Vol.2012-HPC-135 No.11
2012/8/1

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

辻 美和子[1,a)]　佐藤 三久[1,2,b)]　Maxime Hugues[3,c)]　Serge Petiton[3,4,d)]

概要：本稿では，ポストペタスケール計算機環境のための階層型プログラミング環境およびプログラミングツールである FP2C (Framework for Post-Petascale Computing) を提案・開発する．FP2C は，大規模で複雑な科学技術計算のために，複数の並列タスクを上位層にワークフローを用いることで統合する．並列タスクは並列プログラミング言語 XcarableMP (XMP) により記述される．これらの並列タスクは，ワークフロー開発・実行環境 YML により統合される．本稿では，並列タスクの実行と管理のために YML のミドルウエアである OmniRPC を，XMP コンパイラによる並列タスクの生成をサポートするために YML タスクジェネレータを，並列タスクを用いたアプリケーションの実行のためにワークフロースケジューラを，それぞれ拡張した．実験により，並列タスクによりタスク数を減らすことでワークフローの複雑さを軽減すること，逆にワークフローにより巨大な並列タスクを複数の並列タスクに置き換えることで通信コストを削減できることがわかった．

1. はじめに

近年では，複数のスーパーコンピュータの性能がペタフロップスに到達し，ポストペタスケールを見据えた研究が行われている．ポストペタスケールシステムは，多次元トーラスのような局所性のあるネットワークにより接続され，メニーコアやアクセラレータを備えた，階層的な構造を持つ巨大なシステムになると考えられている．ペタフロップ計算に際しては，フラット MPI よりも OpenMP/MPI ハイブリッドのような，分散メモリ型と共有メモリ型の階層的な並列処理が使用されている．これは，膨大なコアをすべて MPI 並列で扱うことは効率的でなく，ハイブリッド化により分散並列数を減らすことで，通信コストが削減される，負荷分散が容易になる，などのメリットがあるからである．しかし，コア数やノード数がさらに数十倍から数百倍に増加すれば，OpenMP/MPI ハイブリッドは再びフラット MPI と同じ問題 — 並列数の増加に起因する通信時間の増加，負荷分散の困難さなど — にぶつかる可能性がある．本稿では，ポストペタスケール環境のために，新たな階層型プログラミング環境 FP2C (Framework for Post-Petascale Computing) を提案する．

FP2C は複数の並列タスクをワークフローに統合する．本稿では，「並列タスク」という用語は並列で実行されるタスクという意味で用いる．並列タスクの記述のために，並列プログラミング言語 XcalableMP (XMP) [4] をサポートする．XMP では，逐次プログラムを指示文を挿入することで，平易に分散並列プログラムを記述することができる．ワークフローは YML [2], [3] により実現される．YML は，依存関係を持つタスクを有向グラフとして表現するワークフロー開発実行環境である．

YML はサブクラスタやノードグループの間で機能し，並列タスクを管理する．XMP により記述され，XMP コンパイラによってコンパイルされた並列タスクは，サブクラスタ内のノードの集合やクラスタ内で比較的近接したノードの集合上で実行される．XMP は OpenMP および GPU プログラミングへの拡張が進められているため，将来的には FP2C は 3 つの層，すなわち (1) ワークフロー，(2) データ並列，(3) スレッド並列およびアクセラレータ，から構成される．

本稿では，

- FP2C を提案し
- その実装について延べ
- 実験により FP2C の性能と性質を明らかにする．

[1] 筑波大学計算科学研究センター
305-8577 茨城県つくば市天王台 1-1-1
[2] 筑波大学システム情報工学研究科
305-8573 茨城県つくば市天王台 1-1-1
[3] INRIA, France
[4] University of Lille, France
[a)] tsuji@hpcs.cs.tsukuba.ac.jp
[b)] msato@cs.tsukuba.ac.jp
[c)] maxime.hugues@inria.fr
[d)] Serge.Petiton@lifl.fr

```
#pragma xmp nodes p(4)
#pragma xmp template t(0:7)
#pragma xmp distribute t(block) onto p
int a[8];
#pragma xmp align a[i] with t(i)

int main(){

#pragma xmp loop on t(i)
    for(i=0; i<8; i++)
        a[i]=0;
    ...
```

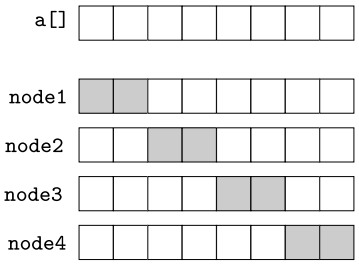

図 1 XMP プログラミングとデータ分散
Fig. 1 XMP programming and data distribution

本稿の構成は以下である：次章で FP2C を構成する XMP および YML について簡単にふれたのち，3 章で FP2C の概要と実装について述べる．4 章で実験を行う．5 章で関連研究について延べ，6 章でまとめる．

2. 背景

FP2C は，XMP および YML から構成される．XMP は，MPI のような分散メモリシステムに対する並列プログラミングを，OpenMP のような指示文で記述するプログラミング言語であり，並列タスクの記述方法を提供する．YML は，ワークフロー開発・実行環境である．

