タスク並列言語MegaScriptにおける高精度実行モデルの構築

湯山紘史[†,☆] 津邑公暁[†] 中島浩[†]

我々は，汎用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る．MegaScriptは，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の記述に基づいて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，負荷が不均一またはメッセージ通信の多いプログラムでは低下する問題がある．そこで本研究では，MegaScript向けのタスク情報を取得する枠組みを導入した．評価の結果，従来のモデルに比べて精度が大幅に高いモデルを構築できることを示した．

Construction of Accurate Task Models for the MegaScript Task Parallel Language

Hiroshi Yuyama,[†,☆] Tomoaki Tsumura[†] and Hiroshi Nakashima[†]

We are pursuing research works on our task-parallel script language named MegaScript for general purpose mega-scale computing. A feature of MegaScript is the capability to describe abstructed 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 that a large modeling error occured in ill-balanced and/or communication-bound parallel programs. This paper proposes a solution for the accuracy problem by introducing a profiling scheme into MegaScript meta-program. Our evaluation exhibited the profiling greatly improves model accuracy for the parallel programs with the accuracy problem.

1. はじめに

近年，数十Tera-FLOPS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・ゲノム解析などではPeta-FLOPS以上の計算能力が求められており，Peta-FLOPS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となる．

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，大規模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て現実的にメガスケール計算環境を実現・運用することを目的とした「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．この研究が目指すメガスケール環境は，性能の異なる計算機やネットワークの集合として構築されるため，それらの資源を効率良く利用する実行環境が必要である．そのための手段として，我々はオブジェクト指向スクリプト言語Rubyをベースとしたタスク並列言語MegaScriptの開発を行っている．MegaScriptで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と，それに基づき特性を考慮したタスクスケジューリングを行う枠組みが用意されている．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は，逐次または負荷が均一な並列プログラムでは高い精度が得られている．しかし，メッセージ通信の多いプログラムや負荷が不均一なプログラムでは，プログラム特性を十分に反映できず精度が低下する問題点がある．

そこで我々は精度向上の手段として，インストゥルメンタを設計・実装した．インストゥルメンタは，プロファイル情報として，実行時間・変数の値・メッセージ通信量を扱う．プロファイル情報の取得位置は，関数名やその中のブロック文を指定することによりメタ

† 豊橋技術科学大学
Toyohashi University of Technology
☆ 現在，株式会社富士通システムソリューションズ
Presently with Fujitsu System Solutions, Ltd.

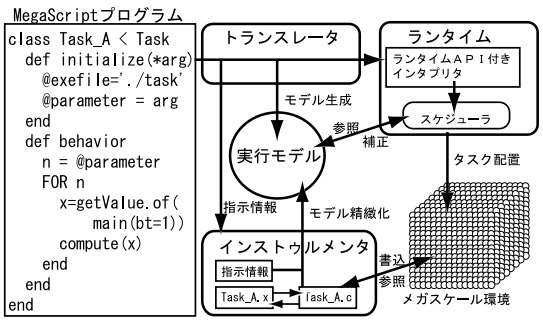


図 1 MegaScript システムの全体像
Fig. 1 Outline of MegaScript system.

プログラム内に記述できる．インストゥルメンタは，この記述をもとに，プロファイルコードをタスクプログラムに埋め込み，実行時に取得されたプロファイル情報を，モデルに反映する．

以下，本論文では，2 章で，本研究の背景となっているタスク並列言語 MegaScript システムについて述べる．3 章では実行モデルについて述べ，現在のモデル精度の問題点をあげる．4 章では本研究の核となる高精度実行モデル構築の設計と実装について述べ，5 章では構築したモデルの精度について評価を行う．6 章では関連研究について紹介し，7 章でまとめる．

2. タスク並列言語 MegaScript

本章では，研究の背景である MegaScript について述べる．

2.1 概　　要

大規模な並列プログラムをユーザが一から記述することは非常に困難である．そこで我々は，SPMD など従来の枠組みで設計された並列または逐次のプログラムを下位層のタスクとし，それを多数組み合わせたタスク並列プログラムを上位層とする 2 階層並列プログラミングと，上位層記述のためのスクリプト言語 MegaScript を提案している[1)]．また大規模かつヘテロな環境下で自動的なタスクスケジューリングを効率的に行うために，MegaScript にはタスクの挙動をユーザが抽象的に記述するメタプログラムの枠組みも用意されている．

2.2 タ　ス　ク

MegaScript の並列実行単位であるタスクは，外部プログラムとして与えられる．したがって MegaScript のプログラムや処理系はタスク内部の動作にはいっさい関与しない．また，タスクは **図 1** のようにシステム定義クラス Task を継承する形で定義され，タスク情報は initialize メソッドと behavior メソッドに分けて記述される．さらに，複数のタスクが協調動作をして 1 つの問題を解くためには，それらのタスク間でデータのやりとりを行う必要がある．そこで MegaScript では，「ストリーム」という論理的な通信路を設け，独立した実行プログラムである各タスクの外部から観測・操作可能な入出力である標準入出力を接続する方法を実現している．ストリームには，複数のタスクが接続可能で，マルチキャストやマージを行うことが可能である．

2.3 メタプログラム

MegaScript では，ユーザがタスクの振舞いに関する情報を記述することができる．この記述プログラムを「メタプログラム」と呼ぶ．メタプログラムは 図 1 の behavior メソッド内に記述する．このようにプログラムを用いているのでメタプログラムはきわめて高い記述性があり，ユーザの知識レベルに応じた記述ができるよういくつかの抽象化表現が用意されている．抽象化表現する項目について簡単に述べる．

- コストオブジェクト
  計算・通信のコストやループ回転数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うえで式が評価される．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．
  コストオブジェクトは次のように記述することで生成することができる．変数 `l`，`m`，`n` は，いずれも内部情報として数式 “`x`” を持つ．

```
l = Cost.new("x")
m = Cost("x")
n = C_.x
```

  コストオブジェクトは，内部情報として式を持ち “n^2+1” といった数式や “10” や “0.1” といった数値を持つこともできる．
- 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*`)` という書式で使用し，*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*`)` という書式で使用し，*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，その内部での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*`)` / `msgrecv(`*cost*`)` という書式で使用し，*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通信されることを表す．

2.4 処 理 系

MegaScript プログラムをメガスケール環境で実行するための処理機構について説明する．機構はトランスレータ，ランタイム，ランタイムから呼び出されるスケジューラの 3 つの基本部分からなり，これに本研究ではインストゥルメンタを付加する．

