Vol.2013-HPC-138 No.26
2013/2/22

スーパーコンピュータ「京」における マスタ・ワーカ型プログラミングモデルの検討

村井 均[1]　南 一生[1]　横川 三津夫[2,1]　梅田 宏明[3]　佐藤 三久[3]　辻 美和子[4]　稲富 雄一[5]　青柳 睦[5]
中島 真[6]

概要：超並列計算機の安定的・有効的な利用のためには，システム自体の耐故障性に加え，ユーザが開発するアプリケーションの耐故障性も重要である．我々は，スーパーコンピュータ「京」における耐故障機能を具備したマスタ・ワーカ型プログラミングモデルを検討中である．本報告では，次の 3 つの検討状況について報告する．1) 京のジョブ管理機構と MPI ライブラリの拡張による耐故障性の実現．2) MPI の動的プロセス生成機能と Remote Procedure Call に基づくマスタ・ワーカ型プログラミングモデルの実現．3) 本モデルに基づくフラグメント分子軌道法コードの実装および評価．

1. はじめに

スーパーコンピュータ「京」（以下，京）のように非常に多数のプロセッサを備える超並列計算機では，部品点数の増大やシステムの複雑化に伴って，いずれかの箇所で故障が発生する可能性は高くなる．したがって，計算機システムの安定的・有効的な利用という観点からは，システム自体が耐故障性を備えていることに加え [1]，ユーザが開発するアプリケーションが，アルゴリズムやプログラミングモデルのレベルで耐故障性を備えていることが重要になる．

計算の全体を管理するマスタと，マスタから割り当てられた仕事を処理する多数のワーカによって計算を進める**マスタ・ワーカ型プログラミングモデル**は，耐故障性に優れた並列プログラミングモデルとして知られている．しかしながら，ジョブ管理機構の制約のため，現在のところ京では本モデルに相当するジョブのタイプは提供されていない．

これに対し，本研究では，京においてマスタ・ワーカ型プログラムの実行環境を実装するとともに，フラグメント分子軌道（Fragment Molecular Orbital, FMO）による大規模分子の電子状態計算を例として取り上げ，本モデルの有効性を検証することを目指す．

より具体的には，次の 3 つの研究開発を進める．

- 京のジョブ管理機構および Message Passing Interface (MPI) [2] ライブラリを強化することによって，ノード異常に対する耐故障性を備えたマスタ・ワーカ型ジョブと MPI 並列処理を実現する．
- 上述の耐故障性を備えたジョブ管理機構と MPI を用いて RPC システム OmniRPC [3] を拡張し，京において RPC に基づくマスタ・ワーカ型プログラミングモデルを実現する．
- OmniRPC を用いて，FMO 法によるシミュレーションコード OpenFMO [4] を実装および評価することにより，提案するプログラミングモデルの有効性を検証する．

本報告では，これら 3 つの項目の検討状況を報告する．以下，2 章で研究の背景について述べ，3 章で京における耐故障性を備えたマスタ・ワーカ型ジョブと MPI 並列処理の実現について説明する．次に，4 章で OmniRPC によるマスタ・ワーカ型プログラミングモデルの実現について，5 章ではマスタ・ワーカ型プログラミングモデルに基づく OpenFMO の実装方式について述べる．最後に，6 章で関連研究に触れた後，7 章で本報告を総括する．

2. 背景

2.1 マスタ・ワーカ型プログラミングモデル

本研究で扱うマスタ・ワーカ型プログラミングモデルを，

[1] 理研
RIKEN
[2] 神戸大学
Kobe University
[3] 筑波大学
University of Tsukuba
[4] ヴェルサイユ大学
University of Versailles
[5] 九州大学
Kyushu University
[6] 富士通
Fujitsu

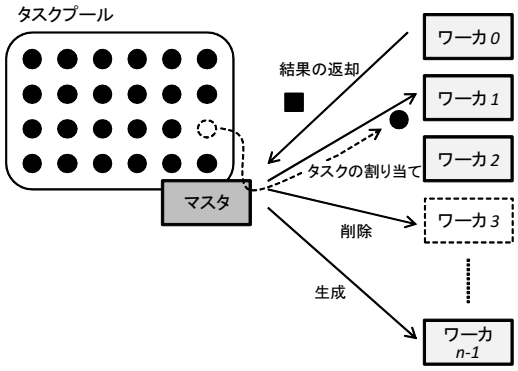

図 1 マスタ・ワーカ型プログラミングモデル

以下のように定義する（**図 1**）.

