Vol.2012-HPC-136 No.24
2012/10/4

ポストペタスケール計算機環境に向けた高可用分散協調セルフスケジューリング機構の提案

竹房 あつ子[1,a)]　中田 秀基[1,b)]　池上 努[1,c)]　田中 良夫[1,d)]

概要：ポストペタスケール計算機環境では，階層型タスク並列が有望なプログラミングモデルの 1 つであると考えられている．タスク並列型アプリケーションでは，タスクの再実行や冗長実行により，耐障害性を備えるように設計することは比較的容易であるが，その実装は容易ではない．よって，我々はそのような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の開発を目指している．アプリケーションフレームワークでは，故障箇所を避けながら適切な計算ノード上でタスクを実行する資源管理機構が必要となるが，ポストペタスケール計算機環境でのスケーラビリティや，資源管理機構そのものの耐障害性，資源管理情報の永続化が課題となる．本稿では，スケーラブルかつ可用性の高い分散協調セルフスケジューリング機構を提案・設計する．提案する資源管理機構では，複数資源管理プロセスを分散協調させてタスクキューを管理し，タスクキュー内のタスクを各計算ノード上の実行デーモンプロセスが自律的に取得して実行する．また，各計算ノードの死活監視を行い，実行中に故障が発生した場合は選択的に再実行または削除する仕組みを提供する．資源管理プロセスの耐障害性と資源管理情報の永続化の実現方法を検討するため，Apache ZooKeeper を用いてこれらの機能を試験実装し，提案資源管理機構の妥当性と課題の明確化を行う．

キーワード：スケジューリング，高可用性，分散協調処理，耐障害性，ポストペタスケール計算

1. はじめに

2018 年にエクサスケール計算機が実現すると予想されている [1], [2]．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 日から 5 分になるとも考えられている．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．問題を静的に分割して並列処理を行う従来のデータ並列型のアプリケーションは，このようなポストペタスケール計算機環境で強スケーリングすることが困難なだけでなく，耐障害性の実現も非常に難しいと考えられている．そのため，米国 DoE が主導する産学官連携プロジェクト FOX[3] でも，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を前提とした研究開発が進みつつある．

タスク並列型のアプリケーションでは，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．具体的には，フラグメント分子軌道法 (FMO)[4] 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 [5] 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．また，耐障害性を保持しつつアプリケーションの性能の劣化を最小限に留める必要がある．

我々は，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の開発を目指している．本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

[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
[a)] atsuko.takefusa@aist.go.jp
[b)] hide-nakada@aist.go.jp
[c)] t-ikegami@aist.go.jp
[d)] yoshio.tanaka@aist.go.jp

するためのデータストア機構と，故障箇所を避けながら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．本研究では，特に資源管理機構に着目し，耐障害アプリケーションフレームワークにおける課題を明確化し，課題を解決するための資源管理機構を提案する．

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と，計算ノードの死活監視を行わなければならない．しかしながら，ポストペタスケール計算機環境でこれらを実現するためには，以下のような技術的課題がある．

スケーラビリティ 資源処理機構では，タスクの粒度，すなわち各タスクの並列度および処理時間が，小さくなるほど，資源処理プロセスに対するリクエスト数が増大し，従来のように 1 つの資源管理プロセスでは処理能力を大幅に上回ることが予想される．タスク処理要求の増大に耐えうるスケーラブルな資源管理機構が求められる．

資源管理機構そのものの耐障害性 アプリケーションプロセスを実行する計算ノードやそのノード間ネットワークの故障に対して，耐障害性を持たせる研究は数多く行われているが，資源管理機構自体に対する耐障害性はあまり考慮されていない．アプリケーションの継続実行を維持するためには，資源管理機構で利用する計算機やネットワークの一部に故障が発生したとしても，資源管理機構自体は継続して処理できなければならない．

資源管理情報の永続化 資源管理機構の継続的な処理を実現するためには，タスクの ID や処理状況，タスクを割り当てられた計算ノードの情報など，資源管理で扱う情報の永続化を行わなければならない．ファイルシステムに書き出すことで永続化する手法では，処理応答時間が長くなってしまう．また，メモリを活用する分散 KVS では，アトミック性を保証できないため，分散環境で排他制御して利用する情報の管理には適していない．

