社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2003－HPC－95 (20)
2003／8／5

タスク並列スクリプト言語MegaScriptによるタスク動作モデル記述

大塚保紀[†] 大野和彦[†,☆] 中島浩[†]

我々は，メガスケールコンピューティング向けの並列プログラミング言語としてMegaScriptを提案している．MegaScriptは，SPMDのような一般的な並列プログラムや逐次プログラムをタスクとして扱い，複数のタスクを並列に実行する．また、階層化されたライブラリを提供し，エンドユーザからヘビーユーザまで幅広く対応する．さらに，タスクの抽象化したふるまいを表現するためにメタプログラムを用いる．メタプログラムは，コンパイラの解析情報や実行プロファイルと組合せスケジューリングに用いられる．メタプログラムは実行されるとそのタスクのコスト式を出力する．その式をもとに動作モデルが構築され，スケジューラは動作モデルをもとに粒度や配置，タスクを実行する順序/タイミングを決定する．

本論文では，メタプログラムについて説明し，メタプログラムから動作モデルを生成する過程について述べる．

Description Behavior Model using a Task Parallel Script Language MegaScript

YASUNORI OHTSUKA,[†] KAZUHIKO OHNO[†,☆] and HIROSHI NAKASHIMA[†]

We proposese a task-parallel script language named MegaScript for megascale computing. A MegaScript program has a tow-tier parallelism; the lower is an ordinary parallelism such as SPMD and the upper is task-level parallelism in which thousands of lower level parallel tasks are involved. The MegaScript provides a hierarchical function library for lower level parallel task management, which is easy enough to write for end users and also enough detailed for heavy users. Moreover, the MegaScript has a special feature for meta-programming to represent an abstracted behavior of parallel tasks. This meta-program, combined with compile-time analysis result and execution profile, is used for optimal runtime task scheduling. The meta-program generates a expression which represents execution cost of the task. The behavior model is constructed base on it, and the scheduler determines the granularity, allocation and ordering/timing of the execution referring the model and refines the model with the profile for future use.

In this paper, we describe the meta-program and process of constructing behavior model.

1. はじめに

近年，数十Tflopsという計算能力を有する数千台規模の大規模計算機が登場している．しかし，複雑な物理系が絡み合う環境・気象シミュレーションや災害シミュレーションなどではPflops以上の計算能力が求められており，Pflops以上の性能を得るためには，100万台規模のプロセッサを用いたメガスケールコンピューティングが必要である．

しかし，従来のような専用並列計算機をメガスケール規模で運用するには，膨大なハードウェアを設置する巨大な施設や電力を必要とする．このため我々は，コモディティな技術を用いた「低電力化とモデリング技術によるメガスケールコンピューティング」の研究を行っている．メガスケールコンピューティング環境は，性能の異なる計算機やネットワークの集合としてシステムを構築するため，それらの資源を効率よく利用する実行システムが必要である．そこで，我々はメガスケールコンピューティング向けの言語としてタスク並列スクリプト言語MegaScript[1)]の開発を行なっている．

メガスケールの並列性を持つプログラムを一から記述するのは非常に困難である．その解決策として，逐次プログラムや部分問題を並列化した小規模な並列プロ

† 豊橋技術科学大学
Toyohashi University of Technology
☆ 現在，三重大学
Presently with Mie University

グラムを組み合わせることにより大規模な並列性を引き出す「多重並列モデル」が考えられる．MegaScriptでは，独立性の高い外部プログラムをタスクとして利用し，並列に実行することが可能である．

また，メガスケールの並列性をもつアプリケーションをGridと同じようにノードやネットワークの性能が非均質なメガスケール環境上で実行するためには，アプリケーションの特性と計算機資源の特性を考慮して制御するスケジューラが必要となる．同様な性質を持つGridコンピューティングシステムにおけるタスクスケジューリングについては，多くの研究が行なわれており，ユーザから提供されるプログラムの詳細な情報とサーバやネットワークの動的な予測情報をもとにスケジューリングを行なうAppLeS(Application-Level Scheduling)[2)3)]や，自動並列化コンパイラにより得られる予測処理時間と計算資源に関する動的な負荷情報を用いたメタスケジューリング[4)]などがある．

MegaScriptのスケジューラは，タスクの特性を把握するためにコンパイラによる解析情報や過去に実行された履歴，ユーザより提供される情報などを用いる．また，実行時に取得できる動的な情報(タスクの実行時間，動的なプロファイリング情報，実行環境の状態など)をもとに，スケジューリングの修正を行なう．

また，ユーザからの情報提供の枠組としてメタプログラムを提案している．メタプログラムは，タスクのふるまいを抽象化されたプログラムで記述したのもであり，メタプログラムを実行することでタスクのコストを得ることができる．本論文では，MegaScriptによるメタプログラムの記述と動作モデルの生成について記述する．

2. MegaScriptの概要

MegaScriptは各タスク間の関係のみ☆を記述するため，コードの処理コストは小さい．このためプログラムの記述性を優先しスクリプト言語を採用した．MegaScriptはベース言語にRuby[5)]を用い，Rubyを拡張する形で設計・実装する．

2.1 タ ス ク

MegaScriptの並列実行単位はタスクであり，クラスとして定義されオブジェクトで扱われる．実際の計算部分は逐次又は並列の外部プログラムとして用意する．

タスクの定義方法は，MegaScirptで定義されているTaskクラスを継承しinitializeメソッドとbehaviorメソッドを定義する形で行う．タスク定義の例を図1にあげる．

☆ タスク間のネットワーク構造以外に実行時引数などがある

```
class Sample < Task
  def initialize(*arg)
    @exefile = './sample'
    @parameter = arg
  end
  def behavior
    # メタプログラムを記述
  end
end
```

図1 タスク定義の例
Fig. 1 Example of task definition

2.1.1 initializeメソッド

initializeメソッドには，以下のようなタスクを実行するために必要な情報を記述する．

- `@exefile` ：実行ファイル
- `@parameter`：実行時引数
- `@usefile` ：入出力ファイル

これらのインスタンス変数は，オブジェクトの内部情報として保持しタスクを実行する際に参照する．

2.1.2 behaviorメソッド

behaviorメソッド内には，メタプログラムを記述する．メタプログラムの記述については3章で詳しく説明する．

2.2 ストリーム

MegaScriptでは，タスクの多くが独立した外部プログラムであることを想定しており，この場合，MegaScriptから確認できるのは，標準入出力と実行時引数，エラー出力，プロセス終了時の返り値である．また，ライブラリを差し替えることで特定の動作を検出する方法も考えられるが，タスクの記述言語を固定していないため非常に困難である．

そこでMegaScriptでは，各タスクの標準入出力を接続する方法でタスク間通信を実現する．我々が既に開発した(Perl)+[6)]では，ファイルと同様に入出力行える通信ストリームを導入しており，この技術を応用し標準入出力を用いた通信を行う．

MegaScriptでは，タスクを論理的に接続する通信路をストリームと呼ぶ．ストリームはタスク同様にオブジェクトとして扱われる．またストリームの入出力端に複数のタスクを接続することができ多対多通信が可能である．

ストリームを流れるメッセージは，改行コードで区切られた文字列である．入力側に複数のタスクが接続されている場合，メッセージは行単位でマージされる．また，出力側に複数のタスクが接続されている場合，

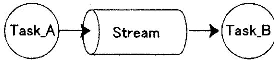

図 **2**　ストリームの接続図
Fig. 2　Stream connection

```
taska = Task_A.new()
taskb = Task_B.new()
strm = Stream.new()
strm.connect(taska, IN)
strm.connect(taskb, OUT)
```

図 **3**　ストリーム接続の例
Fig. 3　Sample program of stream connection

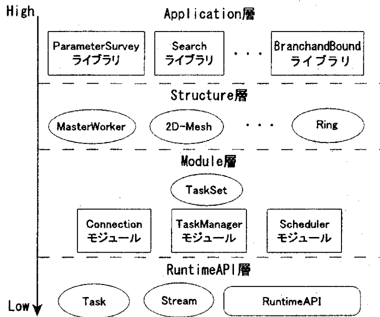

図 **4**　ライブラリの階層構造
Fig. 4　Hierarchical function library

メッセージはマルチキャストされる．

2.3　プログラミング

MegaScript では，使用するタスクのオブジェクトを生成し，それらをストリームオブジェクトに接続する．図**2**のように Task_A と Task_B を接続する場合のプログラムは，図**3**のようになる．

ストリームを用いることで自由にタスクを接続することが可能であるが，学術計算などを目的とし並列処理の知識に乏しいエンドユーザには困難な作業である．そこで，タスクネットワークの構築を支援する図**4**のような階層化されたライブラリ構造を用意し，ヘビーユーザからライトユーザまで対応する．

2.4　処　理　系

MegaScript の処理系は，トランスレータ，ランタイム，スケジューラから構成される (図**5**)．

トランスレータは，起動直後にメタプログラムを実行し，スケジューラが参照・更新可能な動作モデルの生成を行なう．

ランタイムは，基本的な並列実行機構を提供するものであり，分散環境上でタスクの配置やタスク間通信を実現する．また，実行環境の状態やタスクの実行時

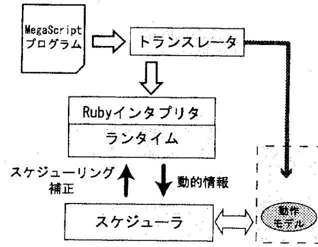

図 **5**　処理系の概要図
Fig. 5　Structure of execution system

間といった動的な情報の収集も行なう．

スケジューラの動作は，静的スケジューリングと動的補正の 2 つに分かれる．

まず，スケジューラは動作モデルから得られる予想コストと実行環境の情報をもとにスケジューリングを行いタスクの初期配置を決定する．各タスクが実行されるとランタイムからフィードバックされる情報をもとに，動作モデルの検証・修正／再構築を行ない実行モデルの精緻化を行なう．そして，このモデルをもとに現在のスケジューリングを評価し補正を行なう．

全てのタスクが終了すると，現在の情報を履歴として保存し次回実行時に利用する．

3.　メタプログラム

ユーザが情報を記述する方法の 1 つとして，タスクの計算量や通信量，必要とするリソースの情報などを直接記述する方法が考えられる．しかし，これらの情報だけではタスクの概要を知ることはできるが，通信頻度やタイミング，入力値による動作の変化といったふるまいを知ることができない．またユーザ自身がプログラムを解析しコストを見積もらなければならず，この作業はエンドユーザには極めて困難である．

MegaScript ではタスクに関する情報の記述方法としてプログラムを用い，ユーザはタスクのふるまいをメタプログラムで記述する．メタプログラムからプログラムの全体構造を把握することが可能であり，計算量や通信のタイミング，総通信量などタスクの特性を知ることができ，実行時の引数に対する動作の変化を予測することもできる．

メタプログラムは，記述方法としてプログラムを用いているため極めて高い記述性があり，詳細な記述が可能である．システム側としては，可能な限り詳細に(プログラムとして実際に実行可能なレベルで) 記述されているほうが望ましいが，それはユーザに相当な労力を要する．また，スクリプト言語に不慣れな場合は更に多くの時間を費すことになる．そこで，いくつか

の抽象化表現を用意する．抽象化表現を用いることでトップレベルの構造だけを簡略化して記述することも可能である．

3.1 抽象化表現

MegaScript でのメタプログラムの記述について述べる．ベース言語である Ruby と以下で説明する抽象化表現を用いて記述する．抽象化表現する項目として以下のものがある．

- 変数・数値
- 計算処理
- 制御構文
- 関数

次に各事項の抽象化について説明する．

3.1.1 変数・数値

前述のように，詳細なメタプログラムをユーザに記述してもらうのは，困難である．しかし，プログラムの一部又は大部分の抽象化を行なうと実行時の変数値を決定する式も省略される可能性がある．そこで，実行時に値が決定しない変数を扱える枠組としてコストオブジェクトを提供する．コストオブジェクトは内部情報として数式をもつオブジェクトである．

```
l = 10
m = n ** 2 + 1
```

と記述されている場合，変数 l は "10" という数値リテラルだが，変数 m は未定義の変数 n を含んでいる．この場合，Ruby ではエラーとなるが，メタプログラム中では，変数 m は "$n^2 + 30$" という数式をもつコストオブジェクトと扱われる．また，変数 n のような未定義な変数やタスクが生成される段階まで値が決定しない実行時引数などは全てコストオブジェクトとして扱われる．また，コストオブジェクトはタスクの実行時に動的な情報をもとにスケジューラが補完する．

コストオブジェクトは，未決定な要素を含んでいるため比較演算を行なうことは行なえず，コストオブジェクトを条件式等に記述することはきでない．コストオブジェクトがループ回数や分岐に影響を与える変数の場合，3.1.3 節に示す抽象化した制御構文を用いる．

3.1.2 計算処理

メタプログラム内でまとまった計算処理を表現するために，compute() という関数を用意する．compute() 関数は次のように定義される．

- compute(cost)
  その場所で，大きさ cost だけの計算処理が行なわれる．cost はコストオブジェクトであり，数式又は数値を引数にとる．具体的な計算量が分からない場合は，引数 cost を省略してもよい．この場合，実行時に未定義変数が動的な情報から補完される．

3.1.3 制御構文

制御構文の条件式を正しく評価するためには変数値が正しく決定する必要があるが，抽象化を行なうと条件式の判断も行なえない．そこで抽象化した制御構文を用意する．ループ構文と分岐構文を例にあげて説明する．

- 抽象化ループ FOR
  Ruby の for は，本来，式を評価した結果のオブジェクトの各要素に対して本体を繰り返し実行するものである．MegaScript の提供する抽象化ループ FOR では，ループ回数 loop_count を直接記述する．以下のようなに記述する．

  FOR *loop_count*
    loop_body
  END

  FOR は，本体 loop_body のコストに loop_count を乗算したコストを求める．式で表すと，

  $Cost(loop_body) * loop_count$

  となる．

- 抽象化分岐 IF
  抽象化分岐は，分岐条件式を記述するのではなく，分岐確率 *prob* を直接記述してもらう．抽象化分岐 IF は以下のように記述する．

  IF *prob*
    if_body
  ELSE
    else_body
  END

  IF の場合は，if_body と else_body に分岐確率を乗算したコストを求める．If の場合，

  $Cost(if_body) * if_prob$
  $Cost(else_body) * else_prob$

  の二つのコストを求める．

compute() と同様に，IF や FOR においてもループ回数や分岐確率が不明な場合省略することが可能である．この場合も，自動的に未定義変数に置き換えれる．

3.1.4 関　数

関数にも上記と同様のことが言えるため，メタプログラムでは抽象化関数を定義する DEF を提供する．抽象化関数は次のように定義される．

- 抽象化関数 DEF *func_name(arg1,arg2,...)*
  抽象化関数は DEF ～ END で定義する．引数 *arg1, arg2, ...* をとり，関数内部 func_body のコストを返す．

```
def behavior
  DEF init(n)
    compute(n)
  END

  n = @parameter
  init(n)
  FOR n
    IF 0.8
      for i in 1..10
        compute(i)
      end
    ElSE
      compute(1)
    END
  END
end
```

図 6 メタプログラムの例
Fig. 6 Example of meta-program

DEF *func_name* (*arg1,arg2,...*)
func_body
END

これらの抽象化表現を用いて，メタプログラムを記述する．図 **2.1.2** 節で述べたように，メタプログラムは behavior メソッド内に記述し，図 **6** の例のようになる．

3.2 メタプログラムの実行

図 6 の例で説明する．まず，変数 n は実行時引数を表すクラス変数@parameter を参照しており，この値はタスク実行時まで決定されない．よって，変数 n はコストオブジェクトとして生成される．抽象化関数 init() からは，コスト式 "n" が得られる．FOR n ～ END は，ループ本体のコスト式にループ回数 "n" を乗算するため，先に loop_body を評価する．同様に IF 0.8 ～ ELSE ～ END も本体の評価を先に行う．if_body 側が評価されるとコスト "55" が得られる．同様に，else_body からはコスト "1" が得られる．この 2 つのコストにそれぞれの分岐確率を掛けた "44" と "0.2" が IF の持つコストになる．次に，FOR の loop_body が評価を行い，loop_body のコスト'44' と'0.2' にループ回数 n を掛けた "$44 * n$" と "$0.2 * n$" の 2 つのコスト式を得る．これより，このタスクのコストは

$$Cost = n + 44 * n + 0.2 * n$$

という数式で得らる．タスクの引数が "10" の場合のコストは "452"，引数が "20" の場合は "904" となる．

4. モデリング

タスクの動作モデルの生成過程について説明する．

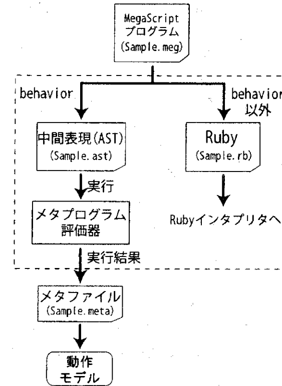

図 7 トランスレータでの処理の流れ
Fig. 7 Flow of processing in translator

記述されたメタプログラムは，Ruby インタプリタでは実行されずにトランスレータで処理される．トランスレータでは図 **7** ような処理が行なわれる．まず，MegaScript プログラムをパースし，メタプログラムが記述されている behavior メソッド部分を取り出す．残った部分は Ruby インタプリタで実行可能なプログラムなので，'.rb' ファイルとしてインタプリタへ渡す．一方，メタプログラムは AST(Abstract Syntax Tree) で構成される中間コードに変換される．生成された中間コードは，トランスレータ内のメタプログラム評価器で 3.2 節のように実行され，その結果は'.meta' ファイルとして出力される．

スケジューラは，この'.meta' ファイルを参照し動作モデルを構築する．動作モデルは，重み付きグラフでモデル化される．抽象化関数毎にグラフ化するため，動作モデルは 1 つ以上のグラフを持つことになる．グラフは，3 つのノード（コストノード，分岐ノード，関数ノード）から構成される．コストノードは，各ブロックのコスト式を持つノードである．分岐ノードは，分岐の開始と終了を表し複数のエッジを持つ．関数ノードは，その時点で関数呼び出しがあった事を表す，関数名と引数を持ち，直接コストを持たずに各関数のグラフからコストを得る．

図 6 のサンプルプログラムから得られる動作モデルは，図 **8** のようになる．

5. おわりに

本論文では，MegaScript の概要と MegaScript によるメタプログラムの記述方法，そしてメタプログラムからタスクの動作モデルの生成方法について述べた．今後は，トランスレータの実装を行ない HPC 分

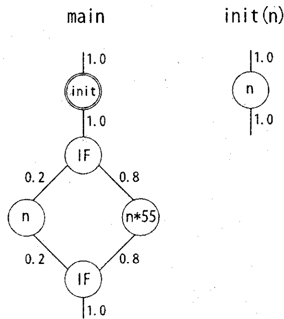

図8 サンプルプログラムから得られる動作モデル
Fig. 8 Behavior model constructed from sample-program

野で用いられるアプリケーションをモデリングし評価を行なう．また，今後の課題として以下のようなものがある．

- 通信コスト
  現段階のメタプログラムでは計算コストに注目しモデリングを行なっている．しかし，Message-intensiveなアプリケーションなどの場合には，計算コストより通信コストの方が重要であり，通信コストを含んだ動作モデルが必要である．
- 並列プログラムの記述
  2章でも述べたように，MegaScripではタスクに並列プログラムを用いることを前提としている．このため，メタプログラムも並列プログラムを表現する機能が必要である．
  また，並列アプリケーションは，スケジューリング方法や割り当てるリソースによって大きく性能が変化する．MegaScriptは，外部プログラムであることからタスク内部の制御に干渉するのは困難である．このため，スケジューリング方針を簡単に記述できる枠組や，「ある構成の環境に割り当てた場合のコスト値」といった環境の特性も考慮する必要があると考えられる．
- 動的プロファイル情報による精緻化
  MegaScriptでは，メタプログラムから得られた動作モデルに対し，実行時や実行終了時に得られるプロファイル情報を反映し動作モデルの精緻化を行なう．反映させる情報として，変数の値や分岐確率・実際のループ回転数，ブロック毎に実行時間，メッセージ量などがあり，これらの情報からノードの追加／削除やノードのコスト式の修正を行なう．
  この機能を実現するため，メタプログラム中にプロファイル取得の指示文を記述し，それをタスクのソースコードに反映させる枠組を検討している．

6. 関連研究

アプリケーションやタスクの特性や性能を予測するためにモデリングを行ない，予測モデルをもとに負荷分散や資源割り当てを行なっている研究の一つにAppLeSスケジューラプロジェクトがある．

AppLeSでは，分散環境上でのアプリケーションの性能予測を行なうために構造モデル[7]を用いている．構造モデルは，計算や通信といったコンポーネントモデルと各コンポーネントモデルの相互関係を示すオペレーションから構成され，数式で表される．また，1つのコンポーネントについて複数の実装を持たせることができ，モデルの予測値が必要な精度に達しない場合，別の実装のコンポーネントを選択する柔軟性がある．AppLeSの構造モデルの場合，問題サイズとリソースセットを渡すことで予測実行時間を知ることができ，「単位のない大きさ」を求める本研究の動作モデルとは異なる．しかし，各コンポーネントを構成する数式を直接記述する必要があり，これはアプリケーションの開発者以外には困難な作業である．

謝辞 本研究は，科学技術振興事業団・戦略的基礎研究「低電力化とモデリング技術によるメガスケールコンピューティング」による．

参考文献

1) 大塚保紀, 深野佑公, 西里一史, 大野和彦, 中島浩: タスク並列スクリプト言語MegaScriptの構想, 先進的計算基盤システムシンポジウムSACSIS2003, pp. 73–76 (2003).
2) Berman, F. and Wolski, R.: Scheduling from the perspective of the application, *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing* (1996).
3) Berman, F., Wolski, R., Figueira, S., Schopf, J. and Shao, G.: Application-level scheduling on distributed heterogeneous networks, *Proceedings of SuperComputing '96* (1996).
4) 笠原博徳, 小幡元樹, 石坂一久: メタスケジューリング-自動並列分散処理の試み-, *bit*, Vol. 33, No. 4, pp. 36–41 (2001).
5) まつもとゆきひろ, 石塚圭樹: オブジェクト指向スクリプト言語Ruby, ASCII (1999).
6) 外崎由里子, 中田尚, 大野和彦, 中島浩: 並列スクリプト言語(Perl)+の実装と設計, 並列処理シンポジウムJSPP'02, pp. 241–244 (2002).
7) Schopf, J.: Structural Prediction Models for High-Performance Distributed Applications, *Cluster Computing Conference '97* (1997).