- 1 つの**マスタ**が存在する.
- マスタは，それぞれ 1 つ以上のプロセスから成る複数の**ワーカ**を生成および削除することができる.
- マスタは，（ワーカの数に比べ十分に）多数の独立な処理（タスク）のプールを持つ.
- マスタはプールからタスクを取り出し，各ワーカに 1 つずつ割り当てる.
- 各ワーカは割り当てられたタスクを行い，終了したら結果をマスタに返し，次のタスクを割り当ててもらう.

本プログラミングモデルは，次の 2 つの利点を持つ.

負荷分散（均衡化） 各タスクの負荷が大きく異なる場合であっても，負荷分散が自然に達成される.

耐故障性 あるワーカで故障が発生したとしても，（マスタさえ正常に動作していれば）マスタがそれを検知し，当該ワーカが処理中であったタスクを別のワーカに再割り当てすることにより，全体としての処理を正常に継続することができる.

京のように非常に多数のプロセッサを備える超並列計算機においては，特にこの 2 つ目の利点が重要である.

このようなマスタ・ワーカ型プログラミングモデルを実現するためには，対象の並列処理環境において以下の 3 つの機能を実現する必要がある.

- ワーカの生成
- ワーカの生存確認
- マスタ–ワーカ間通信

2.2 京におけるジョブおよびプロセスの管理

京において，ユーザの要求によって実行される処理の単位を「ジョブ」と呼ぶ．通常，一つのジョブまたはサブジョブ（後述）は，一つ以上の MPI プログラム（`mpiexec`）の実行から成る．ここで，MPI は，分散メモリ並列計算機上のプログラミング手段のデファクト標準として広く用いられているメッセージ通信ライブラリである [2].

ユーザは，ジョブの内容を記述した「ジョブスクリプト」を作成し，ジョブ投入コマンド `pjsub` を用いてシステムに「投入」する．ジョブ管理機構は，投入されたジョブスクリプトを解釈し，必要な計算ノード（以下，ノード）を割り当てる [5], [6]．あるジョブに割り当てられたノードの中には，**ジョブマスタノード**と呼ぶ特別なノードがただ 1 つ存在し，このノードがジョブスクリプトに記述されたコマンド列を実行し，MPI プログラムを起動する（`mpiexec` コマンドを実行する）.

ジョブは大きくバッチジョブと会話型ジョブに分けられ，さらにバッチジョブは以下の 3 種類のジョブタイプに分けられる.

通常ジョブ 京におけるデフォルトのジョブタイプである.

バルクジョブ 複数の「サブジョブ」から成り，確保したノード内で同時にそれらを実行する.

ステップジョブ 互いに依存関係を持つ複数の「サブジョブ」から成り，確保したノード内で指定した順序でそれらを実行する.

これらのジョブタイプは，`pjsub` コマンドのオプション（`--bulk`：バルクジョブ，`--step`：ステップジョブ）によって指定できる.

以下では，計算ノードで発生する異常を，次の 2 つに分けて考える.

ノード異常 システムの異常．さらにハードウェア（e.g. CPU, メモリ, インターコネクト・コントローラ（ICC））の異常とソフトウェア（e.g. OS，運用ソフトウェア）の異常の 2 つに分けられる.

プロセス異常 ユーザプロセスの異常.

京のジョブ管理機構は，ジョブまたはサブジョブに割り当てられている計算ノード群を監視しており，その実行に参加している任意のノードでノード異常が発生した場合，当該ジョブまたはサブジョブそのものを削除する．特に，ある時点で MPI プロセスが割り当てられていないノードにおいてノード異常が発生した場合であっても，ジョブの全体が削除される．また，MPI の動的プロセス生成の機能（`MPI_Comm_spawn`）によって生成されたプロセスで異常が発生した場合にも，ジョブの全体が削除される.