2.1 XcalableMP (XMP)

XcalableMP (XMP) [4] は，分散メモリ型システム上における並列プログラミング言語である．逐次コードに指示文を挿入することで並列プログラミングが可能であり，C と FORTRAN がサポートされている．XMP の仕様は XMP ワーキンググループ [6] によって定義され，仕様に基づき XMP コンパイラが開発されている．XMP コンパイラは XMP 指示文の挿入された C のソースコードを，XMP ランタイムライブラリ呼び出しを含む C ソースコードに変換する．XMP ランタイムは MPI ライブラリを通信レイヤとして用いており，変換後のソースコードは MPI コンパイラによって実行形式の並列プログラムにコンパイルされる．XMP はグローバルビューと呼ばれるデータ分散に基づく並列化，およびローカルビューと呼ばれる各ノードが持つローカルデータに対する通信をサポートする．

図 1 に XMP プログラミングの例を示す．*nodes* 指示文はプログラムを実行するノード集合とトポロジを定義する．*template* 指示文は仮想的なインデックス配列であり，*distribute* 指示文によってノード集合へ関連付けられる．分散方法としては，*block*, *cyclic* および *block-cyclic* がサポートされている．*align* 指示文は配列をテンプレートを対応させる．配列の各要素は対応するテンプレートが割り当てられたノードに配置される．

2.2 YML

YML[2], [3] はワークフロー開発実行環境である．ワークフローは入出力によって互いの依存関係を定義されるタスクの集合から成る．YML は各タスクおよびワークフローの開発のために 3 種類のソースコードをサポートする，すなわちアブストラクト，インプリメンテーション，アプリケーションのソースコードである．アブストラクト・コード (図 2) は，タスクのインターフェース (入出力) を定義する．インプリメンテーション・コード (図 3) は，主に C/C++で記述され，タスクの処理を定義する．YML のジェネレータはこの記述と，アブストラクトで定義されるインターフェースをあわせて，実行可能なプログラムを生成する．アプリケーション・コード (図 4) においては，YvetteML と呼ばれるグラフ記述言語によってワークフローが定義される．この記述は YML コンパイラによってタスク間の依存関係を表現する有向グラフに変換される．

YML スケジューラはワークフローを実行する．YML は P2P とクラスタという 2 種類の計算環境を考慮している．前者のミドルウェアとして XtreamWeb[1] が用いられ，後者のミドルウェアとして OmniRPC[5] が用いられる．また，スケジューラとミドルウエア間の依存関係を解決するために，各ミドルウエアのためのバックエンドを用いる．本稿では，マスターワーカー型グリッド RPC をサポートする OmniRPC を拡張する．

図 5 は，YML におけるワークフローの実行の様子を示す．YML スケジューラは omnirpc-agent を起動し，エージェントは各リモートノードにワーカーと呼ばれるプログラムを起動する．ワーカーとスケジューラはエージェントを介してタスクのリクエストをやりとりする．リクエストを受け取ったワーカーは，特定のタスクプログラムを実行する．これらのマスター，エージェント，ワーカーの間のメッセージは TCP/IP ソケット通信によりなされる．

3. FP2C Framework for Post-Petascale Computing

本性では提案する階層型並列プログラミング環境 FP2C (Framework for Post-Petascale Computing) について述べる．

3.1 概要

図 6 に FP2C の概観を示す．FP2C は XMP により記述

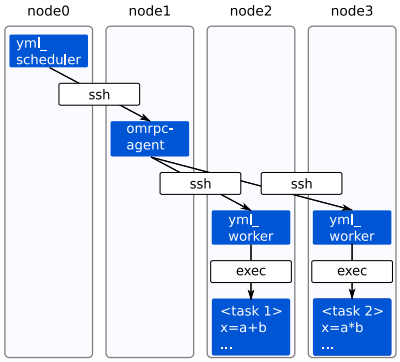

図 5 YML におけるワークフローの実行
Fig. 5 Execution of workflow in YML

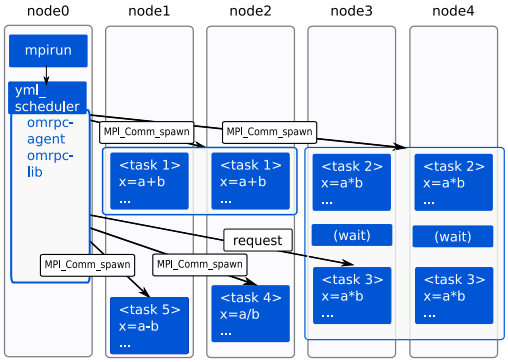

図 7 FP2C におけるワークフローの実行．タスクは並列プログラムである
Fig. 7 Execution of workflow in FP2C

された並列タスクを，より上位のプログラミングとして YML ワークフローを用いることで，統合する．近年の学際的な研究などにおいては，互いに依存するさまざまな分野のプログラムを何度も実行することが珍しくない．そのような互いに依存するプログラムをデータ依存性などを考慮しながら実行し，管理することは容易ではないが，YML は各プログラムをワークフローのタスクみなし，互いに依存するタスクの実行・管理を平易に自動化する手段を提供する．オリジナルの YML におけるタスクは逐次的に計算されたが，本稿における拡張で並列タスクが導入される．並列タスクの記述言語として XMP をサポートする．XMP は OpenMP や GPU プログラミングへの対応が検討されているため，FP2C により以下の 3 階層の階層型並列プログラミングが可能になる：

