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

2006－HPC－107（11）
2006／7／31

タスクネットワークの解析情報を用いたスケジューリング手法

片野　聡[†]　森　英一郎[†]　大野和彦[†]
佐々木　敬泰[†]　近藤利夫[†]　中島　浩[††,☆]

我々はメガスケールの並列処理を想定したタスク並列スクリプト言語 MegaScript を開発している．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現実的ではない．

本稿では，まず，典型的なネットワーク構造の組み合わせでタスクネットワークを表現し，タスク間のパイプライン並列性を判定するためのモデル化を行う．さらに，タスク間依存によるコストを抽出し，待ち時間の推定を行う．これらの情報を用いて，効率の良いスケジューリングが出来ることを目指す．

A Task Scheduling Scheme Using Analytical Information on Task Network

Satoshi Katano,[†] Eiichirou Mori,[†] Kazuhiko Ohno,[†]
Takahiro Sasaki,[†] Toshio Kondo[†] and Hiroshi Nakashima[††,☆]

We are developing a task-parallel script language named MegaScript. To obtain high performance in mega-scale environment, scheduling scheme considering load-balancing and communication cost is required. However, optimal scheduling is difficult because number of scheduled tasks is extremely large.

In our scheduling scheme, a task network model is represented as composition of basic structures. Based on this model, we extract pipeline parallelism and dependency costs. Using the result, efficient scheduling is possible.

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多くのプロセッサを利用して数百 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．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．

そこで我々はタスクネットワーク構造のモデル化を行い，その上で並列実行可能部分を抽出し，それらを並列実行するスケジューリングを提案している[4)]．

しかし，従来手法ではモデルの精度が不十分であり，

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

プログラムによっては適切なスケジューリングが行えない場合がある．そこで本稿では，ループ構造とパイプライン並列性を認識できるようにモデルを改良し，さらに依存関係を考慮したスケジューリングを行う手法を提案する．

これにより，従来手法と比べて，精度の良いスケジューリングが期待できる．

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[5] をベースとするスクリプト言語としている．

2.1 タスクとストリーム

タスクは MegaScript の並列実行単位である．タスクの内部の処理に MegaScript は関与しない．

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．

タスクとストリームは，それぞれ Task と Stream クラスのオブジェクトとして表される．MegaScript のスケジューリングは，このオブジェクトを操作して行う．