したがって，これら既存のジョブタイプまたは MPI の動的プロセス生成の機能を用いて，耐故障性を備えたマスタ・ワーカ型プログラミングモデルを実現することは困難である.

また，バルクジョブは，パラメータを変えながら同一実行モジュールを複数回実行するものであり，マスタ（この場合ジョブマスタノード）が，ワーカ（この場合サブジョブ）の挙動を制御できないという点で，マスタ・ワーカ型プログラミングモデルの機能を代替することはできない.

3. 京におけるマスタ・ワーカ型プログラミングモデルの実現

前節で述べたように，現在の京のジョブおよびプロセス

の管理の仕組みでは，マスタ・ワーカ型プログラミングモデルを実現することは困難である．そこで，我々は，ジョブ管理機構[*1]および MPI の機能を拡張し，マスタ・ワーカ型プログラミングモデルを実現することを計画している．

3.1 ジョブ管理機構と MPI の拡張

ジョブ管理機構を拡張し，新たなジョブタイプとして，**マスタ・ワーカジョブ**を設ける．マスタ・ワーカジョブは，pjsub コマンドの--mswk オプションによって指定され，2.2 節で挙げたジョブタイプに比して以下の特徴を持つ．

- ジョブの実行に参加するノードのうち，MPI プロセスが割り当てられていないノードにおいてノード異常を検知しても，当該ジョブの実行は基本的に継続される[*2]．
- MPI の動的プロセス生成の機能によって生成されたプロセスが割り当てられたノードでプロセス異常が発生しても，mpiexec の実行は継続する．
- MPI_Comm_spawn 呼出し時にジョブ管理機構が選択したノード（の一部）でノード異常が発生していた場合，それらのノードの**全て**を利用不可として，当該ジョブ内において割り当ての対象から除いた上で，再度ノード割り当てを行う．

ここで，前述のように，京では，ジョブマスタノードが mpiexec を実行して MPI プログラムを起動することに注意されたい．

さて，以上のような拡張を加えた上でも，あるプロセス（マスタ）と，動的に生成されたプロセス（ワーカ）との間の通信が失敗した場合に，それらを含む mpiexec そのものが異常終了することは避けられない．例えば，ワーカでプロセス異常が発生している状況で，そのワーカを送信先とする MPI_Send をマスタが呼び出すと，mpiexec は異常終了してしまう[*3]．したがって，安全なマスタ–ワーカ間通信のためには，ワーカの生存確認の手段が必要である．

そこで，MPI の拡張インタフェースの一つとして，MPI 関数 MPI_Comm_connect および MPI_Comm_accept をそれぞれ代替する FJMPI_Comm_connect および FJMPI_Comm_connect を提供する．これらの関数は，以下の点を除いて，元の関数と等価な機能を持つ[*4]．

- 本関数の呼出し時に，通信相手のノードでノード異常またはプロセス異常が発生していた場合，エラーコードを返して復帰する（本関数を呼び出したプロセス自体は異常終了しない）．

MPI 標準の MPI_Comm_connect および MPI_Comm_accept では，呼出し時に，通信相手のノード（ワーカ）でノード異常またはプロセス異常が発生していた場合，呼出し側（マスタ）を含めた当該 MPI プログラム全体が異常終了する．それに対し，本関数では，ワーカでノード異常またはプロセス異常が発生していても，マスタの実行は継続される．逆に，本関数の正常終了は，ワーカの生存が確認されたことを示す．

3.2 MPI に基づくマスタ・ワーカ型プログラミングモデル

前節で述べたジョブ管理機構と MPI の強化の結果，京において，MPI に基づくマスタ・ワーカ型プログラミングモデルを実現することが可能になる．

まず，マスタは，多くの場合，1 プロセスの MPI プログラムとして実行される（2 プロセス以上の MPI プログラムとすることも可能）．

次に，ワーカの生成，マスタ–ワーカ間の通信およびワーカの生存確認は以下のように実現される（**図 2**）．

ワーカの生成 FJMPI_Comm_spawn による．

ワーカの生存確認 FJMPI_Comm_connect または FJMPI_Comm_accept の正常終了による．

マスタ–ワーカ間の通信 FJMPI_Comm_connect または FJMPI_Comm_accept と，FJMPI_Comm_disconnect で挟まれた区間（以下，**マスタ・ワーカ通信区間**と呼ぶ）における MPI 通信による．

