

修士論文

題目

MegaScript における大規模  
ワークフローの実行効率化手法の  
提案と評価

指導教員

大野 和彦 講師

平成 23 年度

三重大学大学院 工学研究科 情報工学専攻  
計算機アーキテクチャ研究室

三田 明宏 (410M525)

三重大学大学院 工学研究科

## 内容梗概

我々は、ワークフロー型の並列プログラミング言語 MegaScript を開発している。MegaScript はオブジェクト指向言語であり、個々のタスクや通信路であるストリームなどをオブジェクトで表す。このため柔軟な記述が可能である一方、タスク数に応じたオブジェクトが生成されるため、実行可能なワークフローの規模がマスターホストのメモリ量に制約される問題がある。そこで、タスクの配列の縮約表現を用いて等価なワークフローで表すことで、大規模ワークフローの情報を少ないメモリ量で保持できる手法を提案している。しかし、上位層と下位層の並列性を持つ階層型のワークフローでは、タスクの配列の縮約手法では十分な効果が期待できない。この問題を解決するために、階層構造の縮約手法を提案している。しかし、MegaScript 処理系のための設計や実装が行われていない。そこで、縮約手法の詳細な検討を行い設計、実装及び評価を行った。

階層構造の縮約手法の実現のために、新たに API クラス TaskNet を用意した。TaskNet はワークフローの一部であるサブワークフローを表すものである。サブワークフローの繰り返しはこのクラスの配列で表現できる。また、この配列はタスク配列の縮約と同様の手法で縮約できる。この手法を用いることで、階層型のワークフローでも少ないメモリ使用量で表現できる。

マスターホスト上におけるメモリ量の制約は縮約手法を用いることで解決できるが、スレーブホスト上で縮約表現を効果的に利用できない問題がある。従来の MegaScript 処理系は一度にすべてのタスクプロセスを生成しタスク間通信を行っている。そのため、タスク間の通信を行う前に縮約されているオブジェクトをすべて展開しなければならない。大量のオブジェクトが生成されることにより、MegaScript 処理系のメモリ使用量が多くなる。また、同時に大量のタスクプロセスが実行されると、各プロセスに十分なメモリ量を割り当てることが難しくなる。縮約されたタスク群から必要なタスクのみを部分展開してプロセスを生成し、タスク間通信を行う手法を提案し、実装した。

性能評価の結果、マスターホストで消費されるメモリ量は従来では 100 万タスクで 100-200MB 程度必要であったが、完全縮約表現が可能であれば 2-3KB 程度に削減することができた。また、提案する通信機構は同時に実行されるプロセス数の違いのため、MegaScript の実行時間が増加し

た．しかし，従来手法では1万個のタスクを生成すると正常に実行できなかったが，提案手法では1万個もしくはそれ以上のタスクを生成しても正常に実行できた．提案手法はタスクプロセス生成のときに必要になるオブジェクトを縮約状態から部分展開して生成するため，少ないメモリ量でオブジェクトが管理できる．提案手法によって，スレーブホストのメモリ使用量を削減しつつ，大規模ワークフローの実行が可能になった．

# Abstract

We are developing a parallel script programming language *MegaScript* for large-scale workflows. MegaScript is an object-oriented programming language and each task and communication channel called stream is represented as an object. Thus, the executions of large-scale workflows are limited by the memory size of the master host. Therefore, we have proposed a scheme largely reducing the number of objects using array contraction. Although contraction of TaskArray largely reduces objects created for a task array, the contraction of each task array may not be so effective in workflows with hierarchical parallelism. Therefore, we have proposed a scheme to contract such workflows. However, this method is not implemented in MegaScript. This paper presents the design and evaluation of contraction method in MegaScript.

We provide a new API class **TaskNet** for brief description of subworkflows. Users can also create a group of subworkflows using array of **TaskNet**. Applying contraction for the array of **TaskNet**, the hierarchical workflows can be contracted. Using this method, workflows are created efficiently and required a small amount of memory.