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．

2.2 メタプログラムとメタ情報

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[2]．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

```
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

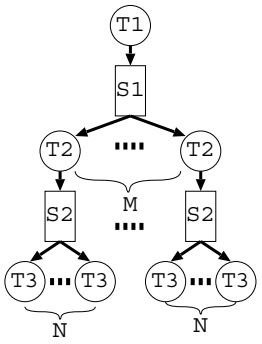

図 2　タスクネットワーク

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

図 3　メタプログラムの記述例

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．図 3 に，メタプログラムの記述例を示した．

メタプログラムを静的解析して得られた情報から，メタモデルが生成される．メタモデルはタスク毎に存在し，対応するタスクのメタ情報を持つ．具体的には，計算コスト C_c，入力コスト C_i，出力コスト 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．

図 3 の場合，C_c は $100 \times arg[0]$ として実行時引数の関数として得られ，C_o は 10 という数値が得られる．

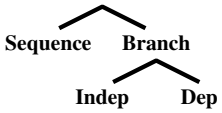

図 4　タスクネットワーク構造の分類

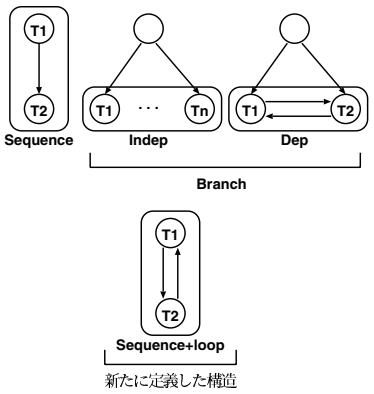

図 5　タスクネットワークの基本構造

2.3　スケジューリング

スケジューラはタスクネットワークの解析情報を利用して効率のよいスケジューリングを行う．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，ユーザが create メソッドを呼ぶと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と実行順を決定する．配置されたオブジェクトは，決定した順番通りに，配置先のホストで実行される．

3.　従来手法の問題点と解決方針

3.1　従来手法の手順と問題点

従来手法は以下のような手順で行われている．

(1)　タスクネットワーク構造を認識する．
(2)　スケジューリングを行う．

タスクネットワークの構造の認識は，個々のタスクについて，データフロー上で直前のタスクとの接続形状により分類することで行う．なお，スケジューリングはネットワーク構造に合わせ，計算コストが均等になるように配置ホストを決定する．しかし，以下のような問題点が存在する．

- ループのあるネットワーク構造は，ループのないネットワーク構造に近似しているため，タスク間依存を正確に認識できない．
- 相互通信のないタスク同士の並列性は認識できるが，通信があってもパイプライン並列性が期待できる構造を認識できない．
- タスク間依存に従いデータフロー順に実行するが，依存による待ち時間を考慮していないため，すぐに実行できないタスクを生成してしまう可能性がある．

3.2　提案手法の手順

従来手法の問題点を解決するため，提案手法は以下のような手順で行う．

(1)　ループが認識可能なモデルで，タスクネットワーク構造を認識する．
(2)　入出力のタイミングによりタスクをいくつかのパターンに分類し，パイプライン並列性の有無を調べる．
(3)　依存関係により，待たされる時間 (依存コスト) を算出する．
(4)　スケジューリングを行う．

4.　提 案 手 法

4.1　タスクネットワーク構造のモデル化

ストリームは，基本的に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．よって，ここではストリームのスケジューリングを考えないこととする．そこで，以下の議論ではストリームを省略し，タスクの入出力が直接繋がっているものとして考える．MegaScript のネットワークは，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と期待できる．このことを利用し，典型的なネットワーク構造をいくつか定義しておき，それらの組み合わせで MegaScript のネットワークを表現することにより，単純化が可能となる．

従来は，図 4 のように分類していた．Sequence は逐次構造で，二つのタスクが一方向の通信で繋がっているものである．Branch は，分岐構造であり，そのうち，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 (図 5 上段)．

また，ループ構造を近似していたため，依存関係を正しく認識できない場合があった．提案手法ではループ構造も認識できるように，従来の分類に加えて Sequence+loop を定義する (図 5 下段)．

Sequence+loop は，Sequence 内にループが現れたものであり，直列に繋がっているのに加え，双方向で通信を行なうものである．

