情報処理学会論文誌　プログラミング　Vol. 2　No. 1　1–17 (Jan. 2009)

ヘテロ型大規模並列環境の階層型タスクスケジューリングの提案と評価

松本真樹[†1]　片野聡[†1,*1]　佐々木敬泰[†1]
大野和彦[†1]　近藤利夫[†1]　中島浩[†2]

我々は大規模な広域分散環境における並列処理を目的とした，タスク並列スクリプト言語 MegaScript を開発している．本言語による並列処理で高性能を達成するには，依存のあるタスク群を非均質環境に効率良くスケジューリングする手法が必要である．しかし，高精度な静的スケジューリングは計算コストが非常に高く，タスクが完全独立でないため単純な動的スケジューリングでは高性能が期待できない．そこで，階層型にスケジューラを配置し，複数の方式を併用するハイブリッド型のスケジューリング方式を提案する．本手法では実行環境のホスト群およびスケジューリング対象であるタスクネットワークを階層モデル化し，階層型スケジューラによる段階的なスケジューリングを行う．上位スケジューラでは抽象モデルによる高速な大域スケジューリングを行い，下位スケジューラでは個々のタスク・ホストに対する詳細な局所スケジューリングを行うことで，大規模なスケジューリングを効率良く扱うことができる．また，動的スケジューリングを併用することで，実行環境の性能変動などによる影響を補正する．抽象シミュレーションにより本手法の評価を行った結果，均質環境用の詳細スケジューリングと比べ，階層型スケジューリングでは 7–10 倍程度の速度向上が得られ，タスク数/ホスト数が 10,000/1,000 規模の場合，約 1/44 の時間でスケジューリングを行うことができた．

A Hierarchical Scheduling Scheme for Large Scale Heterogeneous Environments

Masaki Matsumoto,[†1] Satoshi Katano,[†1,*1]
Takahiro Sasaki,[†1] Kazuhiko Ohno,[†1]
Toshio Kondo[†1] and Hiroshi Nakashima[†2]

We are developing a task parallel script language *MegaScript* for parallel processing on large-scale widely-distributed environments. To achieve high performance, MegaScript requires a scheduling scheme, which efficiently schedules dependent tasks to a heterogeneous environment. However, the scheduling overhead of existing static schemes are too large, and dependency among tasks reduces efficiency of simple dynamic scheduling schemes. Therefore, we propose a hybrid scheduling scheme, which uses multiple schemes. To reduce the overhead of static scheduling, we use multi-layered schedulers: the upper layer makes rough global scheduling, and the lower layer makes precise local scheduling. We also use dynamic scheduling scheme to compensate the effect of dynamic change of the system performance. The evaluation of our scheme using abstract simulation achieved 7–10 times speedup using multi-layered schedulers. The scheduling time of 10,000 tasks/1,000 hosts was approximately 1/44 compared to a precise single scheduler.

1. はじめに

近年，大規模計算の需要がますます増大する一方で，単一プロセッサの性能向上が頭打ちになり，並列処理への要求が高まってきている．とくに，広域ネットワーク上に分散した性能の異なる計算機群を利用して高性能を達成できれば，コストパフォーマンスやスケーラビリティの面で大きな利点がある．我々はこのような環境で大規模な並列処理を容易かつ効率的に行うために，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，並行・並列に実行する．このため高い実行性能を得るには，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適切に行う必要がある．広域分散環境の性能を最大限に発揮するには，各ホストの負荷バランスやホスト間通信時間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．

MegaScript ではタスク間に依存関係があるため，計算量の均衡化のみを目的とした動的スケジューリングでは，十分な効率を得ることができない．しかし，大規模なタスク数・ホスト数を想定しているため，従来提案されてきた高精度の静的スケジューリングは，計算コストが高く現実的ではない．さらに，スケジューリングに用いる情報の不完全性や実行環境の性能変動といった不確定要因があるため，高精度の静的スケジューリング自体が非常に難

†1 三重大学大学院工学研究科
Graduate School of Engineering, Mie University
†2 京都大学学術情報メディアセンター
Academic Center for Computing and Media Studies, Kyoto University
*1 現在，株式会社デンソー
Presently with Denso Co., Ltd

しい．

そこで本論文では，静的・動的スケジューリングを併用すると同時に，複数のスケジューラを階層的に配置してスケジューリング処理を分散させ，上位層と下位層で異なる方式を用いるという，ハイブリッド型のスケジューリング手法を提案する．

以下，2 章で背景である MegaScript の概要とスケジューリング上の性質について述べ，3 章で既存のスケジューリング手法を概観して我々の目的に対する適合性を議論する．4 章で提案手法の詳細を述べ，5 章でシミュレーションによる本手法の評価結果を示し，最後に 6 章でまとめを行う．

2. 背　　景

2.1 並列スクリプト言語 MegaScript

2.1.1 言語の概要

MegaScript は 2 階層並列モデルの上位層を記述するための言語である．逐次または並列の独立したプログラムを計算タスクとして扱い，これらのタスクを並列実行させる．各タスクは並行並列に動作し，ストリームと呼ばれる通信路を介することでタスク間のデータ受け渡しを行う．

計算のコアな部分は外部プログラムとして用意するため，MegaScript プログラム内には主に並列実行に関する制御情報を記述する．実行制御に要する計算量は全体に対してわずかであるため，実行効率より記述性や拡張性を優先し Ruby [4] をベースとするオブジェクト指向スクリプト言語としている．

2.1.2 タスクとストリーム

タスクは MegaScript とは独立したプログラムであるため，ユーザは任意の言語でタスクを作成することができる．このため，既存プログラムを部品として流用したり，処理内容に応じて記述言語を変えたりするといったことが自由にできる．また，MegaScript はタスク内部の処理に関与せず，タスク間の情報のやりとりには標準入出力を利用し，行単位で 1 つのアトミックなメッセージと見なす．

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り，MegaScript におけるタスク間通信を実現する．

ストリームの入出力端にはそれぞれ複数のタスクを接続することができ，1 対多，多対多などの通信を簡潔に記述することができる．入力端に複数のタスクを接続した場合，メッセージは非決定的にマージされる．また，出力端に複数のタスクを接続した場合は，メッセージはそれぞれのタスクにマルチキャストされる．

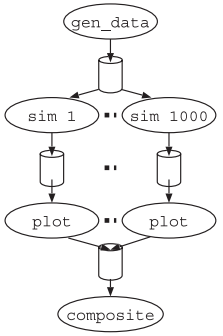


図 1　タスクネットワークの例
Fig. 1　Example of task network.

MegaScript 上では，タスクやストリームの生成・操作は，それぞれ `Task`，`Stream` クラスを用いて行う．同じ種類のタスク/ストリームを複数生成する場合は，それぞれ `TaskArray`，`StreamArray` クラスを用いて，タスク配列/ストリーム配列として生成でき，接続などの操作を一括して行うことができる．

2.1.3 プログラミングモデル

MegaScript では，タスクを並行並列に実行し，その間をストリームで接続し合う，タスクネットワークの形状を記述する．例として，プログラム `gen_data` が生成したデータに対し，プログラム `sim` で引数 1〜1,000 のパラメータサーベイを行った後，それぞれの結果をプログラム `plot` により可視化し，最後にプログラム `composite` で 1 つの画像に合成するという処理を考える．この処理は**図 1** に示すようなタスクネットワークとして表現できる．

また，そのプログラムは**図 2** のように記述できる．MegaScript プログラムでは，まずタスクネットワークの構成に必要なオブジェクトを生成し，`connect` メソッドによって，それらの間の結合を定義する．その後，`create` メソッドによってプロセスなどの実体を生成する許可を与える．最後に `schedule` メソッドを呼ぶことで，生成許可済みのオブジェクトに対するスケジューリングが行われ，タスクの実行が開始される．

2.1.4 メタ情報とメタプログラム

タスク単位の並列性記述やタスク間のデータフローは，抽象度が高く実行環境に依存しないため，ユーザにとって比較的記述上の負担が小さい．一方で，タスクの実行ホスト・実行

```
N = 1000
t1 = Task.new("gen_data")
t2 = TaskArray.new(N, "sim", 1..N)
t3 = TaskArray.new(N, "plot")
t4 = Task.new("composite")
s1 = Stream.new
s2 = StreamArray.new(N)
s3 = Stream.new
s1.connect(t1, IN)
s1.connect(t2, OUT)
s2.connect(t2, IN)
s2.connect(t3, OUT)
s3.connect(t3, IN)
s3.connect(t4, OUT)
t1.create
t2.create
t3.create
t4.create
s1.create
s2.create
s3.create
Scheduler.new.schedule
```

図 2　タスクネットワークのコード例
Fig. 2　Example code of task network.