前節で述べたように，マスタにおける FJMPI_Comm_connect または FJMPI_Comm_accept の呼出し時に，通信相手のノードでノード異常またはプロセス異常が発生していてもマスタの実行は継続する．また，マスタ・ワーカ通信区間の外では，（マスタ–ワーカ間通信は起こらないので）ワーカが異常終了しても，その異常はマスタへ伝搬しない．したがって，本プログラミングモデルに従う場合，ワーカの異常に起因してマスタが異常終了するのは，マスタ・ワーカ通信区間の内で通信相手のワーカが異常終了した場合に限られる．言い換えると，常にワーカの生存確認の後に通信を行うことによって，マスタ（ジョブ全体）が異常終了する可能性を極力下げることができる．

提案方式に基づくマスタ・ワーカ型プログラミングモデルの例を以下に示す．

- ジョブ投入

```
$ pjsub --mswk ./run.sh
```

[*1] 実際には，ジョブマネージャ，ジョブスケジューラ，デーモン，管理ノードなどの複数の構成要素から成る．
[*2] ICC 故障または Port 故障の場合を除く．
[*3] このような状況は，マスタ・ワーカジョブ以外の，既存のジョブタイプでは起こらない．なぜなら、ワーカでプロセス異常が発生した時点で，mpiexec が異常終了するからである．
[*4] 本関数は，実際に，MPI 標準の機能によりエラーハンドラとして MPI_ERRORS_RETURN を設定された MPI_Comm_connect および MPI_Comm_accept そのものである．

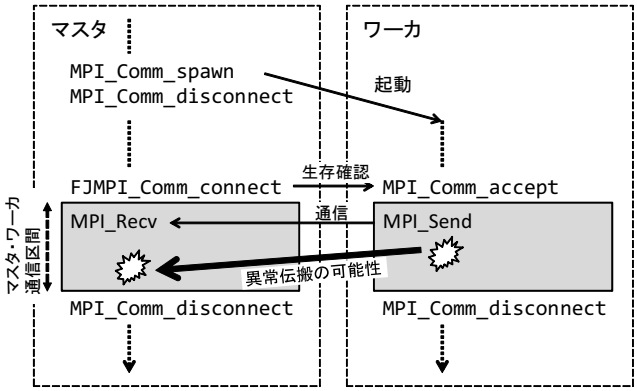

図 2 提案方式に基づくマスタ・ワーカ型プログラミングモデル

- ジョブスクリプト `run.sh`
  この例では，ジョブが使用するリソースとして 96 ノードが確保され，そのうちの 1 ノードでマスタの実行が開始される．したがって，ワーカを割り当てることができるのは残りの 95 ノードである．

```
#!/bin/bash
#PJM -L "node=96"      // 全ノード数を指定
#PJM --mpi "shape=1"   // マスタは 1 プロセ
                       // スで実行
mpiexec ./master
```

- マスタコード `master`（**図 3**(a)）
- ワーカコード `worker`（図 3(b)）
  マスタコード 14 行目およびワーカコード 11 行目の `FJMPI_Comm_disconnect` は，後続する `FJMPI_Comm_connect` または `FJMPI_Comm_accept` によるワーカの生存確認の処理に先立って，マスタとワーカのコネクションを切断するためのものである．なお，実際にはこれらの処理をユーザが直接に記述するのではなく，何らかの API の形で提供することが望ましい．本研究で開発している OmniRPC-MPI は，そのような API の一つであると考えられる．

4. OmniRPC

Remote Procedure Call (RPC) は，マスタ・ワーカ型プログラミングモデルに基づく計算の，プログラム上における典型的かつ自然な表現の一つであると考えられる．本研究では，前節で述べた京におけるマスタ・ワーカ型プログラミングモデルの上で OmniRPC [3] を動作させることにより RPC を実現することを目指す．