Using contraction method, memory size problem has been solved on the master host. Although, contraction methods can not be used effectively on the slave hosts. The current implementation of MegaScript runtime creates all task processes at the beginning of a workflow execution. Thus all contracted task arrays must be expanded. Therefore, we propose a new design of MegaScript runtime which enables workflow execution with progressive creation of task processes. This design can minimize the expansion of contracted arrays and efficient workflow execution is possible.

As the result of evaluation using contraction method, the required memory size was reduced from 100–200MB to 2–3MB on the master host. Although proposed method increased execution time, large-workflow which can not be executed in existing MegaScript runtime can be worked well.

# 目次

|       |                         |    |
|-------|-------------------------|----|
| 1     | はじめに                    | 1  |
| 2     | 背景                      | 3  |
| 2.1   | タスク並列スクリプト言語 MegaScript | 3  |
| 2.1.1 | タスク                     | 3  |
| 2.1.2 | ストリーム                   | 4  |
| 2.2   | MegaScript の API        | 4  |
| 2.2.1 | 処理の流れ                   | 5  |
| 2.3   | タスク配列の縮約表現              | 6  |
| 2.4   | 階層構造の縮約                 | 8  |
| 3     | 階層構造の縮約手法の設計            | 9  |
| 3.1   | TaskNet クラス             | 9  |
| 3.2   | 階層構造の縮約                 | 10 |
| 4     | 縮約手法の実装                 | 13 |
| 4.1   | 実装上の問題                  | 13 |
| 4.2   | データ通信機構                 | 13 |
| 4.2.1 | 一对多のタスク間通信              | 14 |
| 4.2.2 | 多対一のタスク間通信              | 16 |
| 4.2.3 | 多対多のタスク間通信              | 17 |
| 4.2.4 | タスクの縮約                  | 18 |
| 4.2.5 | ストリームの縮約                | 19 |
| 5     | 評価                      | 22 |
| 5.1   | 実ワークフローのメモリ使用量          | 22 |
| 5.2   | 実行時間の評価                 | 24 |
| 6     | 関連研究                    | 29 |
| 7     | おわりに                    | 32 |
|       | 謝辞                      | 33 |
|       | 参考文献                    | 34 |

## 目 次

|      |                                       |    |
|------|---------------------------------------|----|
| 2.1  | ストリームの振る舞い . . . . .                  | 3  |
| 2.2  | ストリームの実装概要 . . . . .                  | 4  |
| 2.3  | マスタープロセス/スレーブプロセスの処理の流れ . . . . .     | 6  |
| 2.4  | タスク配列の縮約表現 . . . . .                  | 7  |
| 2.5  | CG レンダリング . . . . .                   | 8  |
| 2.6  | にタスク配列の縮約を適用した結果 . . . . .            | 9  |
| 2.7  | 階層型ワークフローの縮約 . . . . .                | 9  |
| 3.8  | TaskNet を用いた MegaScript コード . . . . . | 11 |
| 4.9  | 一対多のワークフロー . . . . .                  | 15 |
| 4.10 | 一対多の従来実装手法 . . . . .                  | 15 |
| 4.11 | 一対多のタスク間通信の例 . . . . .                | 16 |
| 4.12 | 多対一のワークフロー . . . . .                  | 17 |
| 4.13 | 多対一のタスク間通信の例 . . . . .                | 17 |
| 4.14 | 多対多のワークフロー . . . . .                  | 18 |
| 4.15 | 多対多のタスク間通信の例 . . . . .                | 18 |
| 4.16 | 縮約タスクの分割 . . . . .                    | 20 |
| 4.17 | 縮約状態の一対多のタスク間通信の例 . . . . .           | 21 |
| 4.18 | 一対一のワークフロー例 . . . . .                 | 21 |
| 4.19 | 縮約した一対一のワークフロー例 . . . . .             | 22 |
| 4.20 | (縮約あり) 一対一のタスク間通信の例 . . . . .         | 22 |
| 5.21 | 評価に用いた実ワークフロー . . . . .               | 24 |
| 5.22 | 一対 $N$ のワークフロー構造 . . . . .            | 26 |
| 5.23 | 10MByte の配列 . . . . .                 | 28 |
| 5.24 | 20MByte の配列 . . . . .                 | 28 |
| 5.25 | 30MByte の配列 . . . . .                 | 28 |