- ワークフロー
- データ並列
- スレッド並列，アクセラレータ

FP2C においては，YML は階層型のシステムの上位に位置し，クラスタ間あるいはノードグループ間で機能する．XMP 並列タスクはノードグループ内で機能する．ノード内のスレッド並列計算や GPU 計算も将来的にはサポートされる．使用例としては，並列プログラムを用いたパラメータサーチなどが考えられる．複数の並列プログラムに異なるパラメータを渡し，計算を実行させる．各並列プログラムは，ワークフローから見ればタスクである．並列プログラムの規模としては，現在のスーパーコンピュータ上で実行されているプログラム程度が想定される．これらは，スレッド並列やアクセラレータを用いた，ハイブリッド並列プログラムであっても良い．別な例としては，大規模並列プログラムをいくつかに分割し，それぞれをもとのプログラムよりも小規模な並列タスクとして実行し，最後に結果をまとめる使い方も考えられる．大規模並列プログラムは実行時間の多くが通信やそれに伴うレイテンシで占められるため，並列数を落とすことで全体の効率化が期待される．また，別な例としては，ワークフローアプリケーションをより大規模なデータで実行する場合に，逐次タスクではノード内のメモリにデータが収まらない可能性がある．逐次タスクを分散並列タスクに置き換えることで，ノードあたりのメモリ量を削減し，アプリケーションを実行可能にすることができる．

3.2 実装

図 7 に，FP2C によるワークフローの実行を示す．YML スケジューラは mpirun により実行される．図 5 に示されたような従来の OmniRPC エージェントの機能は，ライブラリとして YML スケジューラにリンクされる．YML スケジューラは OmniRPC ライブラリを通して，MPI_Comm_Spawn により任意のタスク関数を含むリモートプログラムを並列に起動する．従来の YML (図 5) と異なり，ワーカープログラムは用いない．

本節では，FP2C により階層型プログラミングの実現のために行った以下の実装について述べる：

- OmniRPC の MPI 向け拡張，およびこれを YML スケジューラから使用するための MPI バックエンドの構築
- XMP で記述されたインプリメンテーションソースコードから実行プログラムを生成するためのジェネレータの構築
- 並列実行のための情報をワークフローのグラフに付加するための YML コンパイラの拡張

3.2.1 OmniRPC の拡張と MPI バックエンド

本研究では，MPI をサポートしリモートで並列プログラムを実行するよう OmniRPC を拡張した．従来の OmniRPC はエージェントプログラムを用いたが，本稿で拡張された OmniRPC においては，エージェントを含めた OmniRPC の機能はすべてライブラリとして提供される．また，従来の OmniRPC は TPC/IP ソケット通信を行っていたが，本実装では TCP/IP のユーザによる使用をサポートしないシステムでも FP2C を実行するために，すべての通信は MPI により実行される．

```
<?xml version="1.0">
<component type="abstract" name="add">
  <params>
    <param name="c" mode="out"type="real">
    <param name="a" mode="in" type="real">
    <param name="b" mode="in" type="real">
  </params>
</component>
```

図 2 インターフェースを定義するアブストラクト
Fig. 2 Abstract, which defines an interface of task.

```
<?xml version="1.0">
<component type="implmentation" name="add"
 abstract="add" description="c=a+b">
  <impl lang="CXX">
    <header>
#include<stdio.h>
    </header>
    <source>
      c=a+b;
    </source>
  </impl>
</component>
```

図 3 インプリメンテーション．このソースコードとアブストラクトで定義されるインターフェースに基づき，実行形式のプログラムが生成される
Fig. 3 Implementation. Based on this source code and the interface defined in Abstract, an executable is generated.

```
<?xml version="1.0"?>
<application name="sample">
  <params>
    <param name="A" mode="in" type="real">
    ..
  </params>
  <graph>
  par
    compute add(C,A,B);
    notify(evtC);
    ..
  //
    wait(evtC);
    wait(evtD);
    compute add(E,C,D);
  endpar

  par (i:=1; 10) do
    compute hoge(..
    ...
  enddo
  </graph>
</application>
```

図 4 アプリケーション．YvetteML 言語によりワークフローが定義される．タスクの実行 (call)，イベントの操作 (notify, wait)，条件 (if)，並列実行 (par, par do) などが定義される
Fig. 4 Application. A workflow is defined in YvetteML, which provides statements to execute tasks (call), manipulate evenst (notify, wait), define conditions (if), define parallel execution (par, par do), etc.

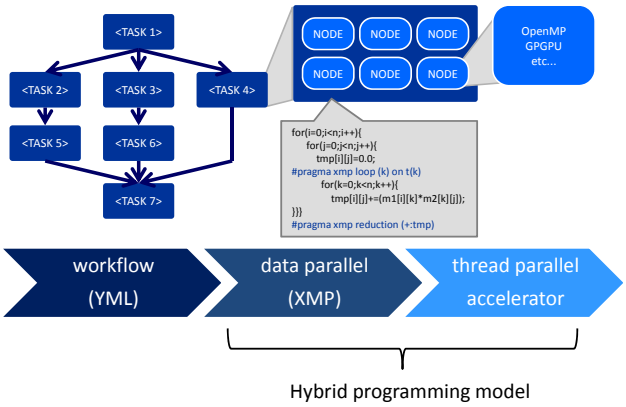

図 6 FP2C の概観
Fig. 6 Overview of FP2C

拡張された OmniRPC は主に以下の 4 つの機能を提供する：

(1) 並列リモートプログラムの起動
(2) 特定のタスク関数を実行するためのリモートプログラムへのリクエストの送信
(3) リモートプログラムからのメッセージ (おもにタスクの終了信号) の受信
(4) リモートプログラムとリソースの管理

クライアント (YML スケジューラ) とリモートプログラムの間の非同期の通信のために，pthread が用いられる．マススレッドが上記のうちの最初の 2 つの機能を提供し，もういっぽうのスレッドが 3 つ目の機能を提供する．(4) は，リモートプログラムおよびそのプログラムの持つタスク関数をリストし，リモートプログラムの状態 (実行中・アイドル) を記録し，また，空きプロセスの数をカウントする．これらの情報は，両スレッドにより更新される．