トランスレータ

トランスレータは，MegaScript プログラムをタスククラスの behavior メソッドと，それ以外の部分に分離する．前者は実行モデルを生成するためのメタモデル生成スクリプトに変換され，後者はランタイムに渡されて解釈・実行される．

ランタイム

ランタイムは，トランスレータによって分離された MegaScript プログラムを解釈し，分散環境上でのタスク配置やタスク間通信を実現する．また，システムリソースの状況やタスクの実行時間といった動的な情報の収集も行う．

スケジューラ

スケジューラは，ランタイム内部で動作し，タスク配置戦略を決定する．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 つからなる．スケジューラは，まず実行モデルと実行環境の情報をもとにタスクの初期配置を決定する．いくつかのタスクの実行が終了すると，ランタイムから通知される動的情報から実行モデルの精緻化を行い，それに基づいてタスク配置戦略を動的に補正する．

インストゥルメンタ

インストゥルメンタは，メタプログラムに記述されたタスク内部情報取得指示関数に基づき，タスクプログラムにプロファイルコードを挿入し，それによって得られた情報を用いて，高精度なモデルを構築する機構である．具体的には，メタプログラム中にプロファイル対象を記述する枠組みを設け，その情報からタスクのソースプログラムにプロファイルコードをインストゥルメントする．またプロファイル結果から計算・通信などのコストを求め，それをモデルに反映して精緻化する．これにより，タスクの挙動がより正確になるのでスケジューリングの効率化を図ることができる．インストゥルメンタについては 4 章で詳しく議論する．

3. MegaScript の実行モデル

前章では，大きな土台である MegaScript システムについて述べた．本章では，本研究で扱う材料であるメタプログラムから構築される実行モデルとモデル精度の問題点について議論する．

3.1 基 本 概 念

MegaScript では，効率良くタスクを割り当てられるように，スケジューラがその役割を果たさなければならない．そのために，スケジューラはタスクの情報（計算量・通信量）を知ることが必要である．そのために，タスクのコスト予測モデルを用意する．このモデルが「実行モデル」である．MegaScript では，タスクの情報はメタプログラムから得ることができるので，その情報から構築される実行モデルを「メタモデル」と呼ぶことにする．

メタモデルの実態は，メタプログラムから MegaScript トランスレータによって生成され，スケジュー

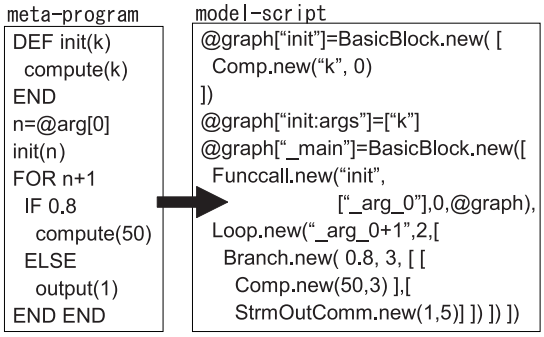


図 2 メタプログラムとそれに対するモデル生成スクリプトの例
Fig. 2 An example of meta-program and the model generation script derived from it.

ラによって解釈実行される一種のプログラムである．この「メタモデル生成スクリプト」と呼ぶプログラムは，メタプログラムの抽象構文木に対応し，複数の構文木とノードから構成される．抽象化構文木は，抽象化関数ごとに生成され，それぞれ関数名を名前として持つ．

図 2 に MegaScript プログラムとその情報を元に生成されるメタモデル生成スクリプトを示す．メタプログラムの制御構文（ループ文や分岐文）や，ステートメントなどが解釈され，すべて関数として出力される．

3.2 コストの算出

図 2 の例を用いて，タスクの実行時引数 `n` によって実行モデルのコストを具体化し，その計算コストを算出するスケジューラ用のモデル解釈関数 `comp_cost(n)` の動作を説明する．

このメタプログラムには，抽象化関数変数 `init` が定義されており，タスクの実行時引数`@arg` を変数 `n` を経由して受け取り，その値に相当するコストの計算が行われる．続いてループカウント `n+1` の `FOR` 文中で `IF` 文が実行される．`IF` 文は 0.8 の確率で `compute(50)` が実行され，0.2 の確率で `output(1)` が実行される記述になっている．

計算コストは，“(`init` 関数のコスト) + (`FOR` 文のコスト)” と算出され，これを式で表すと，“$n + (n + 1) \times (0.8 \times 50 + (1 - 0.8) \times 0)$” となる．なお `output` の計算コストは 0 と定義されている．たとえば，実行時引数に 20 を与えると先ほどの式より，計算コストは 860 となる．

メタモデルには，計算コストメソッド以外にも，コスト取得メソッドとして **表 1** のようなものが定義されている．スケジューラは，メタモデルオブジェクトに対しこれらのメソッドを適用してコストを求め，それに基づいてタスク配置戦略を定める．

表 1 コスト取得メソッド一覧
Table 1 Methods for cost approximation.

メソッド名	機能（メタ関数）
`comp_cost`	計算コスト (`compute`)
`strmcomm_cost`	stream 通信コスト (`input`+`output`)
`strmin_cost`	stream 通信の入力コスト (`input`)
`strmout_cost`	stream 通信の出力コスト (`output`)
`strmcomm_count`	stream 通信の送受信回数
`msgcomm_cost`	message 通信コスト (`msgsend`+`msgrecv`)
`msgsend_cost`	message 送信のコスト (`msgsend`)
`msgrecv_cost`	message 受信のコスト (`msgrecv`)
`msgcomm_count`	message の送受信回数

```
01: SIZE = @arg[0]
02: $maxloop = 251
03: DEF mandel_color()
04:   compute(2)
05:   FOR 0..$maxloop
06:     compute(4)
07:     IF 0.5
08:       BREAK
09:     END
10:   END
11: END
12: procs =16
13: chunk = SIZE /procs
14: PARALLEL procs DO
15:   s=0; e=chunk
16:   FOR k IN 0..procs
17:     compute(2)
18:   END
19:   FOR s..e
20:     FOR SIZE
21:       mandel_color()
22:     END
23:   END
24:   IF ID!=0
25:     msgsend(SIZE)
26:   ELSE
27:     FOR 1..procs
28:       msgrecv(SIZE)
29:     END
30:   END
31: END
```

図 3 Mandelbrot 集合のメタプログラム
Fig. 3 Meta-program for Mandelbrot set problem.