OmniRPC は，筑波大において，クラスタ環境から広域ネットワークで構成されたグリッド環境までシームレスな並列プログラミングを可能にする Grid RPC システムとして開発された．最新版である OmniRPC-MPI では，MPI の動的プロセス生成の機能に基づく RPC を実装中であり，

```
1  #include <mpi-ext.h>
2
3  int main(int argc, char **argv)
4  {
5    MPI_Init(&argc, &argv);
6
7    // ワーカを生成
8    FJMPI_Comm_spawn("./worker", ...,
9                     &worker_comm, ...);
10
11   // ワーカからポート名を受信
12   MPI_Recv(worker_port, ...,
13            worker_comm, ...);
14
15   // ワーカとのコネクションを切断
16   FJMPI_Comm_disconnect(&worker_comm);
17
18   // マスタ-ワーカ間通信
19   FJMPI_Comm_connect(worker_port, ...,
20                      &worker_comm);
21   MPI_Send(..., worker_comm);
22   MPI_Recv(..., worker_comm, ...)
23   FJMPI_Comm_disconnect(&worker_comm);
24
25   MPI_Finalize();
26 }
```

(a) マスタコード

```
1  int main(int argc, char **argv)
2  {
3    MPI_Init(&argc, &argv);
4
5    MPI_Comm_get_parent(&master_comm);
6
7    // マスタへポート名を送信
8    MPI_Send(worker_port, master_comm, ...);
9
10   // マスタとのコネクションを切断
11   MPI_Comm_disconnect(&master_comm);
12
13   // マスタ-ワーカ間通信
14   MPI_Comm_accept(worker_port, ,,,,
15                   &master_comm);
16   MPI_Recv(..., master_comm, ...);
17   MPI_Send(..., master_comm);
18   MPI_Comm_disconnect(&master_comm);
19
20   MPI_Finalize();
21 }
```

(b) ワーカコード

図 3 提案方式によるコードの例

前章までで述べた京におけるマスタ・ワーカ型プログラミングモデルとの親和性が高い．

4.1 クライアントプログラム

クライアントプログラム（マスタ側プログラム）は，OmniRPC-MPI で追加された，以下のような API 群を用いて作成し，コンパイラ・ドライバ `omrpc-cc-mpi` でコンパイルすることによって得られる．

- `OmniRpcHandle *OmniRpcMpiCreateHandle(int nprocs, char *pname)`
  リモート実行プログラム `pname` を並列数 `nprocs` で起動し，それに対応する `OmniRPCHandle` ハンドルを返す．
- `OmniRpcRequest *OmniRpcMpiCallAsyncByHandle (OmniRpcHandle *handle, int nprocs, char *fname, ...)`
  `handle` に対応するリモート実行プログラムに含まれる関数 `fname` を並列数 `nprocs` で非同期的に呼び出し，それに対応する `OmniRpcRequest` ハンドルを返す．引数 `fname` の後には，`fname` に対する引数の列が続く．

これらは，従来の API における `OmniRpcCreateHandle` および `OmniRpcCallAsyncByHandle` に，それぞれ対応する．従来の API は rsh などの機構によって実装されていたのに対し，これらは MPI の動的プロセス生成の機能に基づく．

OmniRPC-MPI によるクライアントプログラムの例を**図 4** に示す．

```
1  int main(int argc, char **argv)
2  {
3    OmniRpcHandle handle;
4    OmniRpcRequest req;
5
6    OmniRpcInit(&argc,&argv);
7
8    // 256ノードでリモート実行プログラム
9    // workerを起動.
10   handle =
11      OmniRpcMpiCreateHandle(256, "");
12
13   // workerに含まれる関数worker_funcを
14   // 非同期的に呼び出す.
15   req =
16      OmniRpcMpiCallAsyncByHandle(handle,
17                    256, "worker_func", ...);
18
19   // worker_funcの終了を待つ.
20   OmniRpcWait(req);
21
22   OmniRpcHandleDestroy(handle);
23
24   OmniRpcFinalize();
```