マススレッドはリモートプログラムを起動し，リモートプログラムにクライアント (YML スケジューラ) からのリクエストを送信する．リモートプログラムの起動には，MPI_Comm_spawn 関数が用いられる．MPI_Comm_spawn 関数はリモートノードに MPI プロセスを生成し，これらのプロセスの間とのコミュニケータを返す関数である．1 つのリモートプログラムは，複数の異なる種類のタスクを関数として持つことができる．もしも，すでに起動しているリモートプログラムが，YML スケジューラから要求されたタスクを持ち，要求されるプロセス数で起動しており，かつ，そのリモートプログラムがアイドル状態であるとき，OmniRPC は新たなリモートプログラムの起動は行わず，すでに実行されているリモートプログラムに対してタスク実行リクエストの送信だけを行う．そうでなければ，指定されたタスクを持つリモートプログラムを指定されたプロ

セス数で起動し，起動したリモートプログラムにタスクの実行をリクエストする．起動に際して，十分な数の空きプロセスがないとき，クライアントはアイドル状態の不要なプログラムに終了信号を送る．それでも空きプロセスが足りないときは，しばらく待つ．

このようにリモートプログラムを再利用することで，新たにリモートプログラムを起動するオーバーヘッドを防ぐことができる．例えば，パラメータサーチなどでは，同じ並列タスクを入力パラメータだけを変えて何度も繰り返すが，クライアントはそれぞれのパラメータに対して新たにリモートプログラムを起動するのではなく，すでに起動しているリモートプログラムにパラメータを送信するだけで良い．また，1 つのリモートプログラムは複数のタスクを関数として持つことができるため，異なるタスクに関しても並列数などの条件が合致すれば，同様にリモートプログラムの再利用が可能である．ゆえに，リモートプログラムの起動の回数，すなわち MPI_Comm_spawn 関数のコール回数は，ワークフローにおけるタスクのコール回数よりもずっと少ない．

各タスクの終了処理は，もう一方のスレッドで実行される．このスレッドは，クライアントとリモートプログラムの間のコミュニケータを繰り返しチェックする．リモートプログラムからタスク関数の終了信号が送信されていたとき，そのリモートプログラムをアイドルプログラムのリストに追加する．タスク関数を終了したリモートプログラムは，再びクライアントからのメッセージ — 別のタスク関数の実行リクエスト，もしくはプログラム自身の終了のリクエスト — が届くのを待つ．

YML スケジューラから拡張 OmniRPC を使用するために，MPI バックエンドを構築した．MPI バックエンドは，YML スケジューラから使用される動的ライブラリであり，拡張 OmniRPC の API にタスクの名前や必要なプロセス数を渡す．

3.2.2 XMP を用いたタスクジェネレータ

図 8 に，XMP により記述されたインプリメンテーションコードを示す．ノード数や配列の分散などの情報は，XMP 指示文ではなく，XML のフォーマットで記述される．これは，これらの情報が XMP プログラムのみならず，YML 側からも利用されるためである．

本稿では，YML タスクジェネレータを XMP をサポートするために改良した．図 9 に，リモートプログラムの生成の過程を示す．まず，インプリメンテーションソースコード (図 9, test.query) から，XMP ソースコード (図 9, test.c) が抽出される．XML により指定されたノードやデータ分散の情報は XMP 指示文を用いて書き直される．このソースコードは main 関数を持たない．XMP コンパイラによりこのソースコードをオブジェクトファイルに変換する (図 9, test.o)．同時に，OmniRPC IDL (IDL=Interface Definition Language) ファイルが生成される (図 9, test.idl)．IDL ファイルにはタスク関数に対する関数呼び出しが記述される．オリジナルの OmniRPC は，IDL ファイルから C ソースコードを生成するために Ninf IDL コンパイラを提供しているが，本稿ではこれを並列タスク向けに拡張して IDL ファイルから MPI-C ソースコードを生成する．生成されたコード (図 9, test.rex.c) は，IDL に記述された関数呼び出しの他に，RPC のためのインターフェースも記述される．MPI コンパイラにより，このコードからオブジェクトファイルを生成する (図 9, test.rex.o)．このオブジェクトファイルとタスク関数を含むオブジェクトファイル (test.o) をリンクして，実行形式のリモートプログラム (図 9, test.rex) が完成する．これらの一連の処理は YML タスクジェネレータにより自動的に実行される．ここでは 1 つのリモートプログラムが 1 つのタスクを持つ例を示したが，前述のように，1 つのリモートプログラムが複数のタスクを持つことができる．これについては，3.2.4 で述べる．

```
<?xml version="1.0">
<component type="implmentation" name="copy" abstract="copy"
                                 description="copy a matrix">
  <impl lang="XMP" nodes="(2,2)">
  <templates>
    <template name="t" size="100,100" format "block,block" />
  </templates>
  <distribute>
    <param template="t" name="In" size="100,100" />
    <param template="t" name="Out" size="100,100" />
  </distribute>
  <source>
#pragma xmp loop (j,i) on t(j,i)
  for(i=0;i<n;i++){
    for(j=0;j<n;j++){
      Out[i][j]=In[i][j];
    }
  }
  </source>
  </impl>
</component>
```

Data mapping

nodes (1,1)	nodes (2,1)
nodes (1,2)	nodes (2,2)