3.3 予備評価

現在の MegaScript 処理系の第 1 バージョンが生成する実行モデルの精度とその問題点を明らかにするために予備評価を行った．評価プログラムには $N \times N$ の空間を対象とする Mandelbrot 集合を用い，N の値は 2048，4096，8192，および 16384 とした．評価環境は 16 台の Mobile Intel Pentium III-M（866 MHz, 512 MB）を GigaBit Ethernet で結合した PC クラスタであり，OS には Red Hat Linux 7.2 を用いた．また，メタプログラムには，初期化を除くすべての制御構文を記述し，関数 `compute(`n`)` の引数 n はプログラムの行数とした．メタプログラムを **図 3** に示す．

結　果

結果を **表 2** に示す．結果より，全体を見るとプロセッサ数の増加にともなってモデルの精度誤差が大きくなっていることが分かる．すなわちノード数が 16 の場合に 60%以上のモデル誤差が生じており，$N = 2048$ の場合に最大値 71.9%となっている．

問　題　点

結果より，現在の MegaScript の実装では，負荷が不均一な場合，モデル精度が低下するという問題点がある．モデル精度は自動スケジューリングの効率に直結するため，モデル誤差が大きいとタスク並列プログ

表 2 Mandelbrot 集合の結果
Table 2 Modeling error of Mandelbrot set problem.

ノード数	$N = 2048$			$N = 4096$			$N = 8192$			$N = 16384$		
	実測値 [sec]	コスト	誤差 [%]	実測値 [sec]	コスト	誤差 [%]	実測値 [sec]	コスト	誤差 [%]	実測値 [sec]	コスト	誤差 [%]
1	12.0	12.0	0.0	48.2	48.0	0.3	192	190	1.3	771	729	5.4
2	6.06	5.97	1.5	24.2	23.9	1.5	96.9	96.4	0.5	388	350	9.7
4	5.68	2.99	47.3	22.6	11.9	47.2	90.5	48.2	46.7	362	178	51.0
8	3.92	1.44	63.3	15.5	5.95	61.5	61.6	24.1	60.9	246	88.6	64.0
16	2.43	0.68	71.9	9.00	2.89	67.9	35.2	11.8	66.4	140	43.7	68.9

ラムの効率が著しく低下する可能性がある．精度向上の解決策として，プログラムの実行時間に影響を与える要素を十分に把握して，ユーザが詳細なプログラムを記述することがあげられる．しかし，メタプログラムの記述容易性や知識レベルを問わないなどのメリットが消えてしまう．そこで，タスク特性の十分な把握を自動的に行うための関数をメタプログラム上に定義するアプローチで，実行モデルの精度向上を図る．次章で詳しく述べる．

4. インストゥルメンタ

本章では，前章の問題点を解決するインストゥルメンタの設計と実装について述べる．

4.1 設　　計 方　　針

現在の MegaScript メタプログラムは，ユーザが知りうる知識に依存した記述となるため，たとえばタスクの実行中に明らかになるコスト情報を反映したモデルを生成することはできない．したがってモデルの精度はタスクの性格やユーザの知識に大きく依存するため，スケジューラがモデルを用いて効率良くタスクを配置できる保証はない．効率良くスケジューリングするにはタスクの内部の情報を得ることが 1 つの手段としてあげられる．タスクの内部情報を得るためには，ユーザがその対象や取得方法を定めるためのインタフェースを用意する必要がある．そこで，プロファイリング API を定義し，メタプログラム内に指示情報を記述する枠組みを提案する．指示情報とはタスクプログラムのどのような情報を得るかを定める関数である．この関数はメタプログラムの内部に記述されるため，MegaScript の設計方針である記述容易性を保たなければならない．本研究では，前述の記述容易性を保つことに加え，タスクの的確な情報を提供できることと，プロファイルに要する時間をできるだけ小さくしタスク実行への悪影響を最小限とすることを目標とする．

API の定義

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振舞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みとして，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あげられる．

- 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．
- 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．
- 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．

これらを記述するための API として，以下に示す 3 種の指示情報を定めた（**表 3**）．

- `getTime.`*how*`(`*func*`[(`*opt*`[,...])])`
  func が示す関数の実行時間を *how* で指定する方法（後述）により取得する．*opt* に `blockCount=`*b* が指定された場合，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．
- `getValueOfVariable.`*how*`(`*func*`[(`*opt*`[,...])])`
  func が指定する関数中の `blockCount=`*b* が指定

表 3 インストゥルメント API 記述例
Table 3 Examples of API for instrumentation.

API 記述方法	取得情報
getTime.averageOf(*func*(blockCount=2))	時間
getValueOfVariable.maxOf(*func*(variableName=i)	変数の値
getAmountOfMessage.of(*func*(blockCount=2, position=after(1)))	メッセージ量

する b 番目のブロックのループカウントを取得する．variableName=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 番目の代入値を取得する．

- getAmountOfmessage.*how*(*func*[(*opt*[,...])])
  func が指定する関数内のメッセージ通信量を *how* で指定する方法で取得する．*opt* に blockCount=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．

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．

また，それぞれの指示情報に対しての取得機能 *how* を以下のとおりに定める．

- of
  指定された情報そのものを取得する．すべての API で採用される．
- averageOf
  取得情報の平均をとった値を提供する．時間と変数で採用される．大まかな挙動を把握するのに適している．
- maxOf
  取得情報の最大値をとる．すべての API で適用される．不均一な負荷のプログラムに対して有効である．
- minOf
  取得情報の最小値をとる．すべての API に適用される．ユーザが知りたい情報として提供されている．
- headOf
  取得情報の最初の要素をとる．ループ内のループカウント数，時間，メッセージサイズに適用される．
- tailOf
  取得情報の最後の要素をとる．headOf と同様の用途に使用できる．

さらに，プロファイリングの位置情報を得るための機能として以下にあげる指定方法を提供する．指定方法は，たとえば

```
blockCount = 1
```

のように，オプションの名前に値を代入するような形で表現することができる．このような形の引数を「ペア引数」と呼ぶ．以下，ペア引数により位置を指定する記述方法を以下にあげる．カッコ内に示すように，オプションの省略名も提供している．