本研究では，スケーラブルかつ可用性の高い分散協調セルフスケジューリング機構を提案する．本機構では，複数資源管理プロセスを分散協調させてタスク並列処理を行うタスク群をタスクキューで管理し，各計算ノード上の実行デーモンプロセスが自律的にタスクキューから実行待ちタスクを取得して実行させる．また，各計算ノードの死活監視を行い，実行中に故障が発生した場合を検知して選択的に再実行または削除する仕組みを提供する．資源管理プロセスの耐障害性と資源管理情報の永続化の実現方法を検討するため，本研究では Apache ZooKeeper[6] を用いてこれらの機能を試験実装する．ZooKeeper は分散アプリケーションに対して分散協調サービスを提供するものであり，Hadoop で用いられる分散データベース Hbase[7] でも利用されている．試験実装により，提案する高可用分散協調セルフスケジューリング機構の妥当性と課題を検討する．

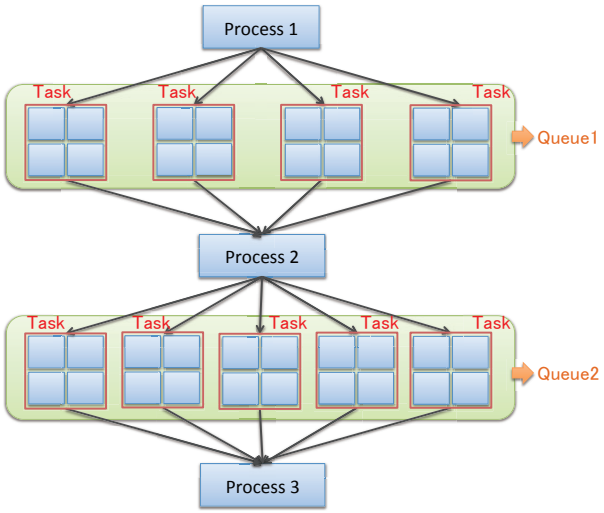

図 1 対象アプリケーションのプログラムロジックイメージ

2. 対象アプリケーションとプログラムイメージ

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は，計算に必要となるデータを障害から保全するための高信頼データストア機構と，故障箇所を避けながら適切な計算ノード上で計算を実行する資源管理機構から構成される．提案する資源管理機構では，このデータストア機構と連係して動作することを前提とする．本節では，まず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とプログラムイメージについて説明する．

2.1 対象とするアプリケーション

1 節で述べたように，ポストペタスケール計算機資源を最大限活用するためには，階層型タスク並列処理が有望なプログラミングモデルの 1 つであると考えられている．本資源管理機構においても，階層型タスク並列なアプリケーションを対象とする．

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を表す．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 Queue1 にそれらのタスクが投入される．資源管理機構は，タスク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 つのタスクが新たな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

```
FunctionId FM01;
int N;  // # of tasks
int M;  // # of required cores
TaskQueue queue = new TaskQueue(FM01, M);
for (int i = 0; i < N; i++) {
  queue.submit(IN[i], OUT[i], RESUBMIT_TRUE);
}
queue.waitAll();
```

図 2 擬似プログラム

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 プログラムイメージ

図 2 にアプリケーションフレームワークの擬似プログラムを示す．図中の FM01 はタスクが実行する関数の ID，N は総タスク数，M は FM01 の実行に必要となるコア数を表す．また，queue はタスクキューのインスタンスを表し，関数 ID とコア数 M を設定しておく．for ループ内では，submit() 関数により生成したタスクキューにタスクを投入している．ここで、submit() の引数である IN[i]，OUT[i] は，それぞれ入出力パラメータのキー配列を表す．また，RESUBMIT_TRUE では失敗時に再実行するかどうかを指定する．タスクキューへのタスク群の投入が終了すると，waitAll() 関数で全てのタスクの終了待ちを行う．ここで，指定された関数のコードの実体と各入出力パラメータの値は，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保存されており，データストアを利用して資源管理機構はタスクおよび計算ノードの管理を行う．

3. 高可用分散協調セルフスケジューリング機構の設計

耐障害アプリケーションフレームワークのための資源管理を実現するため，可用性の高い分散協調セルフスケジューリング機構を提案する．

3.1 システムアーキテクチャ

高可用分散協調セルフスケジューリング機構では，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．各モジュールの機能について，以下で説明する．