4.1.1　ネットワーク構造の把握とグループ化

実用的な並列プログラムのタスクネットワークは，基本構造の入れ子で表現できる．そのため，一つの基本構造で表現できるタスク群を一つにまとめたものを

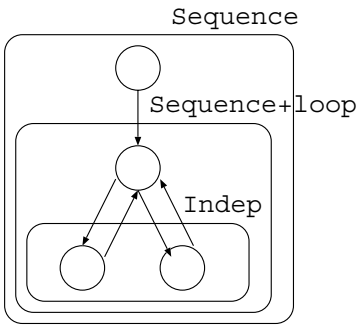

図 6　タスクネットワークのグループ化

タスクグループと呼び，タスクグループを作ることをグループ化と呼ぶことにする．このグループ化の作業を繰り返し，最終的には MegaScript プログラム全体のタスクネットワークを一つのタスクグループにする．

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，どんな構造をしているのか，判定を行い，階層的にグループ化する．なお，ループの存在により，探索中に訪問済みのタスクへの接続があった場合は訪問しない．

タスクグループの種類別に，認識方法とグループ化方法について説明する．図 6 は，グループ化されたネットワークの一例である．

- Sequence
  逐次構造の部分で，注目タスクグループと直前のタスクグループの接続が一方向である場合，二つをまとめて Sequence とする．
- Indep
  分岐先のタスクグループが，お互い通信しないなど，独立しているものを Indep としてグループ化する．
- Dep
  分岐先のタスクグループが，非独立であるものを Dep としてグループ化する．実際の認識方法としては，Indep でないものを Dep として扱う．
- Sequence+loop
  逐次構造の部分で，注目タスクグループと直前のタスクグループの接続が双方向である場合，二つをまとめて Sequence+loop とする．

4.2　パイプライン並列性

パイプライン並列性とは，主に通信と計算を交互に行ない，並行に実行することができるタスクである．パイプライン並列性の有無を判定することにより，スケジューリングする際に，並行実行するか逐次実行するかの区別を行うことが可能になる．図 7 の (1) では，T1 の終了近くでデータを 1 度だけ T2 に渡しており，T1 が終了するまで T2 が実行できない．しかし，図 7 の (2) では，T1 と T2 が交互に通信しあい，並行実行させなければ動作しない．

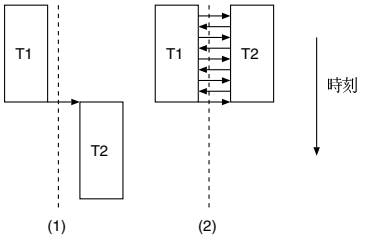

図 7　パイプライン並列性の有無

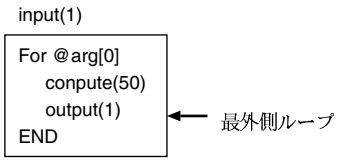

図 8　最外側ループの例

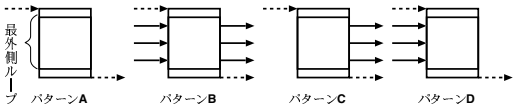

図 9　タスクの分類

4.2.1　通信パターンによる分類

タスクがパイプライン並列性を持つかどうかは，そのタスクの入出力処理が，タスクの生存期間内でどのように分布するかで判別できる．MegaScript のタスクはブラックボックスでプログラムを直接解析できないため，メタプログラム内の入出力処理の分布を調べる．今回は図 8 のようなメタプログラムの最外側ループがタスクの生存期間の大部分を占めるとみなし、最外側ループ内に Input や Output が出現するか否かで以下の 4 パターンに分類した (図 9)．

- パターン A: 入出力なし
- パターン B: 入出力あり
- パターン C: 出力あり
- パターン D: 入力あり

なお，多重ループ内の処理は，最外側の処理として扱う．最外側ループが複数ある場合は，融合して判定する．このように分類した際，タスクネットワークのループ構造になっていない，かつパターン B，D である場合は，一つ上流のタスクとパイプライン並列性があると判定する．ループ構造になっている場合は，パターン C である場合もループ先のタスクとパイプラ

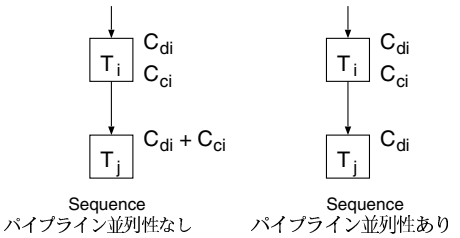

図 10　Sequence の依存コスト

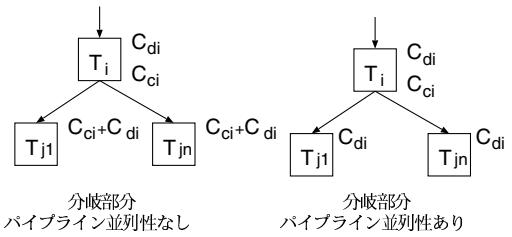

図 11　分岐部分の依存コスト

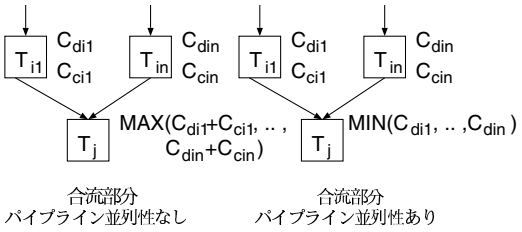

図 12　合流部分の依存コスト

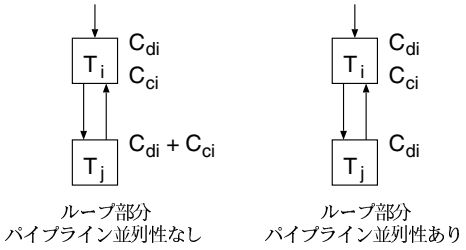

図 13　ループ部分の依存コスト

イン並列性があると判定する．例えば，図 8 の場合は，パターン C に分類される．

4.3　タスク間の依存の考慮

従来手法では，タスクをデータフロー順に実行することで，依存による待ち時間の減少を期待し，スケジューリングの際は，計算コストの均等化を重視していた．しかし，待ち時間の大きさを考慮していないため，実際にタスクを生成しても，依存していたタスクがまだ実行中であるため，アイドル時間が発生してしまうことがあった．今回は，依存コストというものを新たに導入することで，その問題を解決することを考える．

4.3.1　依存コスト

あるタスクを実行するためには，そのタスクが依存しているタスクすべてを実行しなければならない．つまり，依存先タスクの計算コストにより注目タスクの待ち時間が決まることになる．依存先タスクについても同様の考えがなりたつから，依存をたどっていくことで，プログラムの実行開始から注目タスクを実行できるまでの待ち時間が求められる．これを依存コストと呼ぶことにする．