- blockCount (bc)
  ブロック文を上からカウントした番号を指定する．
- blockName (bn)
  ブロック文の名前（for，while，do）を指定する．前述の blockCount と併用して使用する．
- position (pt)
  指定された場所から数えた位置を指定する．指定方法は，position=n，position=after(n)，position=before(n) のように表すことができる．
- variableName (vn)
  変数の名前を文字列で指定する．

なお，これらは順序を問わない記述が可能である．

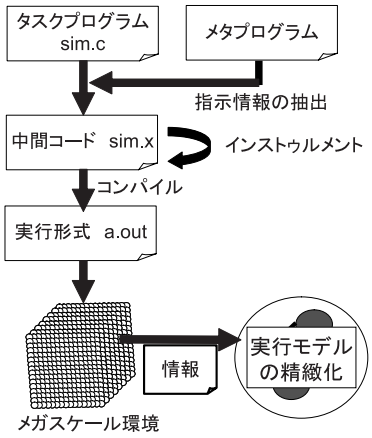


図 4 インストゥルメンタの処理過程
Fig. 4 Flow of instrumentation.

メカニズム

メタプログラムから情報取得までは以下の手順により行う．**図 4** にメタプログラム記述からモデル精緻化までのプロセスを示す．

(1) メタプログラムからインストゥルメント API と

```
01: x=getTime.of(main(blockCount=1))
02: FOR x
03:   compute(20)
04: END
05: FOR 100
06:   y=getValueOfVariable.averageOf(
07:     func_2(blockCount=3)
08:     <-func_1(position=1)
09:     <-main(blockCount=1)
10:    )
11:   compute(y)
12:   IF 0.5
13:     compute(x)
14:   ELSE
15:     BREAK;
16:   END
17: END
```

図 5 メタプログラムへの適用例
Fig. 5 Example of meta-program with instrumentation instructions.

```
#include <stdio.h>

int main(){
  int n;
  n=input();
  func_1();
  t0=MPI_Wtime();
  for(int i=0;i<10;i++){
    func_1(n);
  }
  t1=MPI_Wtime();
  getTime(t1-t0);
  func_2();
  return 0;
}
```

図 6 インストゥルメントされたタスクプログラムの例（getTime）
Fig. 6 Example of instrumented task program with getTime.

タスクプログラムファイルの情報を抽出する．

(2) タスクプログラムのソースファイルに Omni コンパイラのフロントエンドを適用して中間コードを得る．

(3) この中間コードを解析して API が指示するインストゥルメント位置を求める．

(4) API が定めるインストゥルメントコードを埋め込む．

(5) 中間コードをデコンパイル/再コンパイルして，インストゥルメントされたタスクの実行ファイルを得る．

以下に指示情報とタスクプログラムを用いてインストゥルメント方法を述べる．

図 5 にインストゥルメントの指示情報を記述したメタプログラムを示す．**図 6** には getTime の際に埋め込まれるコードが，**図 7** には getValueOfVariable の際に埋め込まれるコードが示されている．それぞれの指示情報について説明する．

まず，図 5 のメタプログラムの 1 行目に記述されている getTime について述べる．getTime は時間取得の指示関数である．機能には of（縮約演算なし）が，また指定位置には main 関数が，それぞれ指定されている．またペア引数 blockCount=1 により，関数中の最初のブロック文がプロファイル対象であることが指示されている．このような指示情報を指定することによって，タスクプログラムでは，main 関数内の最初のブロック文の実行時間を取得するためのコードが埋め込まれる．ここでは MPI 関数の MPI_Wtime 関数が使用されている．この関数は現在時刻を返すので，ブロック文の前後で関数を呼び出し，その差分をとることによって実行時間を取得する．

```
int global_1=0;
int main(){
  int n;
  n=input();
  func_1();
  for(int i=0;i<10;i++){
    global_1+=1;
    func_1(n);
    global_1-=1;
  }
  func_2();
  return 0;
}

void func_1(int m){
  global_1+=2;
  func_2();
  global_1-=2;
  if(m==1){
    func_2();
  }
  else{
    for(i=0;i<M;i++){
      func_3();
    }
  }
  return;
}

void func_2(){
  for(int i=0;i<n;i++){
    process;}
  for(int k=0;k<M;k++){
    for(i=0;i<n;i++){
      if(cond)break;}
      if(global_1==3){
        count++;
        total+=i ;
      }
  }
  if(global_1==3){
    getValue(total/count);
  }
}
```

図 7 インストゥルメントされたタスクプログラムの例（getValueOfVariable）
Fig. 7 Example of instrumented task program with getValueOfVariable.

次に，図 5 の 6 行目に定義されている getValueOfVariable について述べる．getValueOfVariable 関数は大域的もしくは局所的な変数の値を取得する指示関数である．機能には平均値を取得する averageOf が，指定位置には func_2 関数が，それぞれ指定されている．またペア引数 blockCount=3 により，関数中の 3 番目のブロック文，すなわち 2 重 for ループの内側ループの回転数を取得することが指示されている．さらに，<-によって func_2 が func_1 内の最初の呼び出しで起動されたものであることと，その func_1 が main 関数の 1 つ目のブロック文内で呼び出されたものであることが指示されている．このような記述をすることによって，インストゥルメンタは，main 関数の 1 つ目のブロック文で呼ばれた func_1 関数が一番最初に func_2 関数を呼んだときにだけ，func_2 関数内の指定先をプロファイルすると解釈する．そのためまず，埋め込むコードに呼び出し元を示すグローバル変数を用意する．図 7 では global_1 が bit vector になっており，main 関数と func_1 に対応するビットがともに on のときのみにプロファイルが行われるコードが埋め込まれている．これによって func_2 が他の部分で呼ばれてもプロファイルすることを防ぐため，ユーザが的確にプロファイルする位置やタイミングを

表 4 評価環境
Table 4 Evaluation environment.