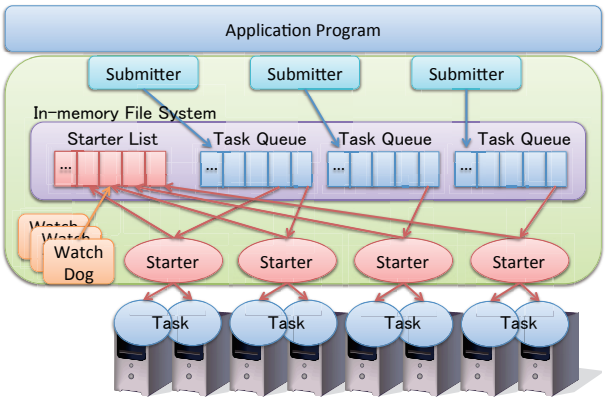

図 3 システムアーキテクチャの概要

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．また，Submitter は投入したタスクの情報をデータストアに格納しておき，Submitter 自体が故障し場合も再起動して自分の投入したタスクの状態を再度確認して継続して実行できるようにする．

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，実行結果をアプリケーションフレームワークのデータストアに格納する．

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．3.1.2 節で述べたように，Starter の故障の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に格納して，連係動作させる．

4. Apache ZooKeeper を用いた試験実装

本研究では，Apache ZooKeeper を用いて高可用分散協調セルフスケジューリング機構の試験実装を試みた．ZooKeeper の適用方法を説明するとともに，実システムの開発に向けた課題について議論する．

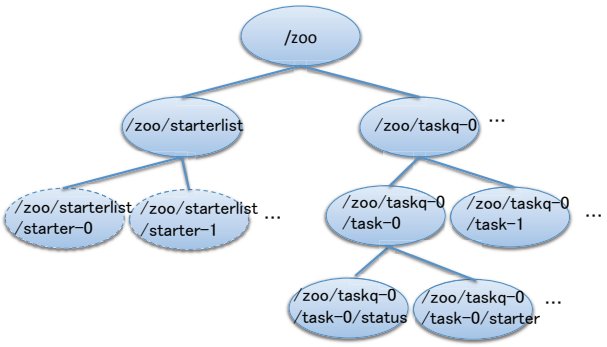

図 4 znode 構成の概要

4.1 Apache ZooKeeper の特徴

4.1.1 ZooKeeper の高可用性

ZooKeeper は，分散協調サービスを提供するものであり，Hadoop で用いられている分散データベース Hbase でも利用されている [8]．ZooKeeper は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[9] をベースとした Zab[10] 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．

4.1.2 ZooKeeper のデータモデル

図 4 に示すように，ZooKeeper はデータモデルとして znode と呼ばれるノードの階層構造のツリーを管理する．図中の楕円で示すノードを znode と呼び，各 znode では 1MB までのデータを保存することができるようになっている．この制限は，アンサンブルでデータの複製を共有するために設けられている．各 znode では，アクセス制御リスト (ACL) を設定することもできる．

znode は生成時に EPHEMERAL と PERSISTENT と呼ばれるモードのいずれかを設定することができる．EPHEMERAL モードでは，zookeeper と znode を登録したクライアントとの接続が切れたらその znode が削除される．PERSISTENT モードでは，明示的に削除されない限り znode は永続化される．また，これらのモードに加えて SEQUENTIAL を同時に指定すると，ノード名の最後にシーケンス番号を自動的につけることができる．これにより，分散システムで生じたイベントのグローバルな順序付けを行うことができる．

さらに，ウォッチと呼ばれる機能を提供することも ZooKeeper の特徴の 1 つである．znode に予めウォッチを設定しておくと，znode に何らかの変化があった場合にウォッチを設定したクライアントに通知される．ウォッチは，各 ZooKeeper サービスの操作時に設定することがで

きる．

4.2 高可用分散協調セルフスケジューリング機構への ZooKeeper の適用

高可用分散協調セルフスケジューリング機構の技術的課題を明らかにするため，ZooKeeper を用いて Java で試験実装を進めている．具体的には，In-memory FS と，計算ノードの死活監視に適用する．

4.2.1 In-memory FS への適用

3.1.4 節で述べたように，タスクキューや Starter リストを管理する In-memory FS では，分散協調する簡易なファイルシステムが必要となる．そこで ZooKeeper の提供するデータモデルを利用する．図 4 は，タスクキューおよび Starter リストを ZooKeeper 上に格納した利用イメージを表している．全ての資源管理情報は，/zoo 以下に格納することにする．