図 8 XMP インプリメンテーションコード
Fig. 8 XMP implementation code

3.2.3 YML コンパイラの拡張

YML コンパイラは YvetteML により記述されたワークフローを有向グラフへと変換する．グラフの接点がタスクである．本稿では，これを拡張して，有向グラフに各タスクを実行するために必要なプロセス数を付加する．YML コンパイラは，インプリメンテーションコードに XML 形式で指定されたノード数を抽出し，グラフに付加する．YML スケジューラは，タスク実行時にこの情報を MPI バックエンドに渡し，MPI バックエンドがこれを OmniRPC に渡す．

3.2.4 リモートプログラムへの複数のタスクのバインド

MPI_Comm_spawn による並列プログラムの起動のコストは，しばしば無視できない大きさになる．そこで，起動の回数を減らし，既に起動しているリモートプログラムの再利用性を高めるために，複数のタスクを 1 つのリモー

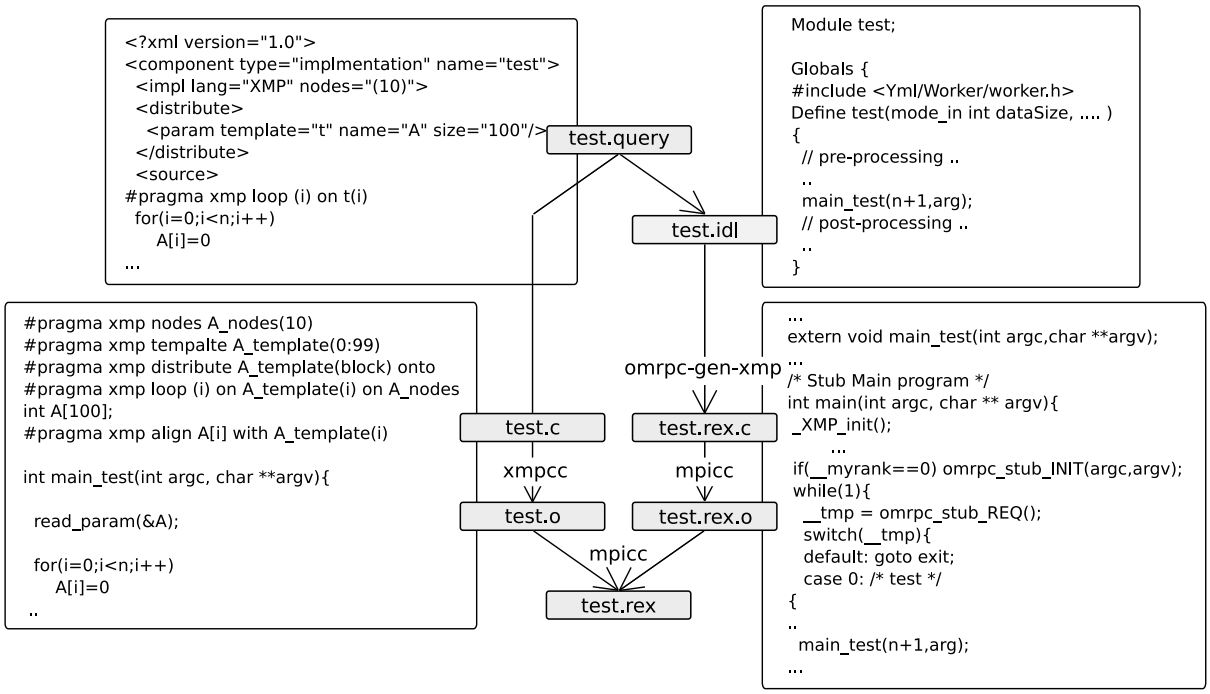

図 9 リモートプログラムの生成
Fig. 9 Generation of a remote execution program

トプログラムにまとめるツールを提供する．このツールは YML コンパイラにより得られたグラフに対して，各タスクとそのタスク実行に必要なプロセス数を調査し，同じ数のプロセスで実行されるタスクを 1 つのプログラムにまとめる．このツールは，複数のタスクへの関数呼び出しが記述された IDL ファイルを生成し，この IDL ファイルと各タスクのオブジェクトファイルから図 9 と同様にリモートプログラムを生成する．

3.2.5 FP2C におけるワークフローの開発および実行の流れ

FP2C におけるワークフローの開発および実行の流れをまとめる：

(1) アブストラクトコードを記述する
(2) XMP によりインプリメンテーションコードを記述する．これを用いて YML タスクジェネレータによりリモートプログラムを生成する
(3) YvetteML 言語によりワークフローを記述する．YML コンパイラを用いて，これを有向グラフに変換する
(4) タスクをマージしたリモートプログラムを生成する
(5) mpirun により YML スケジューラを実行する

4. 実験

4.1 実験環境

本実験では，T2K-Tsukuba システムを用いた．T2K-Tsukuba システムの概要を表 1 に示す．

4.2 テスト問題

ブロックガウスジョルダン法 (BGJ) により，逆行列を計算した．このアプリケーションは，入力された行列 A に関して，逆行列 B を返す．図 10 は，YvetteML で記述された BGJ のワークフローを示す．行列はいくつかのブロックに分割され，計算される．表 2 に BGJ におけるタスクのリストを示す．また，2 つの行列の掛け算を行う prodMat の XMP による記述例を，図 11 に示す．

行列の全体のサイズは，16384x16384 である．この行列を，32x32 個，16x16 個，8x8 個のブロックに分割する．各ブロックのサイズはそれぞれ，512x512，1024x1024，2048x2048 である．表 3 に各ブロックのサイズ，ブロック数，およびそれぞれのワークフローおけるタスク呼び出し回数をまとめる．