ノード数	16 nodes
CPU	Mobile Intel Pentium III-M 866 MHz
Memory	512 MB RAM
Network	Gigabit Ethernet（1000-BASE, 13port SW:Fujitsu PG-SW101×2, 28port SW:CISCO catalyst 4003×1）
OS	Red Hat Linux release 7.2

指示することができる．

また機能に `averageOf` が指定されているので，取得したループカウントの総和と取得回数から，その平均値を算出してプロファイルするコードが埋め込まれる．

4.2 実　　装

インストゥルメンタの実装は大きく以下の 3 つについて行った．それぞれについて述べる．

- トランスレータパーサの拡張
  パーサは，メタプログラムに記述された指示情報をコストオブジェクトに変換し，実行時引数のように未知の値としてモデルスクリプトに変換する．パーサは，メタプログラム上の指示情報を分離し，それをインストゥルメンタに渡す．
- インストゥルメントコードの埋め込み
  メタプログラム上で記述された指示情報に従って，プロファイル位置を決定し，インストゥルメントコードを生成する．プロファイル位置に対応する中間コードのタスクプログラムにインストゥルメントコードを埋め込む．
- メタモデルの精緻化
  実行モデルスクリプトに記述されている未知のコスト変数をタスクの実行によって取得されたプロファイル情報に書き換える．

5. 評　　価

モデル精度向上を示すために評価を行った．以下に評価環境，結果について述べる．

5.1 評 価 環 境

評価に用いた実行環境は，**表 4** に示すような性能のクラスタを利用した．

また評価に用いたソフトウェアのバージョンは以下のようになっている．

- gcc version 2.95.3
- MPICH 1.2.6 with p4 library
- Ruby 1.8.2
- Omni Compiler version 1.6

5.2 評価プログラム

評価プログラムは，負荷が均一かつメッセージ通信の少ない NAS Parallel Benchmarks の EP，負荷が不均一な Mandelbrot 集合，メッセージ通信の多い 1 次元波動方程式の 3 つを用いた．

```
01: procs =16
02: FOR 0..100
03:   compute(1)
04: END
05: compute(4)
06: np=4096/procs
07: FOR k IN 0..np
08:   compute(3)
09:   FOR 0..100
10:     compute(3)
11:     IF 0.5
12:       compute(10)
13:     END
14:   END
15:   FOR 0..100
16:     compute(5)
17:     IF 0.1
18:       compute(10)
19:     END
20:   END
21: END
22: compute(30)
```

図 8 EP のメタプログラム（プロファイルなし）
Fig. 8 Meta-program for EP without profiling.

```
01: procs =16
02: x=getValueOfVariable.of(
03:       main(blockName=br,
04:            variableName=i__
05:            blockCount=2))
06: FOR 0..x
07:   compute(1)
08: END
09: compute(4)
10: np=4096/procs
11: FOR k IN 0..np
12:   compute(3)
13:   FOR 0..100
14:     compute(3)
15:     IF 0.5
16:       compute(10)
17:     END
18:   END
19:   x=getValueOfVariable.of(
20:       main(blockName=for,
21:            blockCount=5,
22:            va riableName=i__1
23:       )
24:     )
25:   FOR 0..x
26:     compute(5)
27:     IF 0.1
28:       compute(10)
29:     END
30:   END
31: END
32: compute(30)
```

図 9 EP のメタプログラム（プロファイルあり）
Fig. 9 Meta-program for EP with profiling.

5.3 モデル精度の結果

NAS Parallel Benchmarks（EP）

まず，NAS Parallel Benchmarks の EP を用いて評価を行った．EP は，2 次元平面上に原点を平均値としてガウス分布した点集合を，疑似乱数により求めるアルゴリズムであり，モンテカルロ法の基本の 1 つである．EP はきわめて計算集約的な並列プログラムで，通信はほとんど発生しない．

EP の構造は比較的単純であり，計算コストを定めるループ回転数も容易に求めることができる．したがって比較的正確なメタモデルの構築が可能であるが，その精度をプロファイルによってさらに高くすることができるかどうかを評価した．指示情報なしのメタプログラムを **図 8** に，指示情報ありのメタプログラムを **図 9** に示す．図 9 に示すように，2 つの遂次および並列ループの回転数をプロファイルすることによる精度向上を評価した．

表 5 の評価結果に示すように，インストゥルメントをしない場合でも高い精度のモデルが構築できているが，プロファイル情報を用いれば精度がさらに向上し，きわめて正確なモデルが得られることが明らかになった．さらに実行時間オーバヘッドを **表 8** に示す．オーバヘッドは最大値でも 1.16%（1 ノード，2.25 秒）であり，その他の場合には 0.5%未満に抑えられている．

表 6 Mandelbrot 集合の結果（profile）
Table 6 Modeling error of Mandelbrot set problem with profile.

ノード数	$N = 2048$			$N = 4096$			$N = 8192$			$N = 16384$		
	実測値 [sec]	コスト	誤差 [%]	実測値 [sec]	コスト	誤差 [%]	実測値 [sec]	コスト	誤差 [%]	実測値 [sec]	コスト	誤差 [%]
1	12.1	12.1	0.00	48.2	48.2	0.00	193	193	0.00	771	771	0.01
2	6.06	6.11	0.67	24.2	24.4	0.57	97.0	97.5	0.57	388	391	0.68
4	5.68	5.77	1.48	22.6	23.1	1.83	90.5	92.2	1.88	363	369	1.70
8	3.92	3.95	0.31	15.5	15.7	1.62	61.6	62.9	2.08	246	251	2.06
16	2.46	2.66	8.47	9.0	9.1	1.06	35.2	35.6	1.13	141	142	1.36

表 5 EP の結果
Table 5 Modeling error of EP.

ノード数	no profile			profile		
	実測値 [sec]	コスト	誤差 [%]	実測値 [sec]	コスト	誤差 [%]
1	194	194	0.00	197	197	0.00
2	98.4	97.4	1.09	98.8	98.6	0.24
4	49.3	48.7	1.18	49.4	49.3	0.23
8	24.7	24.3	1.59	24.8	24.7	0.60
16	12.5	12.2	2.75	12.5	12.3	1.50

```
01: SIZE = @arg[0]
02: procs =16
03: chunk = SIZE /procs
04: PARALLEL procs DO
05:     s=0; e=chunk
06:     FOR k IN 0..procs
07:         compute(2)
08:     END
09:     compute(1)
10:     FOR s..e
11:         FOR SIZE
12:             x=getValueOfVariable.maxOf(
13:                 mandel_color(
14:                     blockCount=1,
15:                     variableName=loop)
16:                 <-main(blockCount=3))
17:             compute(x)
18:         END
19:     END
20:     IF ID!=0
21:         msgsend(SIZE)
22:     ELSE
23:         FOR 1..procs
24:             msgrecv(SIZE)
25:         END
26:     END
27: END
```

図 10 Mandelbrot 集合のメタプログラム（プロファイルあり）
Fig. 10 Meta-program for Mandelbrot set problem with profiling.