順序を決めるスケジューリングや，実行を開始したタスクプロセス間でメッセージを中継する手順などは，実行環境に合わせた最適化が必要であり，ユーザが直接制御するのは非常に難易度が高い．このため MegaScript では，前者はユーザが明示的に記述し，後者は処理系内で自動的に行うようにしている．

とくにタスクのスケジューリング方法は並列実行効率を大きく左右するが，MegaScript のようにタスク間に依存がある場合，個々のタスクの計算量や通信量といった情報を利用しなければ，良いスケジューリングが困難である．このような，タスクそのものの実行には不要だがスケジューリングの際に有用な情報を，本論文ではメタ情報と呼ぶ．

メタ情報を得るためにタスクプログラムを静的解析する方法も考えられるが，MegaScript のタスクは任意の言語で記述できるので，この方法がつねに可能とは限らない．そこで，MegaScript ではメタプログラムという形で，タスクのメタ情報をユーザが記述できるようにしている．

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

図 3　メタプログラムの記述例
Fig. 3　Example of meta program.

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[1),2)]．プログラムの形式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．**図 3** に，メタプログラムの記述例を示す．

FOR のような抽象制御構文および `input()`，`compute()` のようなメタ関数は，引数にコスト値をとる．コスト値は単位を持たず，ユーザが考えやすい単位で表現できる．計算コストや通信コストの間のスケーリングは，スケジューリング時に実行環境などに合わせて行われる．また，コスト値には具体的な数値のほか，変数を含む式を記述できる．

MegaScript 処理系は，メタプログラムを静的解析し，得られた情報からメタモデルを生成する．メタモデルはタスクごとに存在し，対応するタスクのメタ情報を持つ．具体的には，メタモデルより計算コスト C_c，入力コスト C_i，出力コスト C_o を得ることができる．ループの回数がタスクの実行時引数によって決まる場合や入力ごとに処理を行うようなタスクの場合，静的にはコスト値が決まらないため，コスト関数として表される．図 3 の場合，C_c はタスクの第 1 引数の関数 $100 \times$ `@arg[0]`，C_o は値 10 となる．

実行時にスケジューラは，タスクに与えられた実行時引数の値や入出力コストの伝搬結果を用いて，コスト値を決定する．

また，タスクの実行時プロファイリングをメタプログラムのコスト値に反映することで，メタモデルの精度を向上させることもできる[2)]．

2.2　MegaScript のタスクスケジューリング

MegaScript は，タスクの組合せによってユーザが大規模な並列処理を記述し，それを 1 つのプログラムとして実行することを想定している．したがってスケジューラは，MegaScript プログラム全体の実行時間，つまりタスク全体のスケジューリング長を最小化することを目標とする．

MegaScript はタスク間が相互通信し合うようなストリーム接続も可能であるが，本論文では議論を簡単にするため，タスクネットワークが DAG の場合のみを考える[⋆1]．また，MegaScript ではタスクやストリームの動的な追加も可能であるが，今回はすべてのタスクとストリームを生成してから 1 度にスケジューリングを行う場合に限定して議論する．

以上の制限によって，今回扱うのは一般に DAG のスケジューリングとして扱われる問題に近いものになる．3 章で述べるように，この分野は多くの研究が行われているが，MegaScript のスケジューリングでは，以下の特徴を考慮する必要がある．

(1) タスクはネイティブプログラムであり，一般に分割やマイグレーションができない．
(2) 一般の DAG スケジューリングは，タスクの実行終了後に後続タスクに対して通信を行うモデルになっている．しかし MegaScript のストリーム通信はタスクの実行中にも可能なので，タスク間のパイプライン並列性を抽出することが可能である．
(3) 実行環境として広域分散するホスト群を想定しており，スケジューリング対象となるホストは計算・通信性能ともに非均質なヘテロ環境である．また，現実的な広域分散環境では，各ホストの計算・通信性能は一定ではなく，他プロセスの負荷などにより変動する．
(4) MegaScript では，スケジューリングの基準となるメタモデル自体の精度が 100%とは限らない．
(5) 大規模な並列処理を目的としており，タスクやホストの数が 10 万〜100 万規模まで扱えることが求められる．
(6) 自動並列化コンパイラが逐次プログラムから抽出するようなタスクフローグラフ[5] は非常に複雑であり，大量の性質の異なるタスク間に複雑なデータ依存関係がある．これに対し MegaScript では独立したプログラムを粗粒度のタスクとして組み合わせ，ユーザが明示的にタスクネットワークを記述する．このため，タスク数が多くても種類（プログラムの数）としては少数（通常 10 未満，多くても数十程度）と考えられる．また，タスク間のデータフローも粗粒度レベルであり，データの分散・収集や処理の流れの分岐といった，比較的単純なパターンの組合せがほとんどと考えられる．

⋆1 MegaScript で記述されるサイクリックなタスクネットワークは実用上，並列タスク間の定期的なデータ交換や，分割統治法を用いる場合のデータ分割と計算結果の返却など，特定のパターンに限られる．このため，将来的には今回の提案手法をベースに対応できると考えている．

3. 関 連 研 究

本章では，既存のスケジューリング手法[6),7)] を概観し，MegaScript への適用可能性について検討する．

3.1 静的スケジューリング手法

MegaScript のスケジューリングは 2.2 節 (3)，(4) で述べた不確定要素を無視すれば，タスクの計算・通信コストがすべて与えられたオンラインスケジューリング問題と見なせる．ただし，2.2 節 (1) で述べた性質により，プリエンプションはできない．

このような場合のスケジューリングについては，計算量順にソートしたタスクを，そのタスクの処理が完了する時刻が最も早いマシンに割り当てていく LPT（Longest Processing Time first）[8)] や，同様にソートしたタスクをあらかじめ決めたスケジューリング長に収まるように割り当てていき，全タスクが割当て可能な最小スケジューリング長を二分探索する MULTIFIT[9)] などが，高速で良い近似アルゴリズムとして知られている．しかしこれらの研究はタスク間が完全独立であると仮定しているので，MegaScript にそのまま適用することはできない．

タスク間に実行順制約がある場合については，レベルスケジューリングやクリティカルパス（CP）に基づくものが多数提案されている．前者は，各タスクについて自身および後続タスクの計算コストによりレベルを決定し，その順にホストへの割当てを行う．後者は DAG 上の計算コストの合計値が最大のパスを CP とし，スケジューリング長が CP に収まるように，CP 上にないタスクのホスト割当てを行う．ホストの計算・通信性能が均質な環境であれば，これらの手法は通信コストも考慮することが比較的容易であり，MCP（Modified Critical Path）[10)]，DCP（Dynamic Critical Path）[7)] などが提案されている．

しかしながら，非均質環境ではレベルやクリティカルパスが各タスクの割当て先によって変化していくので，計算の複雑さが一気に増大する．DLS（Dynamic Level Scheduling）[11)] は非均質環境を想定しているが，近似なしで計算する場合，タスク数だけ考慮しても 3 乗オーダの計算量となる．タスクやホスト数が小規模な例であれば厳密な最適化も可能であるが[6)]，規模が大きい場合は計算量の爆発を避けるため，A*や遺伝的アルゴリズムなどかなり計算時間を要する最適化を用いる手法や，大幅な近似を用いる手法などが必要になる．

非均質環境において，計算の複雑さを抑えつつ良いスケジューリング結果を得る方法の 1 つに，タスクやホストをグループ化するアプローチがある．CHP（Clustering for Heterogeneous Processors）[12)] は，依存関係にあるタスク群をグループ化し，リストスケジューリ

ングによってこれらのグループを各ホストに割り当てていく．これにより，グループ内のタスク間通信に物理通信が不要となる，リストスケジューリング対象の粒度が大きくなるためスケジューリングコストを削減できる，といった利点が得られる．しかしホストのグループ化は行っていないため，ホスト数が多い場合はやはり計算量が非常に大きくなる．

これに対し Triplet [13] は，DAG 上の分岐数と通信量でタスクをソートしてからグループ化するとともに，計算・通信性能の差が一定値以下のホスト群をグループ化する．これにより，並列性を確保しつつ通信ボトルネック部分をタスクグループ内に閉じ込め，さらに内部通信の多いタスクグループを相互通信性能の高いホストグループに割り当てることができる．しかしこの手法は比較的小規模な並列環境を想定しており，階層的なグループ化を行っていない．したがって，大規模環境へ適用するにはスケーラビリティに問題がある．

3.2　動的スケジューリング手法

タスク間が完全独立な場合には，各ワーカーホストがアイドルになるたびにマスタホストにタスクを要求する要求駆動型の動的スケジューリング手法が，古くから知られている．しかしマスタホストがボトルネックになり，タスク要求の通信回数が多いことから，大規模な広域分散環境では性能が低下しやすい．