各タスクで用いられるプロセス数は，4, 16, 64 とした．タスクにおいて各ブロックは，プロセス数に応じて，2x2, 4x4, 8x8 にさらに分割される．ワークフロー全体で用いられる総プロセス数は，64, 128, 256 である．ただし，スケジューラのためにこれらとは別に 1 プロセスが確保される．ブロック分割数，総プロセス数およびタスク毎のプロセス数を変化させて，実行時間を調査した．

以下では，各問題をブロックサイズおよびブロック数を用いて $\mathrm{GJ}_{(sizeofblock)-(numberofblocks)}$ のように記す．

4.3 結果

図 12 は，“inversion”, “prodDiff”, “prodMat” がブロックサイズ 2048x2048 で実行された時のタスク毎の台数効果

表 1 T2K-Tsukuba
Table 1 T2K-Tsukuba

OS	Red Hat 4.1.2-14
CPU	Opteron “Barcelona” Quad-Core 8000 2.3GHz, 4 sockets (16 core) / node
Memory	DDR2 667MHz 32GB (8GB per socket)
Network	Inifiniband DDR (4 rails) 8GB/s
Compiler	xmpcc-0.5.3, gcc-4.1.2, mvapich2-1.4.1-shared
File System	Lustre cluster file system

表 2 BGJ におけるタスク
Table 2 Tasks in the BGJ

task(arg)	description
genMat(A)	generate a matrix
genMatUnit(A)	generate an unit matrix
fillMatrixZero(A)	generate a zero matrix
inversion(A,B)	$B = A^{-1}$
prodMat(B,A)	$A = A \times B$
mProdMat(B,A,C)	$C = -(B \times A)$
prodDiff(B,A,C)	$C = C - (B \times A)$

表 3 ブロックサイズ，ブロック数，タスク呼び出し回数
Table 3 The size of blocks, the number of blocks and the number of task calls

Name	block size	# of blocks	# of calls
GJ_{512-32}	512x512	32x32	34816
$GJ_{1024-16}$	1024x1024	16x16	4608
GJ_{2048-8}	2048x2048	8x8	640

を示す．図 13–15 は， 表 3 の各問題に対する実行時間を示す．図 16 は，GJ_{512-32}, $GJ_{1024-16}$, GJ_{2048-8} の実行タイムラインを示す．

図 13 は，34,816 のタスク呼び出しを行う GJ_{512-32} の実行結果である．この問題では，タスク呼び出し回数が非常に大きく，各タスクへの入出力データの処理などに時間がかかり，FP2C のスケーラビリティおよび性能は良くない．図 16 における上の図がこの問題に対するタイムラインを示している．この図からも，FP2C がタスクとタスクの間で時間を費やしていることがわかる．

図 14 は， $GJ_{1024-16}$ に対する結果を示す．スケーラビリティ，計算時間ともに他の問題よりも良好であり，特に，各タスクが 16 プロセスで実行されたときがもっとも良い結果を示した．一方で，1 タスクあたり 64 プロセスが用いられたとき，各タスクは内部の通信量の増加などから，図 12 に示されるように効率が低下し，全体の効率も低下した．

図 15 は，GJ_{2048-8} での結果を示す．性能は，$GJ_{1024-16}$ よりも低くなる．34,816 回のタスク呼び出しを行った GJ_{512-32} とは対照的に，この問題ではタスク呼び出しは 640 回しかない．タスク数が少ないぶん，ある時点で同時に実行可能なタスクも少なくなり，計算リソースがアイドル状態になることがあるため，効率が低下した．

これらの実験から，以下のように考えることができる：適切なプロセス数で実行される並列タスクは，逐次もしくは少ないプロセス数で実行されるタスクを多数呼び出すよりも，ワークフローの管理などのコストを軽減することができる．さらに，大規模並列タスクの代わりに，複数の並列タスクを用いることで，タスク内部の通信コストを削減することができる．

5. 関連研究

本章では関連研究について述べる．

Xcrypt [7] は，スクリプト言語であり，互いに依存関係のある並列ジョブの管理を平易に行うことができる．Xcrypt は大規模シミュレーションなどにおけるパラメータサーチに用いられる．並列ジョブを含むワークフローの定義から，Xcrypt はバッチスケジューラに対するジョブスクリプトを生成し，ジョブの実行や同期を管理する．FP2C と異なり，Xcrypt におけるタスクはバッチジョブであるため，Xcrypt では各タスクの実行毎に並列プログラムを起動する必要がある．一方，FP2C は，パラメータサーチなどの場合，新たに並列プログラムの起動することなく，すでに実行されているプログラムにパラメータを送信するのみで済む．また，バッチスケジューラなどを用いたシステムの場合，FP2C は 1 ワークフローを 1 バッチジョブとして投入できるため，Xcrypt と FP2C を組み合わせてより多階層のアプリケーションを実行することも可能である．この場合，Xcrypt における 1 タスクが FP2C における 1 ワークフローに相当する．

6. おわりに

本稿では， FP2C (Framework for Post-Petascale Computing) と呼ばれるプログラミング環境を提案し，開発した．FP2C は，分散メモリシステムに対する並列プログラミングを平易に可能にするプログラミング言語 XMP により記述された並列のタスクを，複雑な依存関係を持つタスクを管理することができるワークフロー開発実行環境 YML により統合し，多階層のプログラム実行を可能にする．