図 4 OmniRPC-MPI によるクライアントプログラムの例

4.2 リモート実行プログラム

OmniRPC-MPI におけるリモート実行プログラム（ワーカ側プログラム）は，MPI による並列プログラムであってよい．リモート実行プログラムは以下の手順によって生成する．

(1) インタフェースを定義する IDL（Interface Description Language）ファイルを作成する．
(2) IDL ファイルから，`omrpc-cc-mpi` でリモート実行プログラムを生成する．

IDL の構文，および IDL ファイルからリモート実行プログラムを生成する手順は，従来の OmniRPC と基本的に同様であるため，本稿では詳しい説明を省略する．詳しくは，文献 [3] を参照されたい．

5. OpenFMO

OpenFMO [4] は，超並列 FMO 計算を行うことを目的として，九州大学と九州先端科学技術研究所で開発された並列 FMO プログラムである．非経験的第一原理電子状態計算手法の中で最も単純な Hartree-Fock（HF）法を基にした FMO 計算に特化したプログラムであるためコードが比較的短く（全体で約 54,000 行），MPI を用いた並列化が行われているため実行プロファイル取得および最適化作業が行いやすい．

FMO 法の階層的な構造は，マスタ・ワーカ型プログラミングモデルによる並列化に適している [7]．

本研究では，OmniRPC-MPI を用いて OpenFMO を並列化して京において実行することにより，我々が提案するマスタ・ワーカ型プログラミングモデルの有効性を評価することを目指している．

図 5 に，OmniRPC-MPI 版 OpenFMO の構成を示す．OmniRPC-MPI 版 OpenFMO は，マスタ，ワーカおよびメモリサーバの 3 種類の構成要素から成る．

マスタ 1 個のプロセスから成る MPI プログラム．RPC によりワーカおよびメモリサーバを起動する．また，各ワーカーが計算すべきフラグメントの割り当てと結果の集計を行う．

ワーカ 1 個以上のプロセスから成る MPI プログラム．マスタからの割り当てに応じて，フラグメント電子状態計算を行う．

メモリサーバ 1 個以上のプロセスから成る MPI プログラム．FMO 計算で必要な中間データ（モノマー密度行列データなど）を保存して，そのデータに対するワーカーからのアクセス（参照，更新）要求に応答する．

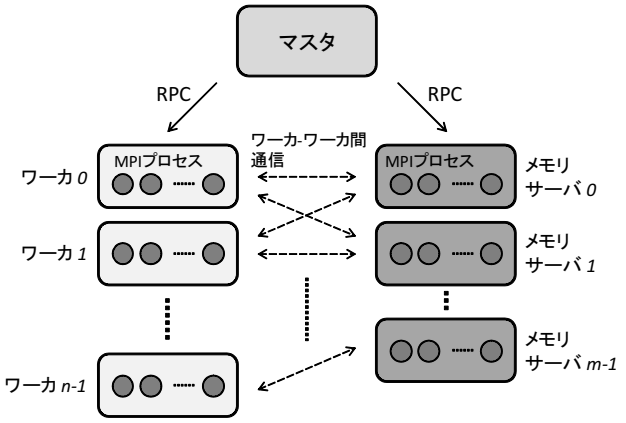

図 5 OmniRPC 版 OpenFMO の構成

マスタから見た場合，メモリサーバは単に特別なワーカである．

ここで，任意のワーカは，MPI の通信確立の機能により，任意のメモリサーバと通信を行い得ることに注意されたい．本プログラミングモデルの下では，マスタ–ワーカ間通信の場合と同様に，3.1 節で述べた `FJMPI_Comm_connect` および `FJMPI_Open_port` を用いて，安全なワーカ–ワーカ間通信を記述できる．

6. 関連研究

グリッドコンピューティングの分野では，自然なプログラミングモデルとして RPC が利用される．したがって，これまでに，OmniRPC と同様のグリッド RPC システムも多く研究されており [8], [9], [10]，大規模なシミュレーションへ適用することも行われている．また，ヴェルサイユ大で開発されている YML [11] では，ワークフロー言語で定義される並列アプリケーションにおいて，スケジューラが各ワーカを起動する機構として OmniRPC が用いられている．これに対し，本研究は，京のような単一の超並列計算機システムを対象として，耐故障性の付与を目的に MPI の動的プロセス生成に基づく RPC を実現するものである．

5 章でも述べたように，その階層的な性質から FMO 法は RPC またはマスタ・ワーカ型プログラミングモデルとの親和性が高い．多くの研究において，それらの方式により並列化が行われている [12], [13].