Mandelbrot 集合

2 つ目は Mandelbrot 集合で評価を行った．3.3 節で述べたように，Mandelbrot 集合の生成プログラムは負荷が不均一な並列プログラムであるため，従来のメタプログラムでは十分な精度を得ることができなかった．そこで **図 10** に示すように，各計算ノードの負荷を決定するループ回転数の最大値をプロファイルする記述を加えたメタプログラムを作成し，その効果を評価した．

表 6 はインストゥルメントを行った場合のモデルによって得られる計算コストと実際の計算時間とを比較したものであり，表の最上欄には問題サイズ N（行列サイズは $N \times N$）を示している．なお計算コストを計算時間に換算するためのスケーリング値は，ノード数が 1 で $N = 2048$ の場合の実測値と一致するように設定した．

```
01: $procs=16
02: $local_n = @a rg[0]/$procs
03: DEF step(myid)
04:     IF myid != 0
05:         msgsend(1)
06:         msgrecv(1)
07:     END
08:     IF myid != $procs-1
09:         msgsend(1)
10:         msgrecv(1)
11:     END
12: END
13: PARALLEL $procs DO
14:     FOR @arg[ 1]
15:         IF ID==0
16:             compute(1)
17:         END
18:         IF ID== $procs-1
19:             compute(1)
20:         END
21:         step(ID)
22:         FOR $local_n +2
23:             compute(1)
24:         END
25:     END
26:     IF ID==0
27:         output(1)
28:     END
29: END
```

図 11 波動方程式のメタプログラム（プロファイルなし）
Fig. 11 Meta-program for wave equation problem without profiling.

表 2 と 表 6 の比較から明らかなように，インストゥルメントによってモデル誤差が劇的に改善している．特にノード数が大きい場合の改善が顕著であり，負荷が不均一な並列プログラムに対しては最大負荷をプロファイルによって求めることが効果的であることが明らかになった．なお，ノード数が 16 で $N = 2048$ の場合のモデル誤差が 8.47%と他のものよりも多少大きいが，これは他に比べて通信コストの影響が相対的に大きく，この見積に誤差が生じているものと考えられる．

一方，表 8 に示すオーバヘッドはきわめて小さく，最大でも 0.13%（ノード数 8，0.02 秒）に抑えられている．

1 次元波動方程式

最後に通信量が比較的多い並列プログラムとして 1 次元波動方程式の差分法による求解プログラムを用いて評価した．プログラムの実行時引数は，格子点数 N と時間ステップ T であり，$N = 10000$，$T = 5000$ として測定した．またノード数は 1，2，4，8，16 とした．

メタプログラムは，**図 11** に示すように初期化を除くすべての制御構文を記述し，`compute()` の引数はプログラムの行数とした．またこのプログラムでは通信頻度が比較的多いため，**図 12** に示すように通信量を

```
01: $procs=16
02: $local_n = @a rg[0]/$procs
03: DEF step(myid)
04:   IF myid !=0
05:     x=getAmountOfMessage.of(
            step(position=1))
06:     msgsend(x)
07:     msgrecv(1)
08:   END
09:   IF myid !=$procs-1
10:     y=getAmountMessage.of(
            step(position=2))
11:     msgsend(y)
12:     msgrecv(1)
13:   END
14: END
15: PARALLEL $procs DO
16:   FOR @arg [1]
17:     IF ID==0
18:       compute(1)
19:     END
20:     IF ID==$procs-1
21:       compute(1)
22:     END
23:     step(ID)
24:     FOR $local_n +2
25:       compute(1)
26:     END
27:   END
28:   IF ID==0
29:     output(1)
30:   END
31: END
```

図 12 波動方程式のメタプログラム（プロファイルあり）
Fig. 12 Meta-program for wave equation problem with profiling.

表 7 波動方程式の求解の結果
Table 7 Modeling error of wave equation problem.

ノード数	no profile			profile		
	実測値 [sec]	コスト	誤差 [%]	実測値 [sec]	コスト	誤差 [%]
1	11.5	11.5	0.0	11.5	11.5	0.0
2	10.3	6.13	40.5	10.3	8.72	15.3
4	7.56	3.25	57.0	7.57	5.84	22.9
8	5.55	1.81	67.4	5.55	4.40	20.7
16	3.89	1.09	71.9	3.89	3.69	5.14

表 8 プロファイルによる実行時間オーバヘッド
Table 8 Profiling overhead.

ノード数	評価プログラム		
	EP	Mandelbrot 集合 ($N = 4096$)	波動方程式
1	2.25 (1.16)	0.02 (0.04)	0.01 (0.09)
2	0.34 (0.35)	0.01 (0.04)	0.01 (0.10)
4	0.13 (0.26)	0.01 (0.04)	0.01 (0.13)
8	0.06 (0.24)	0.02 (0.13)	0.004 (0.07)
16	0.01 (0.08)	0.004 (0.04)	0.004 (0.10)

単位・秒．カッコ内（単位・%）は実行時間比．

プロファイルすることでモデル精緻化を行った．

ここでモデルから求められる計算および通信コストを C および M とし，計算時間の見積り値を

$$\alpha C + \beta M$$

と求める．上式の係数 α は計算コストのスケーリング値であり，ノード数 1 の実測値に基づき定めた．また，通信コストのスケーリング値 β は，評価環境での実測値に基づき 0.4×10^{-3} と定めた．

表 7 に，実行時間と，インストゥルメントの有無それぞれについてのモデルのコストと誤差を示す．この結果から，通信量のプロファイルがモデル精度の改善に大きな効果があることが明らかになった．ただしノード数が 4 あるいは 8 の場合の誤差は小さくなく，コストに基づく実行時間の見積り方法の改善が必要であると考えられる．一方オーバヘッドは，表 8 に示すように最大 0.13%（4 ノード，0.01 秒）と，非常に小さい値に抑えられている．

```
01: procs=16
02: np=4096/procs
03: FOR k IN 0..np
04:   FOR 0..200
05:     compute(4)
06:   END
07: END
08: compute(30)
```

図 13 簡略化した EP のメタプログラム
Fig. 13 Simplified meta-program for EP.

```
01: SIZE=@arg[0]
02: procs=16
03: chunk=SIZE/procs
04: PARALLEL procs DO
05:  s=0; e=chunk
06:  FOR s..e
07:   FOR SIZE
08:    x=getValue.maxOf(
09:       mandel_color(bc=1))
10:    compute(x)
11:   END
12:  END
13: END
```

図 14 簡略化した Mandelbrot 集合のメタプログラム
Fig. 14 Simplified meta-program for Mandelbrot set problem.

