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

2004－HPC－100 (4)
2004／12／17

タスクネットワークの形状に基づく並列スクリプト言語のスケジューラ

森 英一郎[†] 大野和彦[†] 佐々木敬泰[†]
近藤利夫[†] 中島浩[††]

近年，複雑な物理シミュレーション等の分野では，Pflops 以上の計算能力が求められており，そういった用途向けにタスク並列スクリプト言語 MegaScript の開発を行っている．MegaScript は多数のタスクを並列に実行するための言語であるが，メガスケール環境において最大限に性能を発揮するには，各ホストの負荷バランスやホスト間通信量の削減などを考慮したスケジューリングが必要である．しかし，MegaScript で扱うプログラムはタスク数が膨大であると想定されるため，真に最適なスケジューリングを求めるのは要する演算時間の観点から，現実的ではない．

そこで，実用的な並列プログラムの多くはタスク数が増大しても，タスクネットワーク構造は複雑化しないことに注目した．本稿では，プログラムのタスクネットワーク構造を把握し，その構造に合わせたスケジューリングを行うことを提案する．これによって，比較的短時間で，効率の良いスケジューリングを行うことができる．

A Task Scheduler for a Parallel Script Language Based on the Task Network Structure

EIICHIROU MORI,[†] KAZUHIKO OHNO,[†] TAKAHIRO SASAKI,[†]
TOSHIO KONDO[†] and HIROSHI NAKASHIMA[††]

We are developing a task-parallel script language named MegaScript, which is designed for mega-scale environment. In MegaScript programs, huge number of tasks are executed in parallel. Thus, although load-balancing among hosts and reducing inter-host communication are important, strictly-optimized task scheduling is not pragmatic due to its computation cost.

However, even in the programs using mega-scale number of tasks, the complexity of task network is relatively small. So we propose a new scheduling scheme based on the structure of task network. In our scheme, the scheduler matches the structure with several network patterns and decides the strategy. Thus effective scheduling is possible with small computation cost.

1. はじめに

近年，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等では Pflops 以上の計算能力が求められている．そのため，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．現在，専用並列計算機では数千のプロセッサを利用して数十 Tflops の能力を実現しているが，この延長でメガスケールコンピューティングを実現するためには膨大な電力と巨大な施設が必要となる．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．メガスケール環境はヘテロな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ためのプログラミング環境が必要である．そこで，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」を開発している[1)~3)]．

MegaScript では，並列処理の単位となるタスクを大量に生成し，並列・並行に実行する．このため，システムを構成する各ホストに対しどのようにタスクを割り当て，どのような順序で実行するかという，スケジューリングを考える必要がある．さらに，メガスケール環境において最大限に性能を発揮するには，各ホストの負荷バランスやホスト間通信量の削減などを考慮したスケジューリングが不可欠である．しかし，MegaScript で扱うプログラムはタスク数が膨大であると想定されるため，あらゆる割り当て方を探索して

† 三重大学
Mie University
†† 豊橋技術科学大学
Toyohashi University of Technology

真に最適なスケジューリングを行うのは要する演算時間の観点から現実的ではない．

近似解を求める手法として，笠原らの粗粒度タスク並列処理[4]が挙げられる．この手法では逐次プログラムを粗粒度タスクに分割し，依存関係や並列性を解析した上でプロセッサに割り当てるが，MegaScript ではタスクの内部がブラックボックスである (2.1 章参照) ため，この手法が使えるほどの解析を行えない．

一方，MegaScript には以下の性質が期待できる．

- 明示的な並列言語であるため逐次部分よりも並列部分の方が計算量は大きい．
- タスクネットワークをユーザが明示的に記述するため，タスク数が膨大であってもネットワーク構造はそれほど複雑化しない．

以上の点から，個々のタスクよりネットワーク構造に注目して並列部分を抽出し，適切なスケジューリングを行えば，大きな効果が得られる．

本手法では，まずタスクネットワークを辿って全体構造を把握し，並列部分を見つけだす．次に，その構造に合わせて通信データ量が小さくなるように各ホストに割り当てる．これにより，全探索に比べて短い時間で精度の良いスケジューリングを実行できる．

2. MegaScript の概要

MegaScript は，外部プログラムをタスクとして定義し，その間をストリームと呼ばれる通信路でつないだタスクネットワーク構造を記述するための言語である．

タスクを並列実行させるのに必要なオーバヘッドは全体に対してわずかであるため，実行効率より記述のしやすさを優先し，Ruby をベースとするスクリプト言語としている．