7. おわりに

京における耐故障性を備えたマスタ・ワーカ型プログラミングモデルについて，以下の 3 つの項目の検討状況を報告した．

- 京のジョブ管理機構および MPI を強化することによって，一部のノードで異常が発生してもジョブの実行が継続される「マスタ・ワーカジョブ」を実現する．
- MPI の動的プロセス生成機能に基づく RPC システム OmniRPC-MPI を実装し，RPC に基づくマスタ・ワーカ型プログラミングモデルを実現する．
- マスタ，ワーカ，メモリサーバから成る OpenFMO の実装方式を示し，これを OmniRPC-MPI を用いて記述することにより，提案するモデルの有効性を検証する．

今後，以上の各項目の実装を進め，我々が提案するプログラミングモデルの有効性を検証する計画である．

参考文献

[1] 黒川原佳，庄司文由：スーパーコンピュータ「京」：2. システム概要—世界トップクラスの演算性能と使いやすさを両立—，情報処理，Vol. 53, No. 8, pp. 759–766 (2012).
[2] Message Passing Interface Forum: MPI: A Message Passing Interface Standard Version 3.0, `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf` (2012).
[3] Sato, M., Boku, T. and Takahasi, D.: OmniRPC: a Grid RPC System for Parallel Programming in Cluster and Grid Environment, *Proc. CCGrid2003*, Tokyo, pp. 206–213 (2003).
[4] OpenFMO Project: `http://www.openfmo.org/OpenFMO/`.
[5] 宇野篤也，加藤丈治，宮本巧輝，岩田章孝，長屋忠男：スーパーコンピュータ「京」：4. システムソフトウェア-OS，運用管理ソフトウェア，ファイルシステム-，情報処理，Vol. 53, No. 8, pp. 774–779 (2012).
[6] 宇野篤也，庄司文由，横川三津夫：ファイルステージングのあるジョブスケジューリングの評価，情報処理学会研究報告，Vol. 2012–HPC–136, No. 22, pp. 1–6 (2012).
[7] Inadomi, Y., Takami, T., Maki, J., Kobayashi, T. and Aoyagi, M.: RPC/MPI Hybrid Implementation of OpenFMO – All Electron Calculations of a Ribosome, *Advances in Parallel Computing*, Vol. 19, pp. 220–227 (2010).
[8] Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T. and Matsuoka, S.: Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing, *J. Grid Computing*, Vol. 1, pp. 41–51 (2003).
[9] Caron, E. and Desprez, F.: DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid, *Int'l. J. High Performance Computing Applications*, Vol. 20, No. 3, pp. 335–352 (2006).
[10] Yarkhan, A., Seymour, K., Sagi, K., Shi, Z. and Dongarra, J.: Recent Developments in Gridsolve, *Int'l. J. High Performance Computing Applications*, Vol. 20, No. 1, pp. 131–141 (2006).
[11] YML: `http://yml.prism.uvsq.fr/`.
[12] Ikegami, T., Ishida, T., Fedorov, D. G., Kitaura, K., Inadomi, Y., Umeda, H., Yokokawa, M. and Sekiguchi, S.: Full electron calculation beyond 20,000 atoms: Ground electronic state of photosynthetic proteins, *Proc. Supercomputing 2005* (2005).
[13] Ikegamial, T., Maki, J., Takami, T., Tanaka, Y., Yokokawa, M., Sekiguchi, S. and Aoyagi, M.: GridFMO – Quantum chemistry of proteins on the grid, *Proc. 8th IEEE/ACM Int'l. Conf. on Grid Computing (GRID 2007)*, IEEE, pp. 153–160 (online), DOI: http://dx.doi.org/10.1109/GRID.2007.4354128 (2007).