耐障害性を確認した．

今後は，ZooKeeper ノードが増加した場合の資源管理機構のスケーラビリティとデータストア機構で用いた Cassandra のオーバヘッドを調査し，耐障害アプリケーションフレームワークの技術的課題を明らかにする．

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．また，評価実験では情報爆発時代に向けた IT 基盤技術研究のための研究プラットフォーム”InTrigger”を用いた．

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 (2012).
[5] Kitaura, K., Ikeo, E., Asada, T., Nakano, T. and Uebayasi, M.: Fragment molecular orbital method: an approximate computational method for large molecules, *Chem. Phys. Lett.*, Vol. 313, pp. 701–706 (1999).
[6] Ikegami, T., Sakurai, T. and Nagashima, U.: A filter diagonalization for generalized eigenvalue problems based on the Sakurai-Sugiura projection method, *J. Comp. Appl. Math.*, Vol. 233, pp. 1927–1936 (2010).
[7] 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 (2013).
[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, pp. 1–6 (2012).
[9] Apache ZooKeeper: http://zookeeper.apache.org/.
[10] Apache Cassandra: http://cassandra.apache.org/.
[11] Gray, J. and Lamport, L.: Consensus on Transaction Commit, MSR-TR-2003-96, Microsoft Research (2004).
[12] Reed, B. and Junqueira, F. P.: A simple totally ordered broadcast protocol, *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, New York, NY, USA, ACM, pp. 2:1–2:6 (2008).
[13] Eben Hewitt: *Cassandra*, O'REILLY (2010).
[14] The MW Homepage: http://research.cs.wisc.edu/condor/mw/.
[15] 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).
[16] 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).