5.4 メタプログラムの記述容易性

メタプログラムをさらに簡単に記述をしても良いモデル精度が得られるかを評価した．**図 13** に EP のメタプログラム例を，**図 14** に Mandelbrot 集合のメタプログラム例を示す．図 13 のプログラムでは 図 9 のプログラムに比べ約 3 分の 1 の記述量であるが，誤差は 16 ノードの場合が最大で 2.7%，また 1〜16 ノードの平均値は 1.3%であり，表 5 に示したものとほぼ同等の精度が得られた．また，図 14 のプログラムは，図 10 のメタプログラムの約半分の記述量で，$N = 4096$ の場合の誤差は最大 1.8%，平均 1.0%となり，やはり表 6 に示したものとほぼ同等の値となった．これらの結果より，ユーザがプログラムの実行時間に与える重要なファクタを把握していれば，その部分だけを記述することにより前述したものと同等のモデル精度が得られることが明らかになった．

5.5 ヘテロな環境への適用

MegaScript の枠組みでは，メタプログラムやメタモデルの記述・構築は計算環境に依存することなく行われ，特定の環境に関する性能は，環境固有のパラメータを加味したモデル解釈によって推定する．この方法の妥当性を評価するため，5.3 節に示したメタ

表 9 ヘテロな環境での実行時間とモデル誤差
Table 9 Execution time and modeling error in heterogeneous environment.

	実測値 [sec]	モデル誤差 [%]
EP	24.7	0.32
Mandelbrot 集合	17.9	12.4
波動方程式	5.95	26.0

プログラムに基づく性能推定が，ヘテロな計算環境でどのような精度を示すかを評価した．計算環境には，表 4 に示したクラスタ中の 4 台と，Pentium III（1 GHz，256 MB）4 台からなるミニクラスタ☆を，Gigabit Ethernet で結合した合計 8 台のヘテロクラスタを用いた．また評価プログラムには，EP，Mandelbrot 集合（$N = 4096$），波動方程式の 3 つを用いた．なおこれらのプログラムの実行時間は，低性能のプロセッサによって定まると考えられるため，性能推定値には 5.3 節で示した値（プロファイルあり）をそのまま使用した．

表 9 にヘテロな環境で実行したときの時間とモデル誤差を示す．表から明らかなように，EP のモデル精度はほとんど変わらず☆☆，計算コストが支配的な場合にはヘテロな環境でも高精度の性能推定が可能であるといえる．一方 Mandelbrot 集合については誤差がかなり大きくなり，波動方程式についても 6%ほど悪化している．この要因は，後述するようにヘテロ環境のネットワーク構造を反映したモデル解釈ができていないことにあると考えられる．

5.6 考　　察

本節では，5.3〜5.5 節の評価結果に関する詳しい考察を行う．

まず，インストゥルメントにかかる時間に着目してみると，ノード数にかかわらず，3 つのプログラムではどれも低オーバヘッドのプロファイリングが実現できていることが明らかになった．プロファイリングのオーバヘッドとしては，対象情報の取得とその結果のファイル出力が主要因となる．このうち特に大きな影響を与える後者について，縮約演算をプロファイル時に行ってファイル出力を最小限にとどめたことで好結果が得られたものと考えられる．

次に実行時間とモデルとの誤差に着目すると，ホモな環境では EP と Mandelbrot 集合では，後者の一例（ノード数 16，$N = 2048$）を除いて誤差は 2%以下に抑えられている．EP は負荷が均一な計算集約的プログラムであるので，プログラムの制御構造を反映したメタプログラムにより高精度なモデルが構築でき，ループカウントをプロファイルすればさらに精度を向上させることができる．したがって MegaScript プログラムの 1 つの典型であるパラメータサーベイのような問題に対しては，高精度のモデルが構築できるものと期待することができる．

また Mandelbrot 集合のような負荷が不均一なプログラムに対しても，最大負荷を定めるループカウントの最大値をプロファイルすれば，非常に精度の良いモデルを構築できることが明らかになった．最大負荷が割り当てられた計算ノードの実行時間が全体の実行時間を定めるという性質は，多くの負荷不均衡なプログラムに共通しており，ループカウントなど負荷を定める情報の最大値を取得する方法は適用性が高いものと考えられる．

またこの 2 つのプログラムについては，メタプログラムの記述を大幅に簡略化しても，モデルの精度がほとんど低下しないことも明らかになった．メタプログラムの作成者としては，タスクプログラムのプログラマである場合と，タスクプログラムのユーザである場合の 2 種類が考えられるが，後者の場合にはソースプログラムを参照してメタプログラムを記述することが煩雑あるいは困難であることが予想される．したがってタスクプログラムの本質的な部分だけを簡略化して記述してもモデル精度が保てることを示したことは，ユーザの負担軽減の観点でも有意義であったといえる．

波動方程式では，プロファイルを行うことによって 2〜8 ノードでは誤差を 1/2〜1/3 に削減でき，また 16 ノードでは約 1/14 にまで圧縮することができた．このプログラムではノード数に依存して変化する通信量が性能に大きく影響するが，プロファイルをしない場合のモデルにはこの性質が十分に反映されておらず，ノード数の増加により誤差も単調に増加している．したがってプロファイルによって通信量を正確に把握することで，大幅な精度向上が達成できたものと考えられる．

しかしプロファイルを行った場合でも，絶対的な誤差の値は最大 23%と小さくなく，さらに改善することが必要である．前述のように実行時間の推定には，計算量と通信量をパラメータとする 1 次式を用いているが，この単純なモデル解釈法には改善の余地が多く残されている．すなわち通信の頻度，時間的・空間的分布，計算環境のネットワーク構造などを加味した，より高度なモデル解釈の必要性があるものと考えられる．この 1 つの方法として，複数の実行結果から得ら