本研究では，YML ミドルウエアの一つである OmniRPC

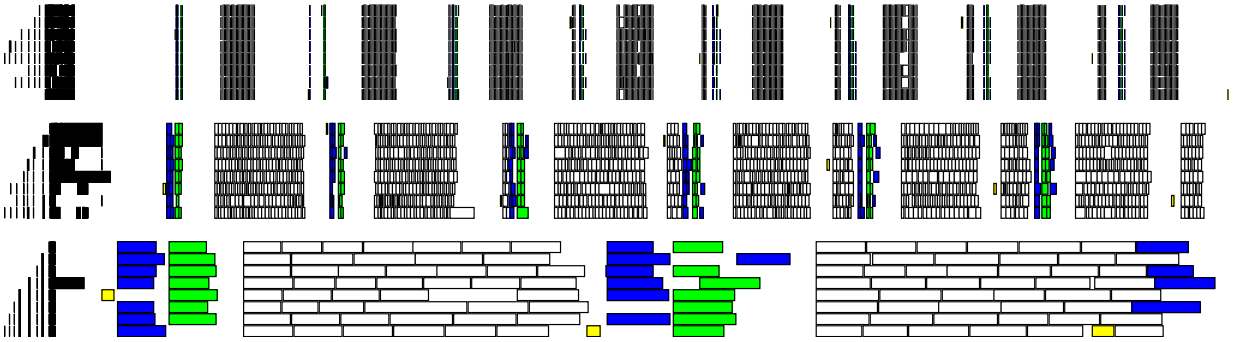

図 16 BGJ におけるタスクの実行タイムライン．上から GJ_{512-32}，$GJ_{1024-16}$ および GJ_{2048-8} のとき．全体では 128 プロセスが用いられ，各タスクは 16 プロセスで実行された．水平軸はタイムラインを，垂直軸は計算リソースを示す．ただし，タイムラインは 1000 秒までで切り捨て．ボックスはタスクを示す．genMat, fillMatrixZero, genMatUnit は黒，inversion は黄，prodDiff は白，prodMat は青，mProdMat は緑で示される．

Fig. 16 Timeline for task executions in BGJ. The top, middle and bottom figures are for GJ_{512-32}, $GJ_{1024-16}$ and GJ_{2048-8} respective. A whole workflow uses 128 processes. Each task uses 16 processes. The horizontal axis shows timeline (the left is zero and the right is truncated at 1000 sec). The vertical axis shows computational resources. The boxes show tasks. The black one shows genMat, fillMatrixZero and genMatUnit. The yellow one shows inversion, white one shows prodDiff, blue one shows ProdMat, green one shows mProdMat.

の MPI に向けた拡張と拡張された OmniRPC を使用するための MPI バックエンドの開発，XMP により記述されたタスクの生成のサポート，YML コンパイラおよびスケジューラの拡張を行った．

実験により性能を評価し，FP2C が巨大で階層的なシステムの効率的な利用を可能にすることを明かにした．

今後の課題としては，失敗したタスクの再実行による耐故障性の向上などが挙げられる．

Acknowledgment

Numerical calculations for the present work have been carried out under the “Interdisciplinary Computational Science Program” in Center for Computational Sciences, University of Tsukuba.

参考文献

[1] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Nèri, and O. Lodygensky. Computing on large-scale distributed systems: Xtrem web architecture, programming models, security, tests and convergence with grid. *Future Generation Computer Systems - Special issue: P2P computing and interaction with grids*, 21(3):417–437, 2005.

[2] O. Delannoy. *YML: A scientific Workflow for High Performance Computing*. PhD thesis, University of Versailles Saint-Quentin, 2006.

[3] O. Delannoy and S. Petiton. A peer to peer computing framework: Design and performance evaluation of yml. In *Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 362–369, 2004.

[4] J. Lee and M. Sato. Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. In *39th Annual International Conference on Parallel Processing*, pages 413–420, 2010.

[5] M. Sato, M. Hirano, Y. Tanaka, and S. Sekiguchi. Omnirpc: A grid rpc facility for cluster and global computing in openmp. In *International Workshop on OpenMP Applications and Tools, 2001*, pages 130–136, 2001.

[6] XcalableMP Specification Working Group. http://www.xcalablemp.org/.

[7] Xcrypt. http://super.para.media.kyoto-u.ac.jp/xcrypt/.

```
par
  A[i][j] is initialized at random
  B[i][j] is initialized as an unit matrix
endpar

par
  par(k:=0;count-1)
  do
    if (k neq 0) then
      wait(prodDiffA[k][k][k-1]);
    endif
    compute inversion(A[k][k],B[k][k]);
    notify(bInversed[k][k]);
    if (k neq count-1) then
      par (i:=k+1; count-1)
      do
        wait(bInversed[k][k]);
        compute prodMat(B[k][k],A[k][i]);
        notify(prodA[k][i]);
      enddo
    endif
    wait(bInversed[k][k]);
    par(i:=0;count-1)
    do
      if(i neq k) then
        compute mProdMat(A[i][k],B[k][k],B[i][k]);
        notify(mProdB[k][i][k]);
      endif
      if(k gt i) then
        compute prodMat(B[k][k],B[k][i]);
        notify(prodB[k][i]);
      endif
    enddo
    par(i:= 0;count-1)
    do
      if (i neq k) then
        if (k neq count - 1) then
          par (j:=k + 1;count-1)
          do
             wait(prodA[k][j]);
             compute prodDiff(A[i][k],A[k][j],A[i][j]);
             notify(prodDiffA[i][j][k]);
          enddo
        endif
        if (k neq 0) then
          par(j:=0;k-1)
          do
            wait(prodB[k][j]);
            compute prodDiff(A[i][k],B[k][j],B[i][j]);
          enddo
        endif
      endif
    enddo;enddo;endpar
```