4.3.2　依存コストの算出方法

計算コストを C_c，依存コストを C_d とする．以下，本稿の図ではタスクの右上に依存コスト，右下に計算コストを表記する．

- Sequence

  図 10 は，Sequence における依存コストの伝搬を示したものである．パイプライン並列性がないときは，T_i の実行が終了するまで T_j が実行できないと考え，T_i の依存コストに T_i の計算コストを加えたもの $C_{di} + C_{ci}$ が T_j の依存コストとなる．パイプライン並列性がある場合は，T_i と T_j が同時実行でき，依存はないと考え，T_i の依存コストがそのまま T_j の依存コストとなる．

- 分岐

  図 11 は，分岐部分における依存コストの伝搬を示したものである．考え方は，Sequence と同じで，パイプライン並列性がないときは，T_i の実行が終了するまで $T_{j1} \ldots T_{jn}$ は実行できないと考え，T_i の依存コストに T_i の計算コストを加えたものが $T_{j1} \ldots T_{jn}$ の依存計算コストとなる．パイプライン並列性がある場合は，T_i と $T_{j1} \ldots T_{jn}$ が同時実行でき，依存はないと考え，T_i の依存コストがそのまま $T_{j1} \ldots T_{jn}$ の依存コストとなる．

- 合流

  図 12 は，合流部分における依存コストの伝搬を示したものである．合流の場合は，複数のタスクに依存している場合があるので，考え方が異なる．パイプライン並列性がない場合は，$T_{i1} \ldots T_{in}$ が終了してから T_j が実行可能と見て，$T_{i1} \ldots T_{in}$ のそれぞれの依存コストと計算コストの和 $C_{di1} + C_{ci1}, \ldots, C_{din} + C_{cin}$ を求め，最も大きいものを T_j の依存コストとする．パイプライン並列性がある場合は，$T_{i1} \ldots T_{in}$ と T_j が同時実行可能であると考え，$T_{i1} \ldots T_{in}$ の依存コストを比べ，最も小さいものを，T_j の依存コストとする．

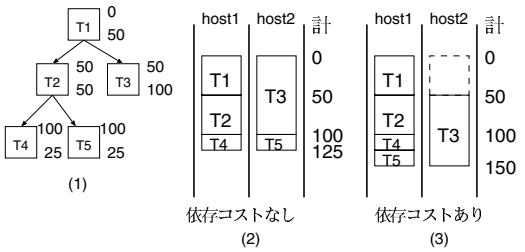

図 14　依存コストを用いたスケジューリングの例

- ループ

  図 13 は，ループ部分における依存コストの伝搬を示したものである．ループの場合は，Sequence と考え方は同じである．

4.4　スケジューリング

従来の手法どおり，なるべく計算コストを均等にする方法をとる．依存コストを考慮に入れる方法としては，注目タスクの依存コストと配置候補のホストの計算コストの合計と比較し，依存コストのほうが大きければ，計算コストの合計と置き換える．その後，計算コストの合計に注目タスクの計算コストを加えるという手順で行う．

例として、図 14(1) のタスクネットワークを考える．なお，パイプライン並列性はないものとする．

従来の依存コストを考慮しない方法での結果が図 14(2) である．計算コストは均等になっているが，T3 は T1 に依存しているため，T1 が実行終了するまで，実行できない．従って，host2 では，T1 が終了するまでアイドル時間が発生する．

提案する依存コストを考慮する方法での結果が図 14(3) である．依存コストを考慮すると，T3 まで配置した時点で，host1 と host2 の計算コストの合計はそれぞれ 100 と 150 となる．よって，計算コストを均等に配置すると，T4 と T5 の配置は host1 となる，

結果として，(1) は終了まで 175 かかり，(2) は 150 かかる．従って，(2) のほうがスケジューリングの精度が良いことになる．このように，依存コストを導入すれば，計算コストの見積もりがより正しくなり，スケジューリング精度の向上が期待できる．

5. 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．タスクネットワークの構造をより正確に認識し，パイプライン並列性や依存を考慮したスケジューリングを行うことにより，従来よりスケジューリング結果が良くなることを期待している．

このスケジューリング手法の妥当性の検証を進め，結果が良好であれば実装と評価を行う予定である．現在は均質な実行環境でのスケジューリングを想定しているが，広域分散環境など非均質な実行環境でのスケジューリングも今後考えていきたいと思っている．

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, 情報研報 2003-HPC-95, pp. 119–124 (2003).
4) 森 英一郎, 大野 和彦, 佐々木 敬泰, 近藤 利夫, 中島浩: 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, 情処研報 2004-HPC-100, pp. 19–24 (2004).
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).