これに対しランダムスチール（RS）法[14] では，あらかじめタスクを全ホストに分配し，実行タスクのなくなったホストが，ランダムに選択したホストにタスクの分配を要求する．この手法ではマスタホストが不要かつ通信回数が最小限で済むため，安定して高性能が得られる．広域分散環境では通信オーバヘッドのため性能が低下するが，この問題に対しては CRS（Cluster-aware Random Stealing）[15] が提案され，高い性能を得ている．

MegaScript のタスクは基本的に静的スケジューリングが可能であるが，2.2 節 (3)，(4) で述べた不確定要素を考慮すると，最終的に負荷のアンバランスが生じるのを防ぐのは難しい．そのような場合に RS や CRS のような動的スケジューリングを併用することは，性能改善に役立つと考えられる．しかし MegaScript の場合，タスク間に依存関係や通信ボトルネックが存在するため，単純にランダムな相手のタスクを半分奪うような RS では，かえって状況を悪化させる可能性もある．

4. 提 案 手 法

4.1　実行環境モデル

大規模な広域分散環境の利用例としては，SETI@home [16] のように，家庭などに存在する個々の PC を集めて利用するものもある．このような場合，各ホストの計算性能や通信性能が完全にバラバラであり，一様に非均質な環境となる．しかし，こうした環境は大量のホスト数を確保できる一方で，ほとんどのホスト間で高速な通信が期待できず，所有者の都合により頻繁かつ予告なくホストが利用できなくなるといった問題がある．このため，完全独立型の大規模問題には非常に有効であるが，MegaScript のようにタスク間の通信が頻繁に起きるような並列処理には適していない．

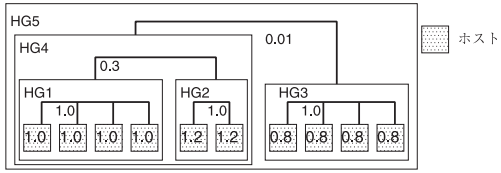


図 4　ホストのグループ化
Fig. 4　Hierarchy of host groups.

したがって，MegaScript のような並列処理に対する現実的な実行環境としては，PC クラスタのようにある程度の規模・品質の並列計算機資源が多数，広域に分散するようなものが想定される．この場合，各クラスタ内は通常，ホストの性能が均質であり，通信性能も高速かつ均質である．他のプロセスの影響による性能変動を考慮するとしても，上記の一様非均質な環境よりは，扱いやすい．

具体的なスケジューリング手法としては，クラスタ内とクラスタ間に別の手法を用いることで，低コストで効率的なスケジューリング結果を期待できる．また，複数のスケジューラを階層型に用いることで，大規模なホスト数に対するスケジューリング時間を軽減できる．これらの目的のため，本手法では，実行環境を以下の手順で階層的にモデル化する．

(1)　計算性能が同一，かつ，相互の通信速度が同一なホスト群を，それぞれ 1 つのホストグループとする．あるホストに対し，条件を満たす他のホストがない場合は，その 1 台のみで 1 つのホストグループとする．
(2)　相互の通信速度が閾値以下のホストグループをそれぞれ 1 つのホストグループとする．
(3)　全体が 1 つのホストグループになるまで，閾値を徐々に大きくしながら (2) を繰り返す．

(1) により，個々のクラスタは最内側のホストグループとなる．また，(2) により，ホストグループは内側ほど通信が高速になるように階層化される．

10 台のホストからなる 3 つのクラスタをグループ化した例を，**図 4** に示す．ホストの中の値は計算性能を，ネットワーク付近の値は通信性能を，それぞれ表す．

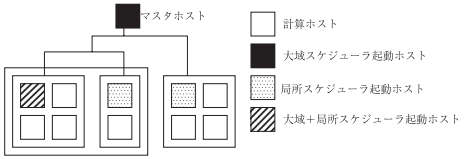


図 5　スケジューラの起動ホスト
Fig. 5　Hierarchy of schedulers.

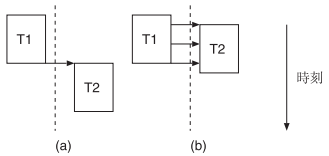


図 6　パイプライン並列性
Fig. 6　Pipeline parallelism.

4.2 基本方針

3 章で述べたように様々な静的スケジューリング手法が提案されているが，MegaScript が必要とするヘテロ環境に対する DAG スケジューリングは，高速に良い結果を得られる手法が発見されていない．とくに MegaScript の場合，タスク数やホスト数が多いため，計算量やメモリ消費量の大きいアルゴリズムは利用が難しい．また，実行環境の性能が動的に変動したり，メタモデルが不完全だったりする場合を考えると，精度の高い静的スケジューリングを行っても，そのまま実行効率の向上につながるとは限らない．そこで，以下のようなハイブリッド型のスケジューリングを行う．

(1)　4.1 節で述べたようにホストを階層モデル化し，各階層ごとにスケジューラを配置することで，段階的なスケジューリングを行う．また，最下層の均質環境とそれ以外とで，異なるスケジューリング手法を用いる．**図 5** に，図 4 のホストグループに対するスケジューラの配置を示す．

(2)　静的スケジューリングを行い，その後，動的スケジューリングによる補正を行う．

最下層のホストグループを対象とする局所スケジューラは，台数が少なく均質環境であるので，従来のように精密なスケジューリング手法が利用できる．これに対し，上位層の大域スケジューラは，直下のホストグループに対するタスクの分配のみ担当し，個々のタスクやホストに関する情報は考慮しないため，高速なスケジューリングが可能である．結果として静的スケジューリングの精度は低下するが，もともとスケジューリング情報が不完全であるため，不必要に高精度なスケジューリング手法に計算時間を費やすよりも，その時間を実行開始後に動的な補正に費やす方が，良い結果が期待できる．ただし，3.2 節で述べたように，MegaScript の動的スケジューリングでは，タスク間の依存関係や通信ボトルネックを考慮する必要がある．

4.3 タスクネットワークのモデル化

本節では，提案するスケジューリング手法に適した形で MegaScript のタスクネットワークをモデル化する手法を述べる．なお，本手法で扱うタスクネットワークは，DAG の開始ノードと終端ノードがそれぞれ単一であるものとする．複数の開始/終端ノードが存在する場合は，計算・通信コスト 0 の疑似タスクを付加することで，この条件を満たすことができる．

4.3.1 パイプライン並列性のモデル化

一般の DAG スケジューリング問題では，**図 6** (a) のように，先行タスクが実行終了時にデータを送信し，後続タスクはこれを受信してから実行を開始するモデルになっている．しかし MegaScript ではタスクの実行途中でタスク間の送受信を行うことができるため，図 6 (b) のように，通信と計算を交互に行うことで依存関係にある複数のタスクを同時に実行する，いわゆるパイプライン並列が利用できる．モデル化の際にこの性質も考慮することで，パイプライン並列性も生かしたスケジューリングが可能になる．

通信ごとにタスクを分割して扱うことで，タスクの開始・終了時に通信するモデルに変換する手法もあるが，スケジューリング対象のタスク数が増えてしまうため，もともとタスク数の多い MegaScript ではスケジューリング効率上好ましくない．そこで本手法では，タスク単位でパイプライン並列性の有無を判定し，スケジューリング時にこれを考慮する方法をとる．

タスク間にパイプライン並列性が存在するかどうかは，先行・後続タスクそれぞれについて，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によって判別できる．2.1.4 項で述べたように，一般に MegaScript のタスクはプログラムを直接解析できないため，メタプログラムを対象に入出力処理の分布を調べる．

本手法では，**図 7** 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見なし，最外側ループ内に `input` や `output` が出現するか否かで，以下の 4 パターンに分類する（**図 8**）．最外側ループが複数存在する場合は，ループを融合して 1 つの最外側ループとし，判定を行う．

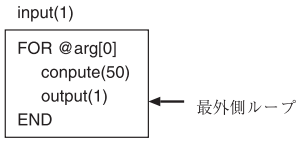


図 7　最外側ループの例
Fig. 7　Example of outmost loop.

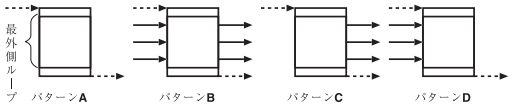


図 8　タスクの分類
Fig. 8　Task classification detecting pipeline paralellism.