☆ OS および他のソフトウェアの構成は 5.1 節に示したものと同一である．
☆☆ 若干向上しているが測定誤差の範囲と考えられる．

れる情報に基づいて，モデル解釈式の係数を最小二乗法によって求めるアルゴリズムを検討している．

この通信コストに関するモデル解釈の問題は，Mandelbrot 集合と波動方程式について，ヘテロ環境での誤差が大きくなっている要因でもあると考えられる．すなわち今回評価に用いたヘテロクラスタでは，2 つのミニクラスタを結合するリンクが通信ボトルネックになるが，現在のモデル解釈ではこのような不均質性を考慮していない．この結果，収集型通信を行う Mandelbrot 集合においてボトルネックの影響が大きく現れ，近接通信を行う波動方程式においても悪影響が生じているものと考えられる．

このネットワーク構造に依存するモデル解釈は，大規模なシステムへの適用の際にも重要な課題になると予想される．MegaScript の 2 階層並列プログラミングモデルでは，下位層である並列タスクの広域的な分散実行は想定していないため，極端な通信ボトルネックが 1 つの並列タスクの実行環境中に生じることは考えにくい．しかし大規模クラスタなどでは，ネットワーク構造が不均質なものとなることは不可避であり，通信の空間的分布も考慮したうえでモデルを解釈する必要が生じると考えている．

また大規模システムへの適用に関する別の課題として，小規模システムを用いて構築したモデルの大規模システムへのスケーリングがあげられる．メタプログラム自体はノード数をパラメータとして記述できるため，生成されるモデルもシステム規模に依存しないものとすることができる．しかし小規模システムで取得したプロファイル情報は，一般にはそのままの形で大規模システムに適用できないため，なんらかのスケーリングを施すことが必要となる．そこで MegaScript プログラムが多数の並列タスクから構成されることを利用し，以下のような方法で大規模システム対応のモデルを構築することを検討している．

(1) 小規模システム対応モデルに含まれるプロファイル情報に対し，ノード数比率を乗じるなどの簡単なスケーリングを施して，大規模システム対応の初期モデルを構築する．
(2) いくつかの並列タスクを，初期モデルに基づいてスケジューリングし，その実行結果に基づくプロファイル情報を収集する．
(3) 収集したプロファイル情報を用いて，大規模システム対応モデルを再構築し，以後のスケジューリングに利用する．

6. 関連研究

並列性を持つタスクプログラムの内部情報の取得やモデリングには，様々な研究が行われている．

分散並列プログラムの性能解析システムの SCALEA[5] は，タスクプログラムを直接解析して構文木を作成する．SCALEA は，ユーザが情報を知るためのコマンドが提供されており，そのコマンドにより，タスクプログラムの内部情報を知ることができる．また，メッセージ通信や並列同期，スケジューリングの際に生じるオーバヘッドの取得が可能であり，幅広い情報取得の機能を備えている．タスクプログラムの言語は，FortranMPI プログラムや OpenMP プログラム，2 つを合わせた Hybrid プログラム，HPF++ プログラムの解析に対応している．SCALEA は，解析情報をスケジューラにフィードバックすることはなく，GUI を用いてユーザに情報を提供することに重きをおいている．

7. おわりに

7.1 まとめ

本論文では，タスク並列スクリプト言語 MegaScript 向けの高精度モデル構築について述べた．具体的には，モデルを記述するメタプログラムの中にプロファイル情報取得を指示する枠組みを設け，それに基づきタスクプログラムにインストゥルメントを行う機構と，取得したプロファイル情報によってモデルを精緻化する機構を実装した．また，モデルの精度の向上を示すため，NAS Parallel Benchmarks の EP，Mandelbrot 集合，波動方程式の 3 つのプログラムを用いて評価を行った．それぞれプロファイル指示情報記述前と後のモデルの精度を比較した結果，EP では最大モデル誤差を 2.8%から 1.5%に，Mandelbrot 集合では 71.9%を 8.5%に，また波動方程式では 71.9%を 22.9%にそれぞれ削減することができた．また，プロファイルにかかる時間もタスクプログラムの実行にほとんど影響を与えないことを示すことができた．以上のことから，広い範囲の性質を持ったプログラムに対応した精度の良いモデルを構築できることを示した．

7.2 今後の課題

今後の課題の主なものとして，以下の項目があげられる．

- 通信コストと計算コストを同時に計算するためのメカニズムを導入することでより安定したモデル構築を実現する．
- タスクプログラムの実行中に動的にモデルの精緻

化を行うことで，短期間のモデル構築を実現する．
- 大規模なクラスタやその集合体に対応してモデル解釈法を改良し，大規模システムでの高精度な性能推定を実現・実証する．

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) 草野和寛，佐藤茂久，佐藤三久：OmniOpenMP 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) 湯山紘史，大塚保紀，西里一史，大野和彦，中島 浩：タスク並列スクリプト言語 MegaScript によるタスクモデルの記述法．*SACSIS 2004*, pp.135–136 (2004).
5) Truong, H-T., Fahringer, T., Madsen, G., Malony, A.D., Moritsch, H. and Shende, S.: On Using SCALEA for Performance Analysis of Distributed and Parallel Programs, *Supercomputing 2001* (2001).

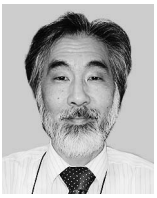
(平成 17 年 1 月 24 日受付)
(平成 17 年 4 月 30 日採録)

湯山 紘史

2004 年豊橋技術科学大学大学院工学研究科情報工学専攻修士課程修了．同年（株）富士通システムソリューションズ入社．在学中は並列プログラムの実行モデルに関する研究に従事．

津邑 公暁（正会員）

1998 年京都大学大学院工学研究科情報工学専攻修士課程修了．2001 年同大学院情報学研究科博士後期課程学修認定退学．同年同大学院経済学研究科助手．博士（情報学）．2004 年豊橋技術科学大学工学部助手．計算機アーキテクチャ，並列処理応用，脳型情報処理等に関する研究に従事．電子情報通信学会，人工知能学会，日本神経回路学会各会員．



中島 浩（正会員）

1981 年京都大学大学院工学研究科情報工学専攻修士課程修了．同年三菱電機（株）入社．推論マシンの研究開発に従事．1992 年京都大学工学部助教授．1997 年豊橋技術科学大学教授．並列計算機のアーキテクチャ等並列処理に関する研究に従事．工学博士．1988 年元岡賞，1993 年坂井記念特別賞受賞．情報処理学会計算機アーキテクチャ研究会主査，同論文誌コンピューティングシステム編集委員長等を歴任．IEEE-CS，ACM，ALP，TUG 各会員．