2.1 タスク

タスクは MegaScript の並列実行単位である．タスクの実体は独立性の高い外部プログラムであり，内部の処理に MegaScript は関与しない．

タスクは Task クラスのオブジェクトとして表され，対応する外部プログラムのファイル名や実行時引数などをメンバとして持つ．MegaScript のスケジューリングは，このタスクオブジェクトを操作して行う．

2.2 ストリーム

ストリームは，あるタスクの標準出力の内容を，他のタスクの標準入力に流し込むための通信路である．一つのストリームの入出力にそれぞれ複数のタスクを接続することで，一対多，多対多などの通信を簡潔に実現できる．ストリームの入力端に複数のタスクを接続した場合，メッセージはマージされる．また，出力単に複数のタスクを接続した場合，メッセージは接続した全タスクにマルチキャストされる．

```
t1 = T1.new()
t2 = T2.new_array(M)
t3 = T3.new_array(M*N)
s1 = Stream.new()
s2 = Stream.new_array(M)
s1.connect(t1, IN)
s1.connect(t2, OUT)
for i in 0..M
  s2.connect(t2[i], IN)
  s2.connect(t3[i*N..(i+1)*N], OUT)
end
```

図 1 タスクネットワークの定義例

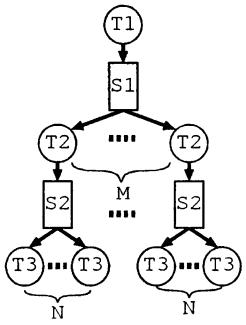

図 2 タスクネットワーク

MegaScript のプログラム (のタスクネットワーク定義部分) の例を**図 1** に挙げる．この記述により，**図 2** のようなタスクネットワークが生成される．

2.3 メタプログラム

適切なタスクスケジューリングを行うには，タスクの計算時間や通信量などの情報をできるだけ正確に知る必要がある．しかし，タスクは任意の言語で記述された外部プログラムなので，MegaScript が常にこれらの情報を取得できるとは限らない．

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を用いる．メタプログラムは元のプログラムの制御構造と計算処理，入出力処理を抽象化して記述するものである (**図 3**)．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高い記述性を持ち，ユーザのタスクプログラムに関する知識や，かけられる手間に応じて，詳細な記述もトップレベルの大雑把な構造のみの記述も可能である．

2.4 メタ情報

処理系はメタプログラムを解析し，タスクごとのメタ情報を得る．具体的には，計算コスト C_c，出力コ

```
WHILE
  input(1)
  FOR @arg[0]
    compute(100)
  END
  output(10)
END
```

図 3　メタプログラムの記述例

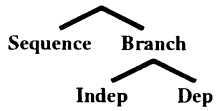

図 4　タスクネットワーク構造の分類

スト C_o を得る．

これらのコストは静的に求まる場合は数値となるが，静的には求まらないケースもある．例えば，ループの回数がタスクの実行時引数によって決まるケースや，入力があるたびに処理を行うようなタスクのケースである．この場合，コスト値はコスト関数として表される．従って C_c，C_o は，数値か，実行時引数と入力コストのコスト関数である．

スケジューラはメタ情報を利用して効率の良いスケジューリングを行う．

2.5　スケジューリング

MegaScript プログラムで生成されたタスク／ストリームオブジェクトは，create メソッドによってスケジューリング待ちキューに移動する．その後，schedule メソッドを呼ぶと，スケジューラはスケジューリング待ちキューにあるオブジェクト全ての配置ホストを決定する．配置されたオブジェクトは，配置先のホストで実行される．

3.　タスクネットワーク構造の分類

ストリームは，ストリームを必要とするタスクと同じホストに配置すればよい☆．従って，スケジューリングにおいてストリームの配置を意識する必要はない．そこで，以下の議論ではストリームを省略し，タスク同士が直接繋がっているものとして考える．

また，一般的にタスクネットワークは環状構造を持つことがあるが，本手法では，そのようなタスクネットワークは環状構造を持たないタスクネットワークに近似してスケジューリングを行う．

逐次構造は，タスクが一対一で接続され，一直線に繋がっているネットワークである．このようなネットワークは，計算と通信を交互に繰り返すようなタスクがなければ，複数のタスクを並列に実行しても高い効果を得られない．

一方，分岐構造は一つのタスクの出力が，複数のタスクの入力に繋がって分岐しているネットワークである．このようなネットワークは，複数のタスクを並列に実行すると，高い効果を得られることがある．並列実行して最も高い効果を得られるのは，分岐した先のタスクが互いに独立していて，通信を行わないケースである．