## 表 目 次

|     |                                 |    |
|-----|---------------------------------|----|
| 5.1 | 評価アプリケーション . . . . .            | 23 |
| 5.2 | タスク配列の必要メモリ量 (bytes) . . . . .  | 23 |
| 5.3 | 実アプリケーションのワークフローの縮約結果 . . . . . | 25 |
| 5.4 | 実行時間 (秒) . . . . .              | 27 |

## 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、単一プロセッサでの性能向上は頭打ちになりつつあり、並列処理への期待が高まっている。大規模な並列アプリケーションを作成する手法の一つである、複数の独立したプログラムをタスクとして組み合わせたワークフローが使われるようになっている [1] ~ [3]。ワークフローは、各タスクの独立性が高く高性能な既存ソフトウェアを再利用しやすいこと、タスク間のデータフローを非循環有効グラフ (DAG) の形で直感的に記述できること、などの利点を持つ。また、ワークフローは一般にタスクやタスク間通信の粒度が大きく、大規模な計算資源を安価に供給できる広域分散環境との相性がよい。このため、プロセッサ・メモリ・ディスクなど大量の計算資源を必要とする天文学・生物学・物理学などの科学技術の分野において、大規模な問題を解く実用的な手段として盛んに利用されている [4][5]。

そこで我々は、ワークフローを記述するだけで容易に大規模並列処理が行えることを目的としたタスク並列スクリプト言語 MegaScript を提案し、開発を進めている [6] ~ [8]。MegaScript では、個々のタスク (実行単位) やタスク間通信路であるストリームをオブジェクトで表し、これらのオブジェクトを生成・操作することでタスクのパラメータ設定やワークフロー構造の定義を行う。このため、任意の形状のワークフローを直感的に記述できる反面、ワークフローのタスク数と同規模の個数のオブジェクトが生成される。その結果、実行可能なワークフローの規模がスクリプトを実行するマスターホストのメモリ量に制約される。また、タスクスケジューリングや各実行ホスト (スレーブホスト) にタスク情報を送信する際のコストも、ワークフローの規模の増加に伴い無視できなくなる。

大規模ワークフローはパラメータスイープのように同じようなタスクが多数生成されることが多い。MegaScript ではこのような同種のタスクを多数生成する場合を想定したタスク配列がある。そこで、このタスク配列を少ないメモリ使用量で表現できるタスク配列の縮約表現を提案している [9]。タスク配列の縮約表現を用いて等価なワークフローで表すことで、大規模ワークフローの情報を少ないメモリ量で保持できる。しかし、上位層と下位層の並列性を持つ階層型のワークフローでは、タスク配列の縮約手法だけを用いても十分な効果が期待できない。この問題を解決するために階層構造の縮約手法を提案している [10][11] が、この手法の設計・実装が行われていない。そこで、縮約手法の詳細な検討を行い設計、実装及び評価を行う必要がある。

MegaScript 処理系に縮約表現を導入するにあたり，タスク間の通信処理が問題となる．現在，スレーブホストはマスターホストから転送されたタスク情報を元に一度にすべてのタスクプロセスを生成しており，それを前提としてタスク間のデータ通信機構が設計されている．縮約手法の効果を高めるには，縮約されたタスク群から必要とされるタスクを適宜展開しプロセス生成を行うことが望ましい．しかし，その場合現行のデータ通信機構は正しく動作しない．そこで，タスクプロセスを一度に生成しなくてもタスク間通信ができる手法を提案し [12]，縮約されたタスクに対しても正しく動作するデータ通信機構を設計・実装した．

以下，2 章で MegaScript の概要を述べる．続いて，3 章で階層構造の縮約手法の設計の詳細を述べ，そして，4 章で現在の実装における縮約表現の問題点とその解決手法を述べる．5 章で実ワークフローのメモリ使用量と実装した通信機構の評価を示す．6 章で関連研究について述べ，最後に 7 章でまとめる．

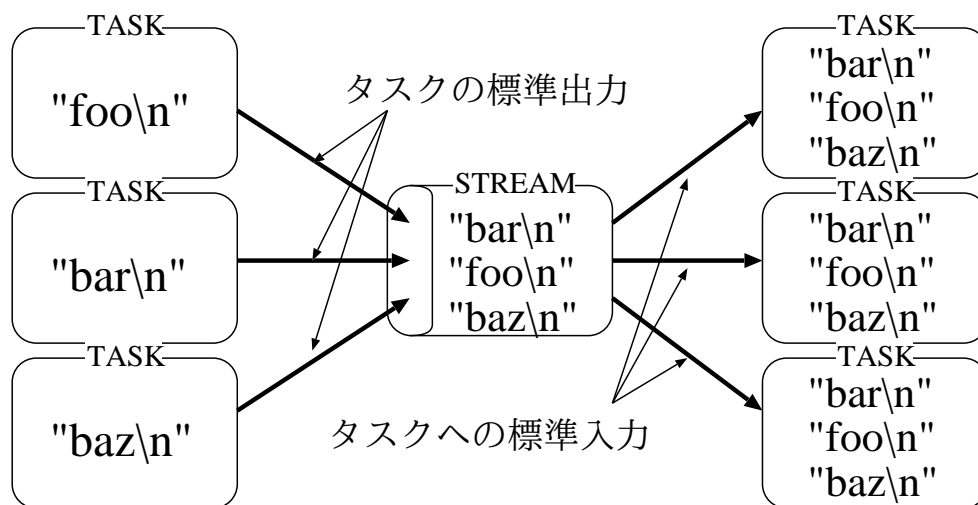


図 2.1: ストリームの振る舞い

## 2 背景

### 2.1 タスク並列スクリプト言語 MegaScript

MegaScript はワークフローモデルに基づく粗粒度並列言語であり，独立したプログラムをタスクと見なして並列実行する．また，タスク間で通信を行うために，ストリームと呼ばれる仮想通信路を提供している．ストリームの入出力端には複数のタスクを接続することができ，入力端に接続したタスク群の出力が非決定的にマージされ，出力端に接続したタスク群にマルチキャストされる (図 2.1)．現状の実装では標準入出力のテキストストリームのみ扱い，行単位をアトミックなメッセージとしている．

#### 2.1.1 タスク

MegaScript では並列の実行単位をタスクとして定義している．タスクは独立した外部プログラムであり，ユーザは任意の言語で生成したプログラムをタスクとして使用できる．このため，既存のプログラムをタスクとして流用したり，処理内容に応じたプログラム言語を使える．また，MegaScript はタスク内部の処理に一切関与せず，タスク間の情報のやりとりは標準入力を利用している．

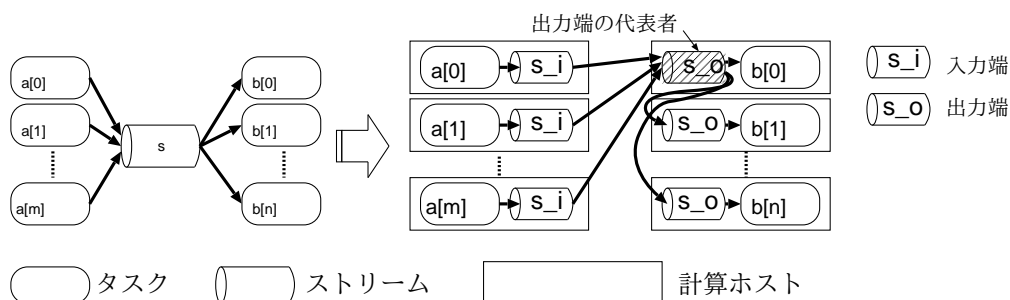


図 2.2: ストリームの実装概要

### 2.1.2 ストリーム

ストリームの実装概要を図 2.2 に示す．MegaScript 処理系では，プログラム中に定義されたストリームに対し，入力端と出力端をそれぞれ生成する．そして，入力側に接続されているタスクを調べ，それらを実行するすべてのホスト毎に対して入力端を 1 個ずつ配置する．また，出力端も同様に，該当するすべてのホストに 1 個ずつ配置する．

出力端のうち 1 つは代表出力端として処理系内で扱われる．ストリームを流れるデータは一旦代表出力端に集められデータのマージが行われる．代表出力端にはそれ以外の出力端の配置先ホストがあらかじめ MegaScript 処理系より与えられており，その情報を元にマージしたデータの転送を行う．以上の実装により，MegaScript 処理系内でストリームを用いたデータのマージ・マルチキャスト処理が実現されている．

## 2.2 MegaScript の API

MegaScript ではタスクやストリームなどを表す API クラスが用意されている．これらのオブジェクトインスタンスによって実際のタスクやストリームを表現する．現在の MegaScript では以下の 4 クラスが提供される．

- Task

MegaScript におけるタスクを表現するためのクラスである．外部プログラムのファイル名や実行時引数などタスクの実行に必要な情報を持つ．1 オブジェクトが 1 つのタスクに対応する．

- TaskArray  
タスクのプロジェクトを要素として持つ配列のクラスである．同種のタスクを多数生成する場合に用いる．
- Stream  
MegaScript におけるストリームを表現するためのクラスである．1 オブジェクトが1本のストリームに対応する．タスク間の接続情報などを持つ．
- StreamArray  
ストリームのオブジェクトを要素として持つ配列のクラスである．一度に多数のストリームを生成する場合に用いる．

### 2.2.1 処理の流れ

図 2.3 に MegaScript の処理の流れを示す．最初に MegaScript プログラムを実行したホスト上で MegaScript 処理系のマスタープロセスが起動する．一方，利用可能な他のホストはスレーブプロセスを起動することでタスク実行ホスト（スレーブホスト）として扱うことができる．各プロセスには Ruby インタプリタ [13] が組み込まれており，マスタープロセスを起動したホスト（マスターホスト）上で MegaScript プログラムを実行し，メモリ上にオブジェクトで表現されたワークフローの全体像を構築する．

MegaScript のスケジューラはワークフローの構造を解析し，各タスクの実行順や実行ホストを決定する．MegaScript では，ユーザの記述したメタプログラムや実行時プロファイリングを元に，タスク毎の計算量やデータ通信量を表すコスト関数を生成しておくことができる．スケジューラはスクリプト実行時に，各タスクに与えられる実行時引数をこのコスト関数に当てはめることで，各コスト値を計算し通信・実行時間を推定する．

スケジューリング結果に従い，各タスクオブジェクトをシリアルライズしてスレーブホストに転送する．スレーブホスト上でこのデータを受信しデシリアルライズを行う．そして，タスクオブジェクトの情報を元にタスクプロセスを生成し実行する．また，各タスクプロセスはストリームを通して、データ通信を行う．

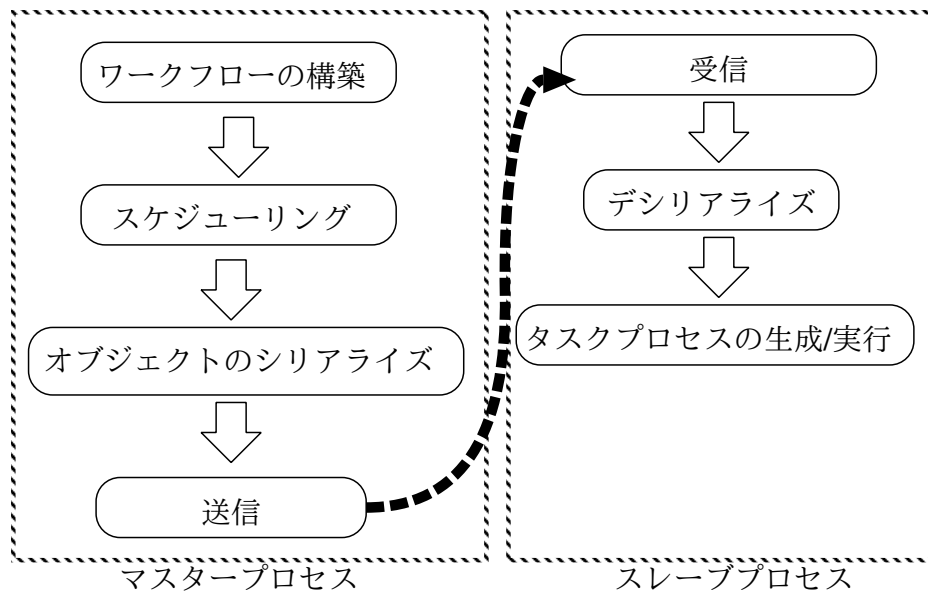


図 2.3: マスタープロセス/スレーブプロセスの処理の流れ

## 2.3 タスク配列の縮約表現

近年，ワークフロー型の並列処理の大規模化が進み扱うタスク数が増えてきている．従って MegaScript においても，タスク数が増えても効率的に実行できるような処理系が必要になる．MegaScript は個々のタスクやストリームなどをオブジェクトで表す．このため，柔軟な記述が可能である一方，タスク数に応じてオブジェクトがメモリ上に生成されるため，実行可能なワークフローの規模がマスターホストのメモリ量に制約されてしまう．そこで，メモリ上のワークフロー構造を縮約して表現することにより，生成されるオブジェクト数を減らしメモリ使用量を削減する．

MegaScript では同種のタスクを多数生成する場合を想定したタスク配列クラスを用意している．通常，大規模なタスク配列を用いるのは，パラメータスイープのように同じプログラムを異なる条件で多数実行する場合であり，そのようなタスク群の実行プログラム名は同一になる．実行時引数についても，同一のものが多く，タスク毎に異なるものは一部に限られる．さらに，その引数は一定の規則がある．従ってその異なる引数に範囲を表す Range オブジェクトや引数付きブロックを与える Proc オブジェクトを用いるならば，これらの実行時引数を表すオブジェクト

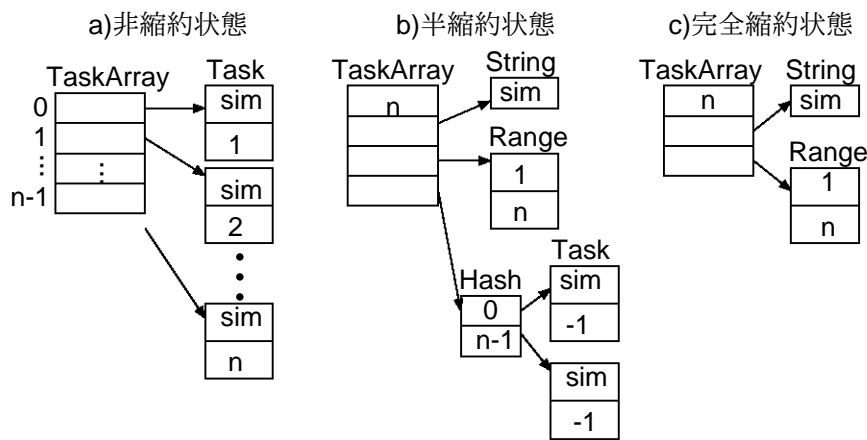


図 2.4: タスク配列の縮約表現

のメモリ使用量はタスク配列の大きさによらず小さくなる．ストリームについてもタスク配列は全体でまとめて接続を行うことが多く，この場合，配列内のタスクの入出力ストリームは同一になる．つまり，ユーザが記述した実行時引数を展開せずにそのまま保持すれば，多くの場合は小規模なデータ構造にとどめることができる．

例えば，従来の手法では以下のコードを記述すると，タスク配列が一つ生成される．

```
TaskArray.new(n, 'sim', 1..n)
```

従来手法では，図 2.4(a) のように個々のタスクオブジェクトを生成し，それらを要素とするタスク配列が作られる．しかし，与えられた配列サイズと実行時引数を展開せずに保持する縮約状態では図 2.4(c) のように少ないオブジェクト数で表現することができる．一部不規則な要素を持つ場合は，そのオブジェクトのみを生成しハッシュで保持する．図 2.4(b) は先頭と末尾のタスクのみ-1 を値として持つ場合の例である．

従来のタスク配列は非縮約状態で実装されているが，これらの3種類の表現を使い分けることで生成するオブジェクト数を最小限に抑えられる．また，タスク配列が保持するストリームの接続情報やストリーム配列についても同様の手法で縮約できる．

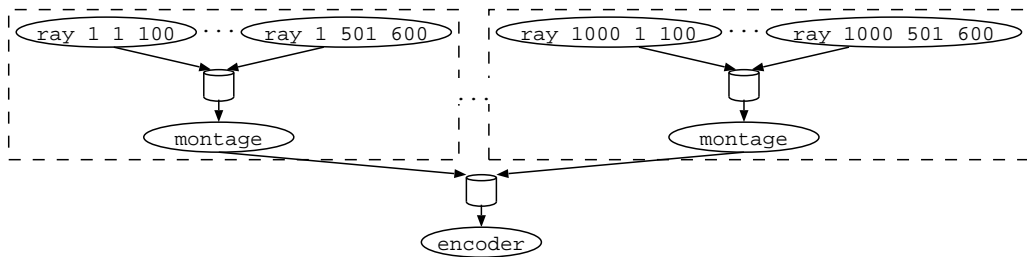


図 2.5: CG レンダリング

## 2.4 階層構造の縮約

2.3 節で述べた縮約手法によりタスク配列に必要なオブジェクトを大幅に削減できるが、階層型ワークフローでは個々の配列単位の縮約で十分な効果が得られない。例えば、図 2.5 に示す CG レンダリングを行うワークフローを考える。このワークフローでは画像を 6 等分し ray タスクを用いて分割レンダリングを行った後、montage タスクによりそれらの部分画像を連結して一枚の画像を生成する。このような生成を 1000 画像分行い、encoder タスクを用いて画像列を連結しエンコードする。

このワークフローにタスク配列の縮約表現を用いると、図 2.6 のようになる。網掛けされたタスク・ストリームは縮約された配列であることを示す。タスク配列の縮約手法やストリーム配列の縮約手法を用いても多数のオブジェクトが生成されてしまう。そのため、配列単位の縮約では効果が低い。

図 2.5 では、各画像を生成するサブワークフロー（点線で囲んだ部分）が繰り返し現れる。この繰り返されているサブワークフローは ray タスクの第 1 引数を除き同一構造である。このようなサブワークフローを一種のタスクオブジェクトのように扱えるようにできれば、サブワークフローの繰り返しは配列で扱えるため縮約が可能となる。しかし、このような同種構造のサブワークフローを自動で検知することは困難である。そこで、サブワークフローを記述するためