

タスク並列スクリプト言語 MegaScript による 実行時情報プロファイリング

湯 山 紘 史[†] 大 塚 保 紀^{††} 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、逐次または並列の外部プログラムである多数のタスクを並列実行するためのスクリプト言語であり、効率的なスケジューリングのためにユーザがタスク特性をプログラムの形で抽象化して表現するメタプログラム記述を特徴としている。このメタプログラム記述により得られるタスク性能モデルの精度は、逐次あるいは負荷が均一なプログラムに対しては高いものの、負荷が不均一なプログラムに対しては問題があることが明らかになっている。そこで本報告では、MegaScript へのタスクのプロファイル情報を取得する枠組の導入と、それに基づくタスク性能モデルの性能向上について提案する。

A Profiling Method for a Task Parallel Script Language: MegaScript

HIROSHI YUYAMA,[†] YASUNORI OTSUKA^{††} and HIROSHI NAKASHIMA[†]

We are pursuing research works on our task-parallel script language named MegaScript for mega-scale computing. A unique feature of MegaScript is the capability to describe abstracted behavior of tasks in a program form named *meta-program*. The model derived from the meta-program is sufficiently accurate for sequential or well-balanced parallel programs. However, our experience showed a large modeling error for ill-balanced parallel programs. This paper proposes a solution for the accuracy problem by introducing a profiling scheme into MegaScript meta-programs.

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な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ための巨大な施設や膨大な電力が、必要となる。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。このような環境で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環境が必要であり、そのための手段としてオブジェクト指向スクリプト言語 Ruby²⁾ をベースとしたタスク並列スクリプト言語 MegaScript の開発を行っている。

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。そこで MegaScript には、ユーザがタスクの特性を抽象化されたスクリプトプログラムで表現するメタプログラム記述の枠組が用意されている。

このメタプログラム記述に基づき生成されるタスク性能モデルの精度は、逐次または負荷が均一な並列プログラムに対しては高いものの、負荷が不均一なプログラムについてはその特性を十分に反映できていないという問題がある。そこで我々は精度の向上手段として、タスクのプロファイル情報を取得する枠組を提案する。またメタプログラム中の指示に基づきプロファイルコードをタスクプログラムに埋め込み、取得情報を性能モデルへ反映する機構の実装についても述べる。

[†] 豊橋技術科学大学
Toyohashi University of Technology

^{††} 三菱電機 (株)
Mitsubishi Electric Corp.

```

class Mandel < Task
  def initialize(*arg)
    @exefile = "/mandel"
    @args = arg
  end
end

def behavior
  FOR 0 .. @args[0]
    compute(2)
    IF 0.5
      compute(1)
    END
  FOR 1 .. 16
    msgrecv(10)
  END
  IF ID!=0
    msgsend(10)
  end
end

```

図 1 タスク定義の例

2. タスク並列スクリプト言語 MegaScript

本章では、ユーザの記述部分であるメタプログラムと、それを処理するための処理系を含めた MegaScript の構成について述べる。

2.1 タスク

タスクは MegaScript の並列実行の単位であり、逐次または並列の外部プログラムとして与えられる。タスクを定義するクラスは図 1 のように記述され、initialize メソッドによりタスクの実行ファイルなどが記述される。個々のタスクはインスタンスオブジェクトとして生成され、スケジューラによって適切な計算ノードに配置される。

タスク間通信は、標準入出力を用いたストリーム通信により実現する。ストリームの入出力端には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。入力側に複数のタスクが接続されている場合、各タスクの出力が非決定的に行単位でマージされ、出力側に複数のタスクが接続されている場合、メッセージ出力側に接続されている全てのタスクにマルチキャストされる。また、ストリームはタスク同様にスクリプト内ではオブジェクトとして扱われている。

2.2 メタプログラム

ユーザによるタスク特性の記述方法として、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ができるが、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。

そこで MegaScript では、タスクに関する情報の記述方式としてメタプログラムと呼ぶスクリプトプログラムを用いる。スケジューラなどは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でき、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る。また、実行時の引数に対する特性の変化を予測することもできる。

メタプログラムは図 1 の behavior メソッドとして与えられ、計算・通信のコストや制御構造を表す抽象化表現を用いて記述される。メタプログラムは、記述方式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、柔軟な記述が可能である。システム側（スケジューラ）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）記述されていることが望ましいが、それはユーザへの負担が大きい。また、Ruby に不慣れな場合はさらに多くの時間を費やすことになる。そこで以下に述べるような抽象化表現を用意し、これらを用いてユーザの知識レベルの範囲で（例えばトップレベルの構造だけ）記述できるように設計されている。

コストオブジェクト

計算・通信のコストやループ回数などは一般の算術式で与えられるが、この式にはタスクの引数のように実行時まで値が確定しない変数を含むことができる。このようにメタプログラムを解釈して性能モデルを生成する時点では必ずしも評価できない算術式はコストオブジェクトとして内部表現され、値が未知の変数についてはタスクを実行する時点で代入を行った上で式が評価される。またコストオブジェクトには単位がなく、ユーザが考えやすい単位で表現することができる。

計算処理

メタプログラム中では、タスクプログラムが行う具体的な計算は省略する。代わりにどの地点でどの程度の計算が行われるかを記述する。

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() というメタ関数を用意する。compute(cost) という書式で使い、cost で指定されるコストオブジェクトの大きさの計算処理が行われることを表す。

標準入出力

MegaScript のタスク間通信は、標準入出力に接続されたストリーム通信によって行われる。スケジューラにとってストリームに流れるメッセージ量の予測・推測は重要であり、タスク間の結合度などを知ることができる。このため抽象化表現でも標準入出力を表す必要がある。

メタプログラムで入出力を表現するために input() 関数と output() 関数を用意する。input(cost) / output(cost) という書式で使い、cost で指定されるコストオブジェクトの大きさのデータ入出力が行われることを表す。

制御構文

タスクプログラムの性能モデルの構築にはループや条件文などの制御構文の動的挙動を知ることが重要である。しかし一般に制御構文の実行条件はタスクの実

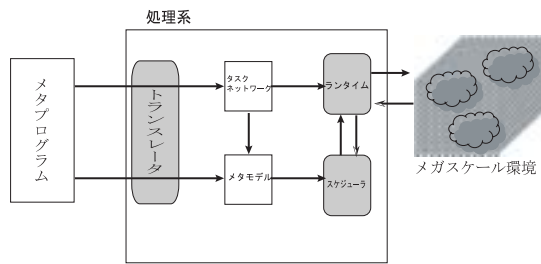


図 2 処理系の概要図

行時に定まるため、Ruby の制御構文を用いることはできない。そこで抽象化された制御構文として、ループを表現する FOR、条件文を表現する IF、および並列化構文を表現する PARALLEL が用意されている。FOR にはループ回転数を表現するコストオブジェクトが、IF には分岐確率が、また PARALLEL には並列度を表現するコストオブジェクトが、それぞれ与えられる。

メッセージ通信

1つのタスクが並列プログラムである場合、その内部でメッセージ通信を表現するために `msgsend()` 関数と `msgrecv()` 関数を用意する。`msgsend(cost, dst)` / `msgrecv(cost, src)` という書式で使用する。`cost` で指定されるコストオブジェクトの大きさのメッセージが `dst` へあるいは `src` から通信されることを表す。

2.3 処理系

MegaScript の処理系の概要を図 2 に示す。処理系は、トランスレータ、ランタイム、スケジューラからなる。

2.3.1 トランスレータ

MegaScript のフロントエンドであるトランスレータは、ユーザが記述した MegaScript プログラムを解析し、メタプログラムである behavior メソッドを抽出して解釈実行する。すなわちメタプログラム中のオブジェクトレベルの記述 (Ruby の制御構文や、式、代入文など) はそのまま実行し、抽象化表現については以下に示すコストなどを表現するコストモデルに変換する。

- 計算コスト
- ストリーム通信 (入出力) コスト / 回数
- 内部メッセージ通信コスト / 回数
- 並列性

基本的には、各コストは各メタ関数 (`compute()`, `input()/output()`, `msgsend()/msgrecv()`) の引数のコストの総和、通信回数はメタ関数の (抽象的な) 実行回数である。

タスク全体の性能モデルであるメタモデルは、上記のコストモデルを組み合わせた一種の制御フローグラフとして生成される。メタモデルはコストノードと制御ノードと呼ぶノードからなる DAG として表現される。コストノードは各ブロックのコスト式をもつノー

ドであり、各メタ関数はコストノードに置き換えられる。制御ノードは、内部に 1 つ以上のノードを持つ階層化されたノードであり、FOR や IF や PARALLEL がこれに置き換えられる。

なお behavior メソッドを除く部分は Ruby プログラムとして実行可能であり、そのままランタイムに渡されて実行される。

2.3.2 スケジューラ

スケジューラは、静的スケジューリングと動的補正の 2 つの動作を行う。まず、スケジューラはメタモデルから得られる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、メタモデルの検証・修正 / 再構築を行ないメタモデルの精緻化を行なう。そして、このモデルをもとにスケジューリングの良否を評価し補正を行なう。

2.3.3 ランタイム

ランタイムはマスタとスレーブで構成される。マスタは Ruby インタプリタを呼出し、タスク、ストリーム、スケジューラを生成する。スレーブはリモートノード上に生成され、そのノードに配置されるタスクやストリームの管理を行う。またランタイムは、メガスケール環境で実行されたデータのスケジューラへのフィードバックや、スケジューラから送られてきたスケジューリング情報をもとにメガスケール環境にタスクの配置を行う。

3. 予備評価

前章で述べたメタプログラムによって記述されたモデルの精度についての評価を行う。

ここでは、負荷が不均一な並列プログラムとして Mandelbrot 集合を選ぶ。実行環境は、Intel Pentium III / 1GHz, 256MB, GigabitEthernet で 17 台接続した PC クラスタである。Mandelbrot 集合では、`compute()` のコストは“行数”とし、初期化を除く全ての制御構文を記述している。また、分岐確率については“0.5”とした。N × N 行列のサイズ N を 2048, 4096, 8192, 16384 と変化させ、それぞれについてプロセッサ数を 1, 2, 4, 8, 16 と変化させ測定した。スケール係数 α はサイズ $N = 2048$ かつノード数 = 1 の時の実測値とメタモデルのコスト値が一致するように決定した。

結果は表 1 に示す。表中のコストはスケール後のコスト値であり、実測値との平均誤差は 13%である。サイズ $N = 2048$ でノード数=16 の時の誤差は 53%となり、これが最大誤差である。Mandelbrot の場合、使用するノード数が多くなるほど誤差が大きくなっており、ノード数=1 の場合はほぼ一致している。

ここではノード数が大きくなるにつれて誤差が大き

表 1 Mandelbrot 集合の結果

ノード数	2048			4096			8192			16384		
	実測値	コスト	誤差	実測値	コスト	誤差	実測値	コスト	誤差	実測値	コスト	誤差
1	19.0	19.0	0.0	76.3	76.0	0.37	303.9	304.0	0.03	1217	1216	0.13
2	9.6	9.5	0.59	38.3	38.0	0.72	153.4	152.0	0.91	621	608	2.11
4	4.6	4.8	3.99	17.0	19.0	11.79	67.8	76.0	12.11	277	304	9.42
8	3.3	2.4	28.8	11.7	9.5	18.9	46.6	38.0	18.4	188	152	19.3
16	2.6	1.2	53.7	7.1	4.75	32.9	27.0	19.0	29.7	109	76	30.4

くなっているのが読み取れる．原因としては，不均一な計算負荷をモデルが十分な精度で表現できておらず，その結果として効率のよいスケジューリングが行われていないことが挙げられる．すなわち，現在スケジューリングに利用している概算コストが，現実のコストと大きく異なっていることが，この結果から分かる．そこで，次章ではコストをより正しく見積もるためのプロファイリング手法について述べる．

4. 実行時情報プロファイリング

前章では，予備評価で明らかになったモデル精度の問題点について述べた．その解決案として，本章では，プロファイリング手法について詳しく説明する．

4.1 プロファイリング API

メタプログラム内に記述された抽象的な表現方法を保ちつつ，簡単な記述方法でタスク内部の情報を利用できれば，より実際のタスクプログラムに忠実なふるまいを表現できると考えられる．そこで，タスクの特性を詳細に知る枠組として，実行時情報プロファイリング機構を導入する．すなわち，メタプログラム中の関数呼び出しの形式で取得すべきプロファイル情報を指示できるようにし，それに基づくプロファイルコードをタスクプログラムに埋め込む機構を実現する．指示情報の抽出は従来のメタプログラムトランスレータを改良して行い，プロファイリングには Omni コンパイラ³⁾ の Java tool kit のメソッドを利用する．

プロファイリング情報として関数やブロック文の実行時間，ループカウントや変数の値，メッセージ通信量などのタスク特性情報が挙げられる．

- プログラム実行時間
主にユーザが知りたいブロック文や関数などの部分的な実行時間を知るのに適している．返す値は実数である．
- メッセージ通信量
メッセージの送受信の量を取得する．返す値はバイトを単位とする整数である．
- 変数の値
タスクプログラムの計算量を支配する変数の値は，性能モデリングに特に重要である．なおループカウントは暗黙の変数として取り扱う．返す値は整数または実数である．

```
class task A
def initialize(param)
  @exefile=tasks @arg=param
end
def behavior
  FOR 100
    x=getTime( func_name=main,
               block_count=1 )
    compute(x)
    IF 0.5
      compute(1)
    ELSE
      BREAK
    END
  END
end
end
```

図 3 メタプログラムへの適用例

これらを記述するための API として，以下に示す 3 種の関数を定めた（表 2）．

- getTime()
function_name が示す関数の実行時間を取得する．block_count = b も指定された場合，関数中の b 番目のループブロックの実行時間を取得する．
- getAmountOfMessage()
function_name が指定する関数中のメッセージ通信量を取得する．block_count = b が指定されれば b 番目のブロック中の通信量を，さらに position=p が指定されればブロック中の p 番目の通信関数での通信量を取得する．
- getValueOfVariable()
function_name が指定する関数中の block_count = b が指定する b 番目のブロックのループカウントを取得する．variable_name = v が指定されると変数 v の値を取得し，position = p も指定されると b 番目のブロックの内部（または before(p): 前，あるいは after(p): 後）の p 番目のように記述された代入値を取得する．

なお上記のように引数はキーワードと値との組で指定されるので，引数順序を記憶する必要はなく，指定不要の引数は適宜省略できる．

4.2 プロファイリング方法

メタプログラムから情報取得までの方法を説明する．

表 2 プロファイリング API

API 記述方法	機能
<code>getTime(function_name=main, block_count=2)</code>	時間の取得
<code>getValueOfVariable(function_name=main, variable_name=i, block_count=2, position=1 after(1) before(1))</code>	変数の値取得
<code>getAmountOfMessage(function_name=main, block_count=2, position=1 after(1) before(1))</code>	メッセージ量取得

meta program

```

getTime( function_name = function,
          block_count = 2 )
getValueOfVariable( function_name = function,
                    block_count = 2 )
getAmountOfMessage( function_name = function,
                    block_count = 1 )

```

task program

```

for( i=0; i<end_1; i++) {
  currentTimeStart():
  for( j= _init = init; j<end_2; j++) {
    /* loop body */
  }
  currentTimeStop():
  getValue(j-_init):
  MPI_Send(buffer, 1024, MPI_INT,
            1, 1234, MPI_COMM_WORLD);
  getMessage(buffer, 1024, INT):
}

```

図 4 プロファイリング例

まずメタプログラムからプロファイリング API とタスクプログラムファイルの情報を抽出する。続いてタスクプログラムのソースファイルに Omni コンパイラを適用し、その中間表現を解析して API が指示するプロファイリング位置を求める。最後に API が定めるプロファイリングコードを埋め込み、再コンパイルしてインストルメントされたタスクの実行ファイルを得る。

以下にプロファイリング例を示す。

図 4 の `getTime()` は、関数の 2 番目の `for` ループの実行時間を取得することを指定している。したがって、内側の `for` ループの前後に実行時間を計測するコードが埋め込まれる。なお現状では例に示したように、外側ループが回転するたびに内側ループの実行時間が計測されるが、計測やロギングのコスト削減のために特定の k 回目の値だけ取得、 n 回ごとのサンプル値を取得、複数回取得した値の統計や平均値をロギングするなどの手法を用意し、ユーザの指示にしたがって選択することも検討している。

図 4 の `getValueOfVariable()` によるプロファイルコードも、2 番目のループ（内側ループ）のループカウンタを外側ループが回転するたびに取得している。またこの例ではループ制御変数 j の最終値を利用しているが、`while` ループなどではループ中にループカウンタを埋め込んで計測する。

図 4 の `getAmountOfMessage()` では、通信関数 `MPI.Send` の引数情報からメッセージサイズを取得するプロファイルコードが埋め込まれている。

4.3 メタモデルの精緻化

プロファイリングに用いられる API はそれぞれで取得される値の種類が異なる。現時点では取得した値をそのまま無単位のコストオブジェクトとして返し、必要な変換（たとえば時間から FLOP へ）は、システムが提供する実行環境に依存する変換関数を用いてユーザが計算することとしている。

例えば、実行時間から計算コストを得るには、以下のような記述をする。

```

def behavior
  x=getTime(function_name=main,
             block_count=2)
  compute(timeToFlop(x))
end

```

上記に対応するメタモデルは、以下の記述に相当するものとして生成される。

```

def behavior
  x=Cost("_getTime_0*_system_flops")
  compute(x)
end

```

たとえば 3 章の例では、実行時引数から算出される値を計算コストとしていたが、上記の例ではプロファイル値を格納する変数 (`_getTime_0`) が自動的に用意され、その値に基づくより正確なコスト式で計算コストを与えることができています。なお、このプロファイル情報を利用したモデルの精緻化は、タスクの 2 回目以降の実行に適用される。

現在、プロファイリング機構を用いての動作確認は、実行環境 Intel Xeon 2.8GHz(Dual Processor) を用いて行っており、機能部分は正常に動作することが確認されている。今後は、プロファイリングによるモデルの精緻化によって、モデルの精度が向上することを検証する予定である。例えば 3 章の Mandelbrot の例では、負荷の不均一性を特定のループのループカウンタから求めれば高精度のモデルが構築できることがすでに明らかになっており、従来の 50% 程度の最大誤差をプロファイリングによって数パーセント程度にできるものと見込んでいる。

5. 関連研究

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。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いる。

AppLeS(Application-Level Scheduling)⁴⁾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を用いている⁵⁾。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る式で表される。開発者はコンポーネントを任意に定義することができ、1つのコンポーネントについて複数の実装もたせることができ、モデル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。しかし、正確な構造モデルを構築するには対象アプリケーションを詳細に解析する必要があり、並列プログラムに対する知識も必要となる。また、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、アプリケーションの開発者以外には困難な作業である。

実行予測技術を用いた Grid システムのリソース管理システム⁶⁾では、性能予測ツール PACE Toolkit⁷⁾を用いてアプリケーションの性能予測を行う。PACE Toolkit はアプリケーションツール・リソースツール・評価エンジンで構成されている。アプリケーションツールでは、ソースコードを解析しアプリケーションの様相と並列性を取得しアプリケーションモデルを生成する。リソースツールはハードウェア (CPU, メモリ, ネットワーク) の性能を測定するベンチマークをもち、リソースモデルの生成を行う。これらのモデルを評価エンジンに与えることで、予想実行時間やスケラビリティ、リソースの予想利用率などを得ることができる。しかし、PACE により生成される予測モデルでは実行時引数による実行時間の特性変化を得ることはできない。また、ソースコードの静的解析によりモデリングを行うため、対象となる言語に限られる。このため、タスクの記述言語を指定していない MegaScript には不向きである。

6. おわりに

本論文では、MegaScript の概要とタスクの実行情報プロファイリングの設計と実装について述べた。また、API のプロファイリング機構は、Dual Processor 上で、定義した API 機能が正常に動作することが確認できている。また、実行時情報プロファイリングでは予備評価で行った最大誤差を最大でも数パーセントに抑えられることが見込まれる。今後は、クラスタによる評価をメタプログラムと合わせて動作させ、取得されたコストからメタモデルの精度向上を示すことで

ある。また、コストに対する評価方法の改善などが挙げられる。

謝辞 本研究の一部は、科学技術振興機構・戦略的創造研究推進事業「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) 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).
- 3) 草野, 佐藤 茂久, 佐藤三久: Omni OpenMP コンパイラの性能評価, 情報処理学会論文誌, Vol. 42, No. 4, pp. 802-811, 2001.
- 4) Berman, F., Wolski, R., Figueira, S., Schopf, J. and Shao, G.: Application-level scheduling on distributed heterogeneous networks, *Proceedings of SuperComputing '96* (1996).
- 5) Schopf, J.: Structural Prediction Models for High-Performance Distributed Applications, *Cluster Computing Conference '97* (1997).
- 6) Jarvis, S. A., Spooner, D. P., Keung, H. N. L. C., Cao, J., Saini, S. and Nudd, G. R.: Performance Prediction and its use in Parallel and Distributed Computing Systems, *International Parallel and Distributed Processing Symposium 2003 (IPDPS'03)*, pp. 228-251 (2003).
- 7) Cao, J., Kerbyson, D. J., Papaefstathiou, E. and Nudd, G. R.: Performance Modeling of Parallel and Distributed Computing Using PACE, *In Proceedings of 19th IEEE International Performance, Computing, and Communications Conference (IPCCC 2000)* (2000).