そこで，並列プログラムのタスクネットワーク構造を図 4 のように分類した．Sequence は逐次構造，Branch は分岐構造である．Branch のうち，分岐した先が互いに独立しているようなタスクの集合を Indep と呼び，分岐した先が非独立で通信が行われるようなタスクの集合を Dep と呼ぶ．

本稿では，並列効果の高い Branch 部分をプログラムのネットワーク構造から探し，並列実行させるようなスケジューリングについて述べる．

4.　本手法の概要

本手法は以下の流れで行う．

(1)　タスクネットワーク構造の把握
(2)　タスクグループの分割
(3)　ホストへの割り当て

まず，タスクの接続関係を調べ，逐次構造と並列構造を見つけだす．

次に，ネットワーク構造を辿りながら，並列構造を分割し，各ホストに割り当てる．逐次構造は基本的には同一ホストに割り当てる．

5.　ネットワーク構造の把握

プログラムのタスクネットワークを辿りながらタスクの接続関係を調べ，並列部分を見つけだすのが目的である．

図 5(1) のようなタスクネットワークでは，入力タスクと出力タスクがそれぞれ同一のタスクであるようなタスク T_i が複数ある場合，T_i の集合が Indep である．

図 5(2) のようなタスクネットワークでは，斜線部分の集合が Indep であるが，T_{1i} から二段下流の T_3 まで読まなければ，全ての分岐が同一のタスクに合流する事 (斜線部分が Indep である事) を知ることはできない．この例では二段下流まで読めば Indep であることがわかるが，どこまで読めばわかるかは，一般的

☆ 必要とするタスクが複数のホスト上に分散した場合は，ストリームのミラーを生成する．

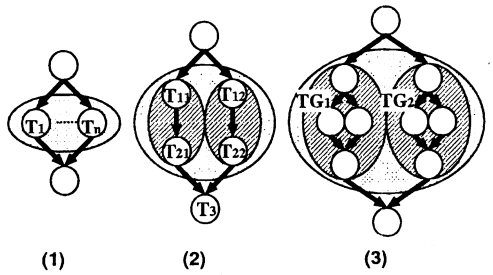

図 5 Indep の例

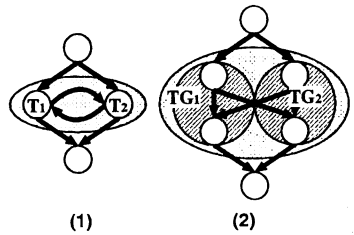

図 6 Dep の例

にはわからない.

また, 図 5(3) のようなタスクネットワークでは, 斜線部分 TG_1 と TG_2 の集合が Indep であると言える.

一方, **図 6**(1) のようなタスクネットワークでは, T_1 と T_2 が互いに通信を行っていて, 独立していない. 従って, T_1 と T_2 は Dep である.

また, 図 6(2) のようなタスクネットワークでは, 斜線部分 TG_1, TG_2 間で通信を行っており, TG_1, TG_2 は Dep であると言える.

5.1 グループ化とタスクグループ

実用的な並列プログラムのタスクネットワークは, 図 5(2),(3) や, 図 6(2) のような入れ子構造になっていることも多い. そこで,「グループ化」という考えを導入し, ネットワーク構造の認識を階層的に行う.

タスクグループとは, Sequence か, Indep か, Dep のネットワークを構成するタスクを一つにまとめたものである. グループ化とは, タスクグループを作り, タスクグループに「タスクグループの種類 (Sequence か, Indep か, Dep か, という情報)」を付加する処理である. また, タスクグループを構成するタスク (またはタスクグループ) を, タスクグループの要素タスクという.

グループ化された要素タスクをタスクグループで置き換え, そのタスクグループを一つのタスクと同様に扱うことで, 階層的なネットワーク構造の認識を行う.

5.2 タスクグループの種類

タスクグループの種類別に, 認識方法とグループ化

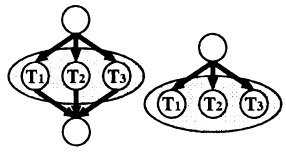

図 7 Indep

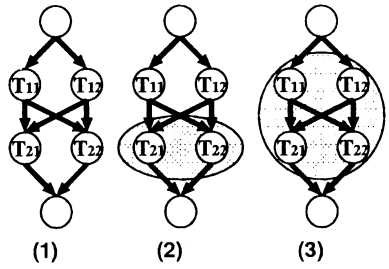

図 8 Dep

方法について説明する.

5.2.1 Sequence

分岐又は合流した直後から, 分岐又は合流する直前まで, 一直線に繋がっているタスクをグループ化する.