図 10 YvetteML により記述されたブロックガウスジョルダン法
Fig. 10 A workflow of Block Gauss Jordan Method written in YvetteML

```
<?xml version="1.0"?>
<component type="impl" name="prodMat" abstract="prodMat">
 <impl lang="XMP" nodes="CPU:(2,2)">
  <template>
    <template name="t" size="1024,1024" format="block,block">
  </template>
  <distribute>
    <param name="A0" template="t" size="1024,1024" />
    <param name="B0" template="t" size="1024,1024" />
  </distribute>
<header><![CDATA[
double A[1024][1024];
#pragma xmp align A[*][j] with t(j,*)
double B[1024][1024];
#pragma xmp align B[i][*] with t(*,i)
]]></header>
<source><![CDATA[
.....
#pragma xmp loop (j,i) on t(j,i)
  for(i=0;i<1024;i++){
    for(j=0;j<1024;j++){
      A[i][j]=A0[i][j];
      B[i][j]=B0[i][j];
    }}
#pragma xmp reduction(+:A) on t(*,:)

#pragma xmp reduction(+:B) on t(:,*)

#pragma xmp loop (j,i) on t(j,i)
  for(i=0;i<1024;i++){
    for(j=0;j<1024;j++){
      x=0.0;
      for(k=0;k<1024;k++)
        x+= (B[i][k]*A[k][j]) ;
      A0[i][j]=x;
    }}
]]></source>
</impl>
</component>
```

図 11 XMP により記述された ProdMat. $A0 := B0 \times A0$
Fig. 11 ProdMat $A0 := B0 \times A0$ written in XMP

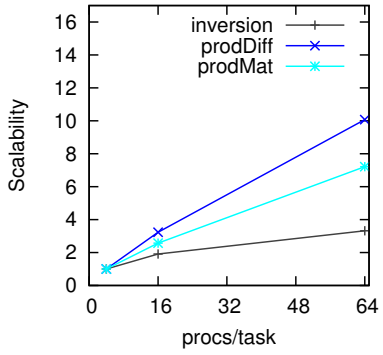

図 12 タスクの台数効果. 1 タスクあたりのプロセス数 4 のときを基準とした
Fig. 12 Scalability of tasks. A execution time of each task is normalized by the case that a task uses 4 core.

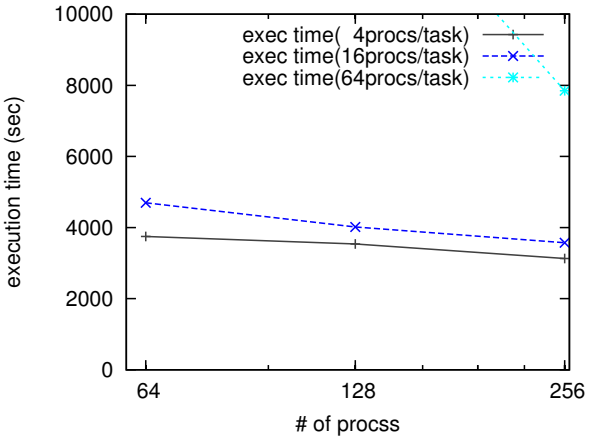

図 13 GJ_{512-32}, 32x32 blocks (block size is 512x512)
Fig. 13 GJ_{512-32}, 32x32 blocks (block size is 512x512)

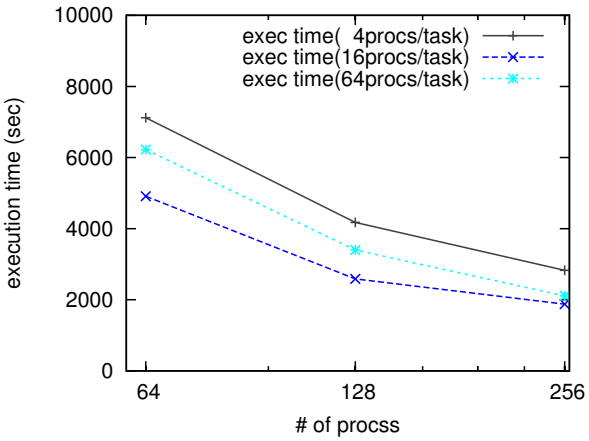

図 14 $GJ_{1024-16}$, 16x16 blocks (block size is 1024x1024)
Fig. 14 $GJ_{1024-16}$, 16x16 blocks (block size is 1024x1024)

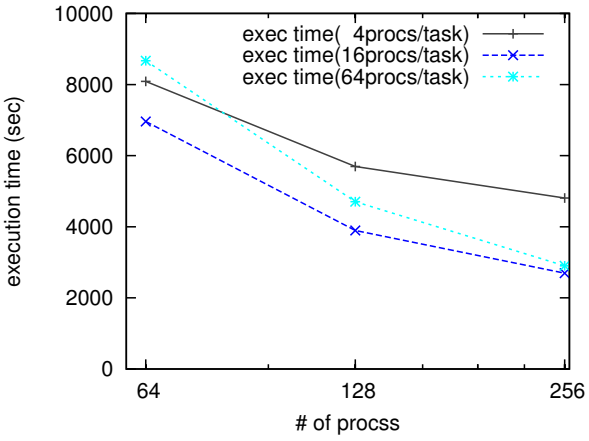

図 15 GJ_{2048-8}, 8x8 blocks (block size is 2048x2048)
Fig. 15 GJ_{2048-8}, 8x8 blocks (block size is 2048x2048)