タスクキュー taskq-0 の情報は，/zoo/taskq-0 以下に格納する．/zoo/taskq-0 に投入されるタスク群の情報は，PERSISTENT_SEQUENTIAL モードで生成した znode /zoo/taskq-0/task-*に格納し，taskq-0 に投入されるタスクの投入順の管理と情報の永続化を行う．各タスクでは，タスクの実行状況とタスクが割り当てられた Starter の情報を/zoo/taskq-0/task-*/{status,starter} にそれぞれ格納する．Submitter や WatchDog は ZooKeeper にアクセスしてこれらの情報を取得することができる．

Starter リストの情報は，/zoo/starterlist 以下に格納する．Starter の情報は，各 Starter が/zoo/starterlist/starter-*に EPHEMERAL_SEQUENTIAL モードで生成，格納する．これにより，再起動された Starter を再起動前の Starter の情報と区別することが可能となるとともに，EPHEMERAL ノードとすることで Starter の死活監視に活用する．図 4 では，EPHEMERAL ノードは破線の楕円で表している．死活監視への活用方法は次節で述べる．

4.2.2 WatchDog による Starter の死活監視への適用

WatchDog は，定期的に RUNNING 状態にあるタスクの Starter 情報を調べ，Starter リストでその Starter に障害が発生していないかどうか確認する．Starter リストでは，EPHEMERAL ノードとして Starter 情報が格納されているため，Starter で障害が発生すると該当する Starter の znode が自動的に削除される．WatchDog では，該当する Starter の znode が存在しているかどうかを ZooKeeper に問い合わせるだけで，Starter の障害発生を検知することができる．タスクを割り当てられた Starter の znode が削除されていた場合は，3.1.3 節で述べたようにそのタスクの status の値を FAILED にする．

また，複数 WatchDog におけるリーダ選出にも ZooKeeper を用いる．WatchDog のリストを Starter 同様 EPHEMERAL_SEQUENTIAL モードで生成し，シーケンス番号の一番小さい WatchDog をリーダーとする．リーダー WatchDog の znode に対してウォッチを設定しておき，リーダーに障害が発生した場合は次にシーケンス番号の小さい WatchDog が処理を始めるようにする．

4.3 議論

本稿では，高可用分散協調セルフスケジューリング機構の試験実装を ZooKeeper を用いて実装することで，耐障害性を備えた資源管理機構が実現できることを確認した．一方，データストアとの連係，スケーラビリティ，および性能の点で課題が残る．

提案する資源管理機構は，アプリケーションフレームワークの高信頼データストア機構と連係して動作する．資源管理機構の評価実験を行うためには，データストア機構の実装も必要となる．よって，試験実装で用いるデータストア機構として，分散 KVS 実装の 1 つである Apache Cassandra[11] の利用を検討している．Cassandra は複数 Cassandra ノード間で分散協調して複製によるデータの冗長管理が可能であり，耐故障性を備えている．

今回の試験実装では，Starter と計算ノードとの対応を 1 対 1 となっていることを前提として実装しているが，1 対 1 ではポストペタスケール計算機環境ではスケールしない．しかしながら，1 対多とすると死活監視を階層的に行わなければならないなど，より複雑な処理が必要となる．まず，提案する 1 対 1 での試験実装で性能特性を調査した上で，1 対多での実装方法を検討していく．

また，本稿では資源管理機構の耐障害性に着目して設計を行ったが，耐障害性を考慮した実装によるアプリケーションの実効性能への影響は明らかでない．今後は試験実装を用いて耐障害性の高さとアプリケーション性能との関係を明らかにする．

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[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，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 [14]．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

本稿では，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アプリケーションフレームワークの実現のため，高可用分散協調セルフスケジューリング機構を提案・設計し，その試験実装方法を検討した．分散協調サービスを提供する Apache ZooKeeper を用いて実装することで，耐障害性を備えた資源管理機構が実現可能であることを示した．今後は，データストア機構を含めた試験実装を進め，提案資源管理機構のスケーラビリティと，耐障害性とアプリケーション性能との関係について明らかにする．

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] 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).
[5] 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).
[6] Apache ZooKeeper: http://zookeeper.apache.org/.
[7] Apache HBase: http://hbase.apache.org/.
[8] Tom Wbite: *Hadoop: The Definitive Guide*, O'REILLY (2009).
[9] Gray, J. and Lamport, L.: Consensus on Transaction Commit, MSR-TR-2003-96, Microsoft Research (2004).
[10] 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).
[11] Apache Cassandra: http://cassandra.apache.org/.
[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] 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).