5.2.2 Indep

- タスクの入力と出力がそれぞれ唯一のタスクと接続されていて, 出力タスクの入力タスクが他になく, しかも出力タスクが Dep でない (**図 7**(1)).
- タスクの入力が唯一のタスクと接続されていて, 出力には何も接続されていない (図 7(2)).

以上のようなタスクをグループ化する.

5.2.3 Dep

- 注目タスクの入力と出力がそれぞれ唯一のタスクと接続されていて, 出力タスクの入力タスクが注目タスク以外にもある (**図 8**(2)).
- 注目タスクの入力と出力がそれぞれ唯一のタスクと接続されていて, 出力タスクの入力タスクが他になく, 出力タスクが Dep であるようなタスク (図 8(3)).

以上のようなタスクをグループ化する.

5.3 階層的なグループ化の例

階層的なグループ化では, 内側のタスクグループから外側のタスクグループに向かってボトムアップ的にグループ化を行う. 具体的にはタスクネットワークを深さ優先探索で辿り, 探索中, 下から上に戻るときに, 下のネットワークがどんな構造をしているのか, 判定を行う. その結果, 最終的には並列プログラム全体のタスクネットワークが一つのタスクグループになる.

例えば**図 9** では, 以下のようにグループ化が行わ

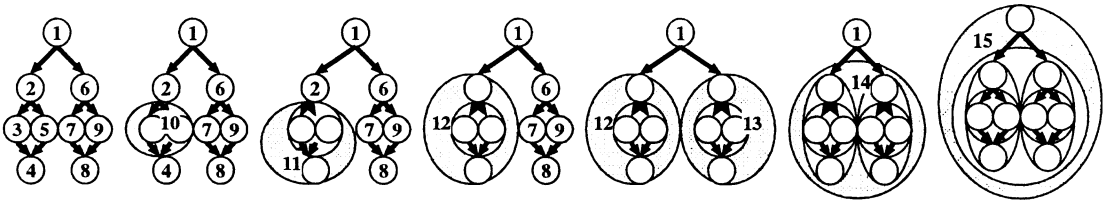

図 9 階層的なグループ化の例

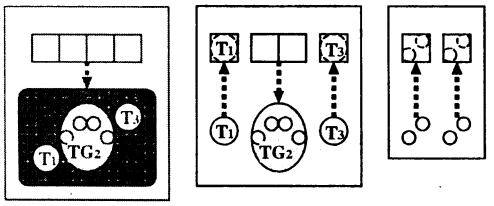

図 10 タスクグループの分割

れる.

(1) タスク 3,5 が Indep と認識され，タスクグループ 10 が作られる.

(2) タスク 10,4 が Sequence と認識され，タスクグループ 11 が作られる.

(3) タスク 2,11 が Sequence と認識され，タスクグループ 12 が作られる.

(4) タスクグループ 12 と同様の手順で，タスクグループ 13 が作られる.

(5) タスク 12,13 が Indep と認識され，タスクグループ 14 が作られる.

(6) タスク 1,14 が Sequence と認識され，タスクグループ 15 が作られる.

6. タスクグループの分割

タスクグループの分割は，あるタスクグループの要素タスク集合を，与えられた分割目標数に分ける処理である．分割目標数と同じだけの「枠」を導入し，ある要素タスクに対して，いくつの枠を割り当てるのかを決定する作業とも言える.

各々の枠に割り当てられた要素タスクはそれぞれ異なるホストに配置されることになるため，以下の事が分割の基本方針となる.

(A) 枠の間の通信コストはなるべく少なくする

(B) 並列部分の計算コストはなるべく均等にする

タスクグループの種類によって，分割方法は異なる．また，あるタスクグループの分割処理中に，その要素タスクの分割を必要とすることがあるが，その場合は再帰的に要素タスクの分割処理を行う.

図 10 は，T_1，TG_2，T_3 からなるタスクグループを四個に分割する例である．(1) で元のタスクグループに四個の枠を与え，(2) で T_1，T_3 にそれぞれ一つずつ，TG_2 に二つの枠を割り当てている．TG_2 はさらに分割され，(3) で TG_2 の要素タスクを二つに分割し，それぞれ枠に割り当てている.

6.1 分割条件

タスクグループ TG の分割処理中，要素タスク T_i をこれ以上分割すべきかどうかを判断する事がある．この判断に，次の式を使う.

$$\alpha < TG\_num \qquad (1)$$

α : しきい値

TG_num : $num \times C_c(T_i) \div C_c(all)$

num : TG に与えられた分割目標数

$C_c(T_i)$: T_i の計算コスト C_c

$C_c(all)$: TG の要素タスクの C_c の合計

TG_num は，TG に与えられた num 個の枠のうち，T_i にいくつの枠を与えればいいのかを，計算コスト比で求めたものである．TG_num がしきい値 α を越えていれば，T_i は計算コストが大きすぎ，TG_num 個に分割すべきだと判断できる.