- 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

このようにタスクを分類すると，先行タスクの出力と後続タスクの入力がともに最外側ループ内の内側にある場合，つまり先行タスクがパターン B または C，後続タスクがパターン B または D の組合せの場合に，パイプライン並列性が成立すると判定できる．

4.3.2　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成り立つので，依存をたどっていくことで，プログラムの実行を始めてから，注目タスクの実行を開始できるまでの最小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．

MegaScript では，メタ情報として得られるタスクの計算・入出力コスト，および前記のパイプライン並列性判定をもとに，タスクネットワーク上でコスト伝搬を行うことで，タスク間の入出力コストならびにパイプライン並列性を考慮した依存コストを求めることができる．しかし，依存コストは先行タスクを実行するホストの性能により変化するため，非均質環境ではスケジューリング結果に左右され，事前に算出しておくことができない．そこで本手法では，均質環境を対象とする局所スケジューリングのみに使うことを前提に，スケジューリング結果に左右されない依存コスト計算方法を定義する．

タスク T の計算コスト，出力コスト，依存コストをそれぞれ $C(T)$，$O(T)$，$D(T)$ とする．また，メタプログラムより得られる，このタスクの最外側ループ回数を $L(T)$ とする．

まず，T の後続タスクに対し，プログラムの実行開始から T の出力内容が最初に到着するまでのコスト $D_{first}(T)$，および，すべての出力内容が到着するまでのコスト $D_{all}(T)$ を考える．T が 4.3.1 項のパターン A，D のときは，T の計算終了後，すべての出力が行われる．よって，

$$D_{first}(T) = D_{all}(T) = D(T) + C(T) + O(T)$$

T がパターン B，C のときは，T の計算および出力が $L(T)$ 回に分けて行われる．ループ各回の計算・通信量が均等になると単純化して考えると，

$$D_{first}(T) = D(T) + (C(T) + O(T))/L(T)$$
$$D_{all}(T) = D(T) + C(T) + O(T)/L(T)$$

タスク T_p と T_s がストリームで 1 対 1 接続されている（T_p の出力を T_s の入力とする）場合，T_s の依存コストは以下のように求められる．

- T_s がパターン A，C のとき，両者の間にパイプライン並列性がないため，T_p の実行開始後，T_p の実行時間および通信時間が経過しないと，T_s は開始できない．よって，
  $$D(T_s) = D_{all}(T_p)$$
- T_s がパターン B，D のとき，T_p の実行開始後，最初の出力結果が到着した時点で，T_s は開始できる．よって，
  $$D(T_s) = D_{first}(T_p)$$

これを一般化し，タスク $T_{p1}, \ldots, T_{pm}$ の出力が 1 本のストリームでマージされてタスク $T_{s1}, \ldots, T_{sn}$ にマルチキャストされるとき，T_{si}（$i = 1, \ldots, n$）の依存コストは以下のように求められる．

- T_{si} がパターン A，C のとき，$T_{p1}, \ldots, T_{pm}$ の出力をすべて受け取らないと，T_{si} は実行を開始できない．よって，
  $$D(T_{si}) = max(D_{all}(T_{p1}), \ldots, D_{all}(T_{pm}))$$
- T_{si} がパターン B，D のとき，$T_{p1}, \ldots, T_{pm}$ のいずれかの最初の出力結果が到着した時点で，T_{si} は実行を開始できる．よって，
  $$D(T_{si}) = min(D_{first}(T_{p1}), \ldots, D_{first}(T_{pm}))$$

DAG の開始ノードにあたるタスクを T_0 とすると，$D(T_0) = 0$ であるから，ここから上

記計算式を順に適用していくことで，すべてのタスクの依存コストが求められる．

4.3.3　通信コスト

通信コストは，ストリームを通過する通信量を表すものであり，ストリームの入力端に接続されているタスクの出力コストの合計値である．タスクの出力コストが実行時引数や入力コストの関数で決まる場合は，タスクネットワークの上流から伝搬させていくことにより，通信コストを決定する．

2.1.2 項で述べたように，多対多のストリーム通信はストリームの入力側タスクの出力が非決定的にマージされ，複数の出力側タスクに同じ順序で到着する．つまり出力内容を 1 度 1 カ所に集めて到着順序を決定する必要があり，個々の入力側タスクから直接個々の出力側タスクに通信することができない．このことから通信コストは，このストリームの入出力側タスク群が複数のホストやホストグループに分割配置されたときの，通信オーバヘッドを近似する値として利用できる．

本提案手法では，この通信コストをタスクネットワーク上の結合度情報と考え，分割位置の優先度判定に用いる．将来的にはストリーム通信の実装方法や，ホスト間ネットワークのバンド幅制約による遅延も考慮した通信コスト評価モデルにすることで，精度が上がると考えられる．

4.3.4　タスクネットワークの階層モデル

大域スケジューリングでは，大量のタスクやホストを直接扱うことで計算コストが増大するのを防ぐため，タスクのグループをホストのグループに割り当てるという抽象スケジューリングを行う．このため，4.1 節でホストを階層モデル化したように，タスクも以下のアルゴリズムにより，グループ階層の形でモデル化する．

(1)　入力側と出力側の両方について，タスク/タスクグループ/タスク配列が各々1 つずつしか接続されていないストリームを探す．
　(a)　該当するストリームが存在した場合，その中で通信コストが最大のストリームとそれに接続されているタスク/タスクグループ/タスク配列を，まとめて新たなタスクグループとし，(1) へ戻る．
　(b)　該当するストリームが存在しない場合，(2) へ進む．

(2)　以下の条件を満たすストリームを探す．
　(a)　入力側に接続されているすべてのタスク/タスクグループ/タスク配列が，同一のストリームの出力側につながれている．
　(b)　出力側に接続されているすべてのタスク/タスクグループ/タスク配列についても，同様に同一のストリームの入力側につながれている．

(3)　(2) を満たすストリームが存在した場合，その中で通信コストが最大のストリームとそれに接続されているタスク/タスクグループ/タスク配列を，まとめて新たなタスクグループとし，(1) へ戻る．

(4)　該当するストリームが存在しない場合，処理を終了する．

また，以下のようにタスクグループのメタ情報を定義することで，タスクグループは疑似タスクとして扱うことができる．

計算コスト　グループ内に含まれるタスク/タスクグループ/タスク配列の計算コストの合計

入出力コスト　グループ内に含まれるタスク/タスクグループ/タスク配列の，グループ外への入出力コストの合計

このように階層グループ化することで，タスクネットワーク中で通信量の多い部分ほど下位のグループになる．大域スケジューリングの際に，階層の上位からグループを分割していくことで，通信量の少ない上位のグループが，通信の遅いホストグループにまたがって割り当てられるようになり，全体として通信遅延による速度低下が抑えられる．

また，1 対 1 接続のタスクはパイプライン並列性を持たないかぎり別々のホストグループに配置しても並列性が得られないため，優先的に下位のタスクグループになるようにしている．

図 9 に，この階層グループ化の適用例を示す．図中でタスク内の数字は計算コストを，ストリーム横の数字は通信コストを，それぞれ表す．このネットワークに対し，まず手順 (1) により，ストリーム s4，続いてストリーム配列 s3 が選ばれ，これらがグループ化される (i，ii)．続いて手順 (2) に移ると，残るストリームの中で条件 (a)，(b) を満たすのは s2 だけなので，手順 (3) によりこれらがグループ化される (iii)．その後，手順 (1) に戻るが，条件を満たすものがないので手順 (2) に進むと，(iii) のグループ化により s1，s5 も条件 (a)，(b) を満たすようになっている．このうち通信コストの大きい s1 の方がグループ化される (iv)．その後，手順 (1) に戻ると，残る s5 が条件を満たしているので，グループ化される (v)．

4.4　大域スケジューリング手法

大域スケジューリングでは，タスクとホストの階層モデルを用いて，タスクグループをホストグループに割り当てていく[17]．

大域スケジューラは，4.2 節で述べたように階層化され，各スケジューラは自身の直下にあるホストグループに対し，自身が割り当てられたタスクグループを分配する．各ホストグ

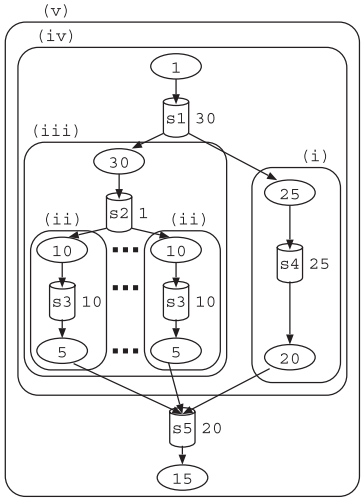


図 9　タスクネットワークの階層モデル化
Fig. 9　Hierarchy of task groups.

ループには下位のスケジューラが 1 つずつ動作しており，同様にしてさらにタスクグループを分配していく．これを繰り返すことで，タスクは段階的に細かいホスト群へと分配されていき，最終的に最下位の局所スケジューラによって，個々のホストに配置される．

本手法では，大域スケジューラはホストの計算・通信性能を個別に扱わず，自身のタスク分配作業の対象となる，直下のホストグループの計算性能（グループ内ホストの計算性能の合計）のみ考慮する．また，タスクについても階層モデルを用いることで，分配に必要な上位層のタスクグループとその計算コストだけを用いる．このため，大規模な並列処理を対象としても，非常に高速なスケジューリングが可能である．さらに，複数のスケジューラによる段階的スケジューリングにより，スケジューリングを行うホストの計算性能やメモリ量がボトルネックになることも避けられる．

ホストグループの計算性能およびタスクグループの計算コストは，それぞれ含まれるホストおよびタスクの計算性能・計算コストの合計である．したがって，計算負荷の均等化という観点からは，ホストグループの計算性能に計算コストが比例するように，タスクグループを分配すればよい．しかし，タスク間には依存や通信量の偏りが存在するので，計算コストのみ考慮したのでは，アイドル時間や通信ボトルネックが生じてしまう．

4.3.4 項で述べたアルゴリズムにより，1 つのタスクグループは 1 つのストリームとその入出力端に接続されたタスク/タスク配列/タスクグループで構成されている．そこで，このストリームの入力側・出力側のタスク群それぞれについて上記の比例配分を行うことにより，各ホストグループで前者のタスク群の実行が終了し後者の実行に移るタイミングを揃えることができる．実際には計算量を完全に比例配分できなかったり，不確定要因による誤差が発生したりするが，この方法を用いることによって，非常に小さいスケジューリングコストで依存によるアイドル時間の期待値を最小化できる．また，タスクの階層モデル化の段階で通信量の多いストリームが下層になるようにしているため，通信ボトルネックは自然にホスト内や下位のホストグループ内に閉じ込められる．

1 つの大域スケジューラは，ホストグループの集合 $H = \{H_1, \ldots, H_n\}$ に対し，タスクグループ T をスケジューリングする．ここで，$P(H_i)$，$N(H_i)$ をそれぞれ，ホストグループ H_i の計算性能（そのホストグループに含まれるホストの計算性能の合計），H_i に含まれるホスト数とする．H_i に含まれるホストの平均計算性能は，$P_{avg}(H_i) = P(H_i)/N(H_i)$ である．H に含まれるすべてのホストの計算性能の合計は，$P(H) = P(H_1) + \ldots + P(H_n)$ で表記する．また，タスクグループ T は，ストリーム S，その入力側に接続されたタスク/タスク配列/タスクグループ $I = \{I_1, \ldots, I_l\}$，および，出力側に接続されたタスク/タスク配列/タスクグループ $O = \{O_1, \ldots, O_m\}$ で構成されるとする．入力側および出力側タスクの計算コストの合計値を，$C(I) = C(I_1) + \ldots + C(I_l)$，$C(O) = C(O_1) + \ldots + C(O_m)$ とする．

これらを用いたスケジューリングアルゴリズムを以下に述べる．

(1)　タスク未分配のホストグループのうち，ホストの平均計算性能 $P_{ave}(H_i)$ が最大のものを選び（H_i とする），これを次の分配対象とする．

(2)　入力/出力側タスクのそれぞれについて，H_i に割り振るべき計算コストを以下のように求める．

$$C_{in}(H_i) = C(I) \times P(H_i)/P(H)$$
$$C_{out}(H_i) = C(O) \times P(H_i)/P(H)$$

(3)　入力側タスクを H_i に分配する．

(a)　I から計算コストが最大である要素を取り出す（I_j とする）．

(b)　$C(I_j) < C_{in}(H_i) - L$ の場合，I_j を H_i に割り当て，$C_{in}(H_i) \leftarrow C_{in}(H_i) - C(I_j)$ として，(3) (a) に戻る．ここで，L は閾値として用いる微小な正数である．

(c)　$C_{in}(H_i) - L \leq C(I_j) \leq C_{in}(H_i) + L$ の場合，I_j を H_i に割り当て，(4) に進む．

(d)　$C_{in}(H_i) + L < C(I_j)$ の場合，

(i)　I_j がタスクならば，I_j はこれ以上分割できないので，I_j を H_i に割り当て，(4) に進む．

(ii)　I_j がタスク配列ならば，その要素であるタスクの計算コスト合計値が $C_{in}(H_i)$ に最も近い位置で I_j を分割して割り当て，(4) に進む．

(iii)　I_j がタスクグループならば，I_j に対して再帰的に (2) から繰り返す．これを，(3) (d) (i) によりそれ以上分割できなくなるか，H_i に分配された総計算コストが $C_{in}(H_i) \pm L$ の範囲に収まるまで再帰的に繰り返す．その後，(4) に進む．

(4)　出力側タスクについても，同様にして H_i に分配し，(1) に戻る．

平均性能が大きいホストグループや計算コストの大きいタスクグループから順に処理を行うのは，計算量の大きいタスクを性能の高いホストを含むホストグループに配置し，各ホスト上の実行時間を均等にしやすくするためである．

図 4 に示した階層モデル化したホスト群と図 9 の階層グループ化したタスクネットワークに対し，上記アルゴリズムを適用する場合を例に説明する．最上位スケジューラには，直下のホストグループとして $HG3$，$HG4$ が与えられる．各ホストグループの性能として，以下の値が計算できる．

$P(HG3) = 0.8 \times 4 = 3.2$

$P(HG4) = 1.0 \times 4 + 1.2 \times 2 = 6.4$

$P_{avg}(HG3) = 0.8$

$P_{avg}(HG4) = 6.4/6 \approx 1.07$

$P_{avg}(HG4) > P_{avg}(HG3)$ なので，(1) より，まず HG4 を分配対象とする．最外側のタスクグループ (v) に注目すると，(2) で入力側タスクはタスクグループ (iv)，出力側タスクはコスト 15 のタスク 1 つである．仮にタスクグループ (ii) の配列の要素数を 4 とすると，

$C_{in}(HG4) = (1 + 30 + (10 + 5) \times 4 + 25 + 20) \times 6.4/9.6$

$C_{out}(HG4) = 15 \times 6.4/9.6$

となり，タスクグループ (iv) のうち 2/3 の計算コストの部分を割り当てる必要がある．そこでタスクグループ (iv) に対し同じ手順を適用すると，入力側タスクはコスト 1 のタスク 1 つであり，出力側タスクはタスクグループ (i)，(iii) である．前者についてはこれ以上分割できないので，HG4 に割り当てられる．後者については (3) (a) により，まずタスクグループ (iii) が選択される．(iii) のコストはちょうど (i) と (iii) のコストの和の 2/3 なので，(iii) が HG4 に割り当てられ，(i) が HG3 に割り当てられる．最後に，最初に注目したタスクグループ (v) の出力側タスクの割当てに戻ると，これはコスト 15 のタスク 1 つしかないため，HG4 に割り当てられる．以上の流れにより，**図 10** (a) のように分割される．もしタスクグループ (ii) の配列の要素数が 4 より大きければ，タスクグループ (iii) がさらに分割され，図 10 (b) のように (ii) の一部が HG3 に割り当てられる．

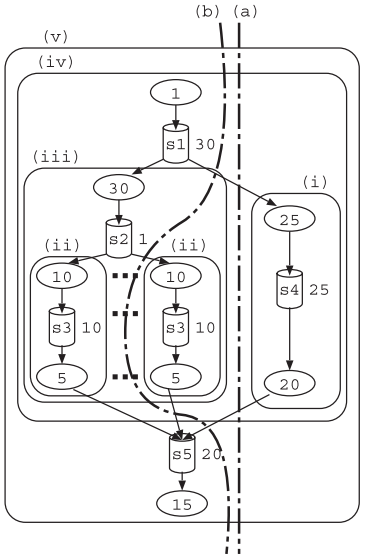


図 10　大域スケジューリングの例
Fig. 10　Example of global scheduling.

4.5　局所スケジューリング手法

局所スケジューリングでは，対象となるホストは計算・通信性能が均質であり，扱うホストやタスクの数も比較的少ない．そこで，割り当てられたタスクグループモデルを展開して元のタスク単位のネットワークモデルとし，依存コストやパイプライン並列性などの情報を考慮した，精度の高いスケジューリングを行う[18]．

本手法では，各ホストの終了時刻が均等になるように，以下のアルゴリズムでタスクを順にホストに配置していく．各ホストはアルゴリズムでの配置順にタスクを実行するものとする．

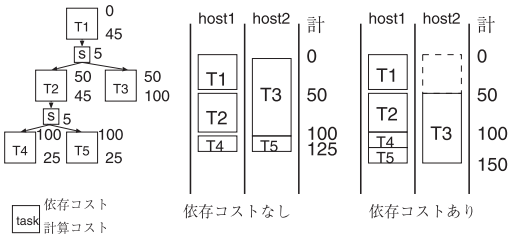


図 11　局所スケジューリングの例
Fig. 11　Example of local scheduling.

なお，対象となるタスク，ホストの集合をそれぞれ $T = \{T_1, \ldots, T_m\}$，$H = \{H_1, \ldots, H_n\}$ とする．タスク T_i の計算コスト，依存コスト，入力コストはそれぞれ $C(T_i)$，$D(T_i)$，$I(T_i)$ と表記する．また，ホスト H_j が割当て済みタスクの実行をすべて終了する時刻を $E(H_j)$ とし，全ホストについて初期値は 0 とする．

(1)　T に含まれるタスクのうち最小の依存コスト $D(T_a)$ を持つタスク T_a を次の配置対象とし，$T \leftarrow T - \{T_a\}$ とする．
(2)　$E(H_{a0}) = min(E(H_1), \ldots, E(H_n))$ であるホスト H_{a0} を次の配置候補とする．
(3)　T_a の依存タスクが割り当てられているホストも，配置候補ホスト H_{ai}（$i = 1, 2, \ldots$）とする．
(4)　$E(H_{a0}) + I(T_a) + C(T_a)$ と $min(E(H_{ai}) + C(T_a))$（$i = 1, 2, \ldots$）を比較し，小さい方を与えるホストを配置ホスト H_a に決定する．
(5)　$E(H_a) \leftarrow max(E(H_a) + I(T_a) + C(T_a),\ D(T_a) + C(T_a))$ とする．
(6)　(1) に戻る．

この手順により，パイプライン並列性や依存を考慮に入れながら，各ホストの計算量を均等にすることが可能になる．**図 11** はパイプライン並列性がないタスクネットワークを 2 台のホストに割り当てた例である．依存コストを考慮しないと，計算コストを均等に割り当てても，T3 は T1 の出力を受け取るまで実行できないため，実際は時刻 175 まで終了しない．依存コストを用いることで，時刻 150 で終了するような，より良いスケジューリングが得られる．

4.6　動的スケジューリング手法

4.2 節で述べたように，静的スケジューリングだけではメタ情報の不正確性や実行環境の性能変動に対処できない．このため，本手法では動的スケジューリングを併用する．

静的スケジューリングは，各ホストにおける担当タスク群の実行がほぼ同時に終了するように割り当てる．したがって，実行の途中で残っている計算量の不均衡が生じた場合に，動的な補正を行えばよい．

また，大域スケジューリングでは各ホストグループでの依存によるアイドル時間が少なくなるようにタスクグループを分割し，局所スケジューリングでもなるべく無駄なアイドル時間が発生しないようにタスクを配置するため，全体としてはおおむね各ホストの性能に応じた計算量のタスクが割り当てられていると期待できる．したがって，計算量の不均衡を補正する手法としては，3.2 節で述べたランダムスチール法が有効と考えられる．

ただし，MegaScript のタスクは依存や通信量の大小によるボトルネックがあるため，タスク独立型のランダムスチールのように，単純にランダムな相手からタスクの半分を奪う，といった方法では，かえって性能が悪化する可能性がある．これを改善するために，各ホストに割り当てられたタスクについて，タスクグループの情報を保持しておき，スチール時には同じグループに属するタスクをまとめて奪うことによって，通信ボトルネックがホスト間に分散するのを防ぐ．また，スチール相手を単純にランダムに選ぶのではなく，クラスタ内外を区別する手法を用いることで，CRS のように単に広域スチール時のオーバヘッドを減らすだけでなく，タスク間の通信ボトルネックが広域に分散するのを防ぐ効果が期待できる．

5.　シミュレーション評価

抽象シミュレーションにより，提案手法の評価を行った．

5.1　シミュレーション方法

本評価では，実際にタスクの実行は行わず，以下のような条件によりイベントドリブン型の抽象シミュレーションを行う．各タスクはメタ情報の計算コストだけの計算量を要し，入出力コストに従う量の通信を行うとする．

- 4.3.1 項のモデルに従い，タスクはメタプログラムの内容により，実行終了時にすべての通信を行うか，最外側ループの回数に分けて通信する．
- 各ホストは 1 度に 1 つのタスクしか実行せず，スケジューリングされた順序は追い越さないものとする．つまり，次に実行するべきタスクが依存関係により実行できない場合，そのホストは当該タスクが実行可能になるまでアイドル状態となる．
- 実行時間は最初のタスクの実行開始からすべてのタスクが終了するまでとし，スケジューリング自体に要する時間は考えない．
- タスク間通信が同一ホスト上で行われる場合の通信時間は 0 とする．

- 実環境の動的な性能変動をシミュレートするため，各ホストの計算・通信性能をランダムに低下させる．

ホストの性能が動的に変動する要因としては，OS などシステムプログラムによるものと，他のユーザのプロセス投入によるものがある．計算サーバとして使用されるホスト群においては，通常低負荷で頻繁に変動する前者に対し，比較的長時間にわたって高負荷をかけるものが多い後者の影響が大きい．また，後者は計算が主体であり，通信時間の比率は低いことが多いと考えられる．そこで今回は後者の影響を評価するために，以下のようにして変動をシミュレートした．

- N_c 個の競合タスク（プロセス）が環境内で実行されていると考える．各競合タスクの配置ホストは，それぞれ全ホストから等確率で選択し，複数タスクの同一ホストへの配置も許す．n 個の競合タスクを実行するホストの実行性能は，$1/(n+1)$ に低下する．これらの競合タスクは，1 回のシミュレーション中は実行が継続するものとし，途中で終了したり新たに生成されたりすることによる性能の変化は起こらないものとする．
- ホスト H_i，H_j 間の通信は，これらのホスト上に存在する競合タスクの通信と競合し，通信性能低下を引き起こす．競合タスクが通信を行う確率を p とし，ホスト H_i，H_j 上に存在する競合タスク数をそれぞれ n_i，n_j とすると，性能低下が起きる確率は p が小さいとき $(n_i+n_j)p$ で近似できる．そこでシミュレーション中にホスト H_i，H_j 間の通信イベントが発生するごとに，$\mu=(n_i+n_j)p$，$\sigma=\mu/3$ となるような正規分布 $N(\mu,\sigma^2)$ に従う乱数値 R を発生させ，通信時間を基本性能 $\times(1-R)$ として計算した．今回の評価では，$p=0.1$ とした．

計算性能の低下量を固定したのは，頻繁に性能が変動し平均的な性能低下量がシステム全体で一様になるケースよりも，静的スケジューリングの精度への影響が大きく，今回の評価では重要と考えたためである．今後，ランダムにユーザプロセスが発生・終了するようなシミュレーションも行うことで，実行中に変動が起きたときのスケジューラの反応速度なども評価していく予定である．

また，通信性能については，通信を中継するホストへの影響や，ホストグループ間の通信などで通信経路を共有する場合の競合などは考慮していない．今後，ネットワークトポロジや通信経路を考慮したシミュレーションを行うことで，精度を高められると考えている．

動的スケジューリング手法を併用する場合は，以下の方法で RS をシミュレートする．

(1)　割り当てられたタスクをすべて実行し終えたホスト（以下ホスト H_r）は，未実行のタスクを 2 つ以上持つ他のホスト（以下ホスト H_s）をランダムに選択する．
(2)　H_s は未実行タスクの半分を H_r に渡す．
(3)　H_r は受け取ったタスクを順次実行する．
(4)　(1) で条件を満たす H_s が存在しない場合，H_r は以後アイドル状態となる．

この方法では，RS 自体のオーバヘッドは 0 となっている．実際の RS では (1) で選ぶ H_s が渡すことのできるタスクを持っているとは限らないが，本シミュレーションではリトライ時間も 0 なので結果に影響しない．

5.2　シミュレーションの対象と環境

シミュレーション対象として，以下の手順によりタスク数 N_T 個程度のランダムなタスクネットワークを 40 通り生成した．

(1)　初期構造として，2 個のタスクを 1 本のストリームで接続したタスクネットワークを生成する．
(2)　ランダムに選択したタスク T_i に対し，以下のいずれかを等確率で適用し，ネットワークを成長させる．
　(a)　**連接**　新たにタスク T_j およびストリーム S_k を生成し，T_i を，T_i と T_j を S_k で接続した構造で置き換える．
　(b)　**コントロール並列**　新たにタスク T_j を生成し，T_i と同じ入力側・出力側ストリームを持つように接続する．
　(c)　**データ並列**　新たにタスク配列 T_a を生成し，T_i を T_a で置き換える．配列の要素数は，N_T の 1/10〜1/5 の範囲でランダムに決定する．
(3)　総タスク数が N_T 未満であれば，(2) から繰り返す．

今回の評価では N_T を 1,000 とした．また，各タスクの計算・通信コストはそれぞれ 1〜100 の間でランダムに決定し，同一タスク配列内のタスクについては同じコストを持つものとした．

2.2 節で述べたように，MegaScript のタスクネットワークはユーザが明示的に記述するため，比較的単純な構造の組合せで構成されると想定している．このため，基本的には上記手順により，実アプリケーションの構造を反映したランダムタスクネットワークが得られると考えている．ただし現在の生成手順ではタスク配列内のコストが一様でないものや，タスク配列の個々のタスクにそれぞれタスク配列が後続するような階層的なデータ並列構造を生成できない．今後，生成方法を改良して実アプリケーション全体をカバーできるようなランダムタスクネットワークによる評価を行っていきたい．

シミュレーション環境として，均質環境と不均質環境の 2 通りを用意した．前者について

表 1　均質環境の生成条件
Table 1　Attributes of homogeneous environments.

ホストの計算性能	1.8
ホスト間の通信性能	75
総ホスト数	100

表 2　非均質環境の生成条件
Table 2　Attributes of heterogeneous environments.

ホストの計算性能	0.5 – 3.0
同一グループ内のホスト間通信性能	50，100
グループの異なるホスト間通信性能	1.0
1 つのホストグループの台数	8，16，32
総ホスト数	100 – 121

は**表 1** の条件により，全ホストの計算性能およびホスト間通信性能が同一とした．後者については**表 2** の条件により，性能の異なるホストグループ（ホストグループ内は同一性能）複数からなる環境をランダムに生成したものを 40 通り用いた．

以下に述べる各評価では，各タスクネットワークごとに 1 種類の均質環境または 40 種類の非均質環境それぞれを組み合わせ，各組合せごとに 5 回のシミュレーションを行って平均実行時間を得た．

5.3　局所スケジューリング手法

均質環境・非均質環境のそれぞれに対し，性能変動の原因となる競合タスク数を変化させながら，提案手法の局所スケジューリング手法のみ用いてスケジューリングした場合，および，RS による動的スケジューリングを併用した場合のシミュレーション結果を，**表 3**，**表 4** に示す．数値は RS なし・競合タスクが 0 個の場合を 1 として正規化した，速度向上率である．

動的スケジューリングを併用しない場合，競合タスク数が増加するに従って，実行時間が増加している．これに対し，RS により補正を行うことで，競合タスク数がいずれの場合にも改善効果が得られた．均質環境・競合タスクなしの場合にも RS により 34%程度の速度向上が得られており，これはもともとの静的スケジューリングの解が最適ではなく，動的に改善されたと考えられる．非均質環境・競合タスクなしの場合には，さらに高い 47%程度の速度向上が得られたが，これは非均質性により静的スケジューリングの精度がさらに落ちたのに対し，RS による改善効果が出たものと考えられる．

表 3　均質環境での速度向上率
Table 3　Speedup on homogeneous environments.

競合タスク数	0	10	100
RS なし	1.0000	0.8330	0.4621
RS あり	1.3440	1.0090	0.5326

表 4　非均質環境での速度向上率
Table 4　Speedup on heterogeneous environments.

競合タスク数	0	10	100
RS なし	1.0000	0.8529	0.4774
RS あり	1.4740	1.1648	0.6208

表 5　階層型スケジューリング手法による速度向上率
Table 5　Speedup using hierarchical scheduling.

競合タスク数	0	10	100
単一スケジューラ	1.0000	0.8529	0.4774
階層型スケジューラ	9.8294	6.2993	3.3062

この結果は，通信ボトルネックや依存を考慮しない単純な RS でも，ある程度まで静的スケジューリングに対する補正効果が得られることを示している．競合タスクがある場合の実行時間の下界値が不明であるため今回の結果からは判断できないが，競合タスクによる性能変動についても，同様に RS による動的な改善効果が得られているものと推測できる．今後，局所スケジューリング手法の精度を上げ，非均質性や性能変動に対する RS の補正効果を詳しく分析していく必要がある．

5.4　大域スケジューリング手法

大域スケジューラと局所スケジューラを組み合わせた階層型スケジューリング手法を評価するため，単一の局所スケジューラのみを用いた場合との比較を行った．結果を**表 5** に示す．ホストは非均質環境を用い，数値は単一スケジューラ・競合タスク数 0 の場合を 1 として正規化した，速度向上率である．RS は併用していない．

階層型スケジューリングにより，単一の局所スケジューリングと比較して 6.9〜9.8 倍の速度向上が得られた．局所スケジューラは個々のタスク・ホスト単位で詳細なスケジューリングを行っているが，ホストの計算・通信性能が一様であることを想定しているため，非均質環境では非効率的なスケジューリングを行ってしまう．これに対し大域スケジューラを併用することで，非常に単純なスケジューリング方式であるにもかかわらず大幅に効率的な結

表 6　動的スケジューリング併用による速度向上率
Table 6　Speedup using hierarchical scheduling and RS.

競合タスク数	0	10	100
RS なし	1.0000	0.6631	0.3560
RS あり	0.9886	0.7168	0.3861

果が得られている．

また，RS を併用したシミュレーションも行い，RS なしの場合と比較した．結果を**表 6** に示す．数値は RS なし・競合タスク数 0 の場合を 1 として正規化した，速度向上率である．

前節にあげた局所スケジューラのみの場合と異なり，RS による改善効果はわずかしか見られない．今回のシミュレーションで用いた RS は，タスクを奪うホストの選択はホストグループ内外を区別せずに行っているため，通信量の大きいタスク群の一部が他のクラスタに移動してしまい，通信時間の増大による速度低下を生じる可能性がある．競合タスク数 0 の場合に RS によりわずかに悪化しているのは，もともとの静的スケジューリングの精度が高く動的補正による改善があまり期待されないのに対し，この通信時間増大が生じてしまったものと考えられる．今後，適切なホストから適切なタスクを奪うような RS の改良手法を構築し，提案手法の性能改善を行っていきたい．

5.5　スケジューリング精度の評価

提案した階層型手法で得られるスケジューリングの精度を確認するため，最適スケジューリングとの比較評価を行った．評価には，**図 12** に示す比較的簡単なタスクネットワークを用いた．各図のタスク内部の数字は計算コスト，ストリーム内部の数字は通信コストをそれぞれ表している．また，実行環境として**表 7** に示す小規模な非均質ホスト群を用いた．各ホストグループ内およびグループ間の通信速度は，それぞれ 100，10 とした．

タスクネットワーク TN_A，TN_B のそれぞれをこの小規模環境で実行するとき，手動で求めた最適スケジューリングと提案手法で求めたスケジューリング結果のそれぞれについてシミュレーションを行い，実行時間比を求めた．結果を**表 8** に示す．最適解に対し，提案手法による実行時間の増加は単純な TN_A で 2.5%，少し複雑な TN_B で約 11%にすぎず，高い精度の解を得ていることが分かる．

5.6　大規模環境での評価

より大規模なタスクネットワークや実行環境に提案手法を用いた場合の効果を確認するために，前節のタスクネットワークや実行環境に相似な大規模ネットワークと大規模環境を用いて，評価を行った．

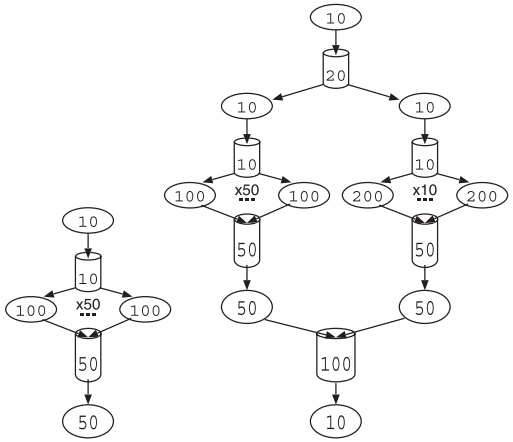


図 12　精度評価用のタスクネットワーク
Fig. 12　Task network for accuracy evaluation.

表 7　手動スケジューリング用の小規模環境
Table 7　Small environment for hand-scheduling.

ホストグループ	計算性能	ホスト数
A	4	5
B	2	10
C	1	20

前節のタスクネットワーク TN_A，TN_B に対し，タスク配列部分のタスク数を 10 倍，100 倍にしたものを用意し，各々に対して表 7 のホストグループ A，B，C の各ホスト数をそれぞれ 10 倍，100 倍にしたものを実行環境として，提案手法によるスケジューリングおよび実行シミュレーションを行った．競合タスク数は 0 とし，RS は併用していない．

シミュレーションで得られた実行時間より，以下の手順により計算機使用率を求めた．

(1)　計算量 C：全タスクの計算コストの和
(2)　実行環境の計算性能 P：全ホストの計算性能の和
(3)　実行時間 T：シミュレーションで全タスクが終了するまでの時間
(4)　計算機使用率：$C/(P \times T)$

表 8　最適スケジューリングに対する実行時間比
Table 8　Time ratio to optimal scheduling.

タスクネットワーク	実行時間比
TN_A	1.025
TN_B	1.109

表 9　計算機使用率の比較
Table 9　Scalability of CPU usage.

タスク数/ホスト数	約 50/35	約 500/350	約 5,000/3,500
TN_A	0.6970	0.6895	0.6888
TN_B	0.7339	0.7256	0.7247

表 10　スケジューリング時間
Table 10　Scalability of static scheduling time.

タスク数/ホスト数	1,000/100	10,000/1,000
単一スケジューラ（秒）	4.83	2,812.90
階層型スケジューラ（秒）	1.91	63.94

結果を**表 9** に示す．小規模な場合に 70%前後の計算機使用率が得られ，規模が大きくなった場合の減少度もわずかであった．大規模なタスクネットワーク・実行環境に対しては手動で最適スケジューリングを行うことが難しいため，提案手法の精度を直接評価することができないが，計算機使用率がほとんど低下していないことから，大規模な場合でも小規模のときと同等の精度が得られたと考えられる．1 万タスクを超える場合やより複雑なタスクネットワーク・実行環境を用いる場合のスケーラビリティについては，今後評価を行っていく必要がある．

5.7　スケジューリング時間

実装したスケジューラが静的スケジューリングに要する実時間を，PC（Xeon 2.13 GHz）上で測定した．階層型スケジューラについても，すべてのスケジューラを 1 台の PC 上で動作させて測定した．結果を**表 10** に示す．1,000 タスク/100 ホストの評価には，5.2 節で説明したランダムなタスクネットワークと不均質環境を用いた．10,000 タスク/1,000 ホストの評価は，同じタスクネットワークのタスク配列部分をそれぞれ 10 倍にしたものと，表 2 の条件のうち 1 ホストグループの台数を 32，64，128，全ホスト数を 1,000–1,127 としてランダムに生成した非均質環境を用いた．

単一スケジューラでは，タスク数・ホスト数ともに 10 倍ずつになったとき約 580 倍の時間を要するのに対し，階層型スケジューラでは 30 倍程度の増加であった．実環境で階層型スケジューラを動作させる場合はスケジューラ間の通信オーバヘッドが生じるが，複数のスケジューラが並列動作することによる時間短縮も期待できる．したがって単一スケジューラに比べ，大規模環境でも非常に高速なスケジューリングが可能であるといえる．

6. おわりに

本論文では，複数の方式を併用するハイブリッド型のスケジューリング手法を提案した．複数のスケジューラを階層的に配置してスケジューリング処理を分散させ，上位層と下位層で異なる方式を用いることによって，大規模なタスク数・ホスト数を扱う場合にも，ある程度の精度を持つ静的スケジューリングを効率良く行える．また，動的スケジューリングを併用し，実行状況にあわせた補正を行うことで，スケジューリング情報や実行環境の変動といった不確定要因に柔軟に対応でき，安定した実行性能を達成できる．

シミュレーションによる性能評価の結果，従来型の単一スケジューラを用いた場合に対し，階層型スケジューリングにより 7〜10 倍程度の速度向上を達成した．また，大規模環境に対するスケジューリングについては，スケジューリング結果に高いスケーラビリティが得られ，10,000 タスク・1,000 ホスト規模の場合，従来型の約 1/44 の時間でスケジューリングを行うことができた．

動的スケジューリングの併用については，精度が不十分であったと見られる局所スケジューリングにおいて今回の単純な RS 併用型でも性能向上が得られたが，階層型スケジューリングではほとんど効果が見られなかった．今後，局所スケジューリングの精度を高めるとともに，本手法に適した RS の改良手法を考えていく必要がある．

さらに，ネットワークの挙動をより現実的なモデルにするなど，より精度を高めたシミュレーション評価を行うとともに，広域分散環境での実評価を行っていくことも今後の課題である．

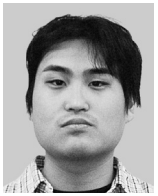
謝辞　本研究の一部は文部科学省科学研究費補助金（特定領域研究，研究課題番号 19024041，「高性能計算の高精度モデル化技術」）による．

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) 湯山紘史，津邑公暁，中島　浩：タスク並列言語 MegaScript 向け高精度実行モデルの構築，情報処理学会論文誌：コンピューティングシステム，Vol.46, No.SIG12 (ACS11), pp.181–193 (2005).
3) 阪口裕輔，大野和彦，佐々木敬泰，近藤利夫，中島　浩：タスク並列スクリプト言語処理系におけるユーザレベル機能拡張機構，情報処理学会論文誌：コンピューティングシステム，Vol.47, No.SIG12 (ACS15), pp.296–307 (2006).
4) 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).
5) 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, pp.910–920 (2001).
6) 須田礼仁：ヘテロ並列計算環境のためのタスクスケジューリング手法のサーベイ，情報処理学会論文誌：コンピューティングシステム，Vol.47, No.SIG18 (ACS16), pp.92–114 (2006).
7) Kwok, Y. and Ahmad, I.: Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors, *IEEE Trans. Parallel and Distributed Systems*, Vol.7, No.5, pp.506–521 (1996).
8) Gonzalez, T., Ibarra, O.H. and Sahni, S.: Bounds for LPT Schedules on Uniform Processors, *SIAM Journal on Computing*, Vol.6, No.1, pp.155–166 (1977).
9) Friesen, D.K. and Langston, M.A.: Bounds for MULTIFIT Scheduling on Uniform Processors, *SIAM Journal on Computing*, Vol.12, No.1, pp.60–70 (1983).
10) Wu, M. and Gajski, D.D.: Hypertool: A Programming Aid for Message-Passing Systems, *IEEE Trans. Parallel and Distributed Systems*, Vol.1, No.3, pp.330–343 (1990).
11) Sih, G.C. and Lee, E.A.: A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures, *IEEE Trans. Parallel and Distributed Systems*, Vol.4, No.2, pp.175–187 (1993).
12) Boeres, C., Filho, J.V. and Rebello, V.E.F.: A Cluster-based Strategy for Scheduling Task on Heterogeneous Processors, *SBAC-PAD '04: Proc. 16th Symposium on Computer Architecture and High Performance Computing*, pp.214–221 (2004).
13) Cirou, B.: Triplet: A Clustering Scheduling Algorithm for Heterogeneous Systems, *ICPPW '01: Proc. 2001 International Conference on Parallel Processing Workshops*, pp.237–244 (2001).
14) Blumofe, R.D. and Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing, *35th Annual Symposium on Foundations of Computer Science* (*FOCS'94*), pp.356–368 (1994).
15) van Nieuwpoort, R.V., Kielmann, T. and Bal, H.E.: Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications, *Proc. 8th ACM SIGPLAN symposium on Principles and Practices of Parallel Programming* (*PPoPP'01*), pp.34–43 (2001).
16) SETI@home. http://setiathome.berkeley.edu/
17) 片野　聡，森英一郎，大野和彦，佐々木敬泰，近藤利夫，中島　浩：不均質環境におけるタスクネットワークの静的スケジューリング手法，情報処理学会研究報告 2007-HPC-111, pp.37–42 (2007).
18) 片野　聡，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解析情報を用いたスケジューリング手法，情報処理学会研究報告 2006-HPC-107, pp.61–66 (2006).

(平成 20 年 7 月 9 日受付)
(平成 20 年 10 月 9 日採録)



松本　真樹（学生会員）

2007 年三重大学工学部情報工学科卒業．現在，同大学大学院工学研究科情報工学専攻博士前期課程在学中．広域分散環境における大規模並列処理に興味を持つ．

片野　聡

2006 年三重大学工学部情報工学科卒業．2008 年同大学大学院工学研究科情報工学専攻博士前期課程修了．同年（株）デンソー入社．在学中は大規模並列処理の静的スケジューリングに関する研究に従事．

佐々木敬泰（正会員）

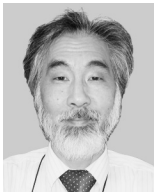
1998 年広島市立大学工学部情報工学科卒業．2000 年同大学大学院情報科学研究科修士課程修了．2003 年同大学院博士後期課程修了．同年三重大学工学部情報工学科助手，現在に至る．博士（情報工学）．マルチプロセッサ，細粒度並列処理アーキテクチャ，低消費電力プロセッサ，動画像高圧縮技術に関する研究に従事．電子情報通信学会会員．

大野　和彦（正会員）

1998 年京都大学大学院工学研究科情報工学専攻博士後期課程修了．同年豊橋技術科学大学助手．2003 年三重大学講師．言語の設計・実装・最適化等，並列プログラミング環境に関する研究に従事．博士（工学）．

近藤　利夫（正会員）

1976 年名古屋大学工学部電気工学科卒業．1978 年同大学大学院修士課程修了．同年日本電信電話公社入社．2000 年三重大学工学部情報工学科教授．SIMD プロセッサに関するアーキテクチャ，プロセッサ配列型の専用 LSI 構成，文字認識処理への応用等の研究・開発，MPEG-2 映像符号化 LSI の開発を経て，現在，高精細映像符号化システム等への並列処理の応用に関する研究に従事．工学博士．電子情報通信学会，IEEE 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．2006 年京都大学教授．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，同理事等を歴任．IEEE-CS，ACM，ALP，TUG 各会員．