6.2 Sequence の処理

要素タスクが一つのタスクなら，特定の枠に詰める．要素タスクが Indep か Dep なら，その要素タスクを分割目標数に分割し，別々の枠に一つずつ詰めていく.

図 11 の例では，Sequence 中の TG_2 を T_{21}，T_{22}，T_{23} に分割し，全ての枠に割り当てている.

6.3 Indep の処理

Indep では要素タスク間の通信はないため，方針 (A) については特に考える必要がない．従って，方針 (B) の，計算コストをなるべく均等にすることだけを考えればよい.

要素タスクが 6.1 章の分割条件式 (1) を満たすかどうか調べ，満たした要素タスクは分割する．その後，キューの計算コストをなるべく均等にするように割り

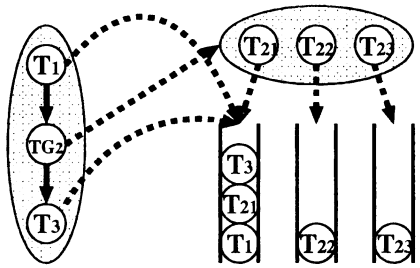

図 11 Sequence の処理

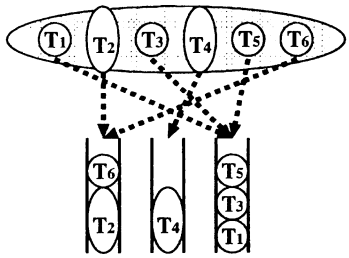

図 12 Indep の処理

当てる．

図 12 の例では，計算コストの大きい T_2，T_4 と，計算コストの小さい T_1，T_3，T_5，T_6 を，枠の計算コストがなるべく均等になるように割り当てている．

6.4 Dep の処理

できるだけデータフロー方向に沿って分割し，かつ，枠間の通信コストも小さくする．そのために，最上流タスクから下流のタスクに向かって，最も通信の多い出力タスクを選びながら塊にしていく．その塊を，通信コストが小さくなるように枠に割り当てる．

図 13 は，真下のタスクへの通信コストが，斜め下のタスクへの通信コストよりも大きな場合の分割の例である．

7. 動的スケジューリング

これまで述べてきたのは，schedule メソッドが呼ばれたときに行う静的スケジューリングについてである．この静的スケジューリングは，全てのメタ情報が正確に判明している場合は有効である．しかし，メタ情報の精度はメタプログラムの記述具合によって変化するため，メタ情報が不完全で誤差が生じることがある．

そこで，あるホストで実行できるタスクがなくなった時点で動的なスケジューリングを行い，ホスト間のタスク処理進度の差を吸収する．

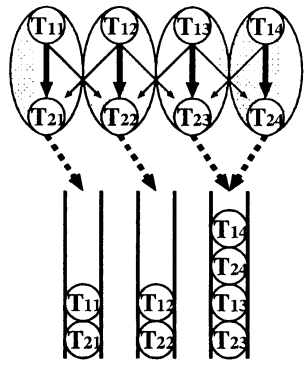

図 13 Dep の処理

8. おわりに

本稿では，並列スクリプト言語 MegaScript 用のタスクスケジューリング手法として，プログラムのタスクネットワーク構造の情報を利用したスケジューリングについて述べた．本手法は，一対一や一対多対一，木構造などの単純なタスクネットワークや，正確なメタ情報を得られる場合は効率的な実行が可能である一方，そうでない場合にも動的に対処できる．

現在，このスケジューラの実装を進めている．実装作業が終わり次第，評価を行う予定である．

また，本稿では均質な実行環境でのスケジューリングについて説明したが，広域分散環境下では，ホストが持つ処理能力はそれぞれ異なっているのが一般的である．比均質な実行環境や広域分散環境への対応も，今後の課題である．

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) 大塚保紀, 大野和彦, 中島浩: タスク並列スクリプト言語 MegaScript によるタスク動作モデル記述, 情処研報 2003-HPC-95, pp. 113–118 (2003).
3) 西里一史, 大野和彦, 中島浩: タスク並列スクリプト言語 MegaScript のランタイムシステムの設計と実装, 情報処理学会研究報告 HPC-95, pp. 119–124 (2003).
4) 笠原博徳, 小幡元樹, 石坂和久: 共有メモリマルチプロセッサシステム上での粗粒度タスク並列処理, 情報処理学会論文誌, Vol. 42, No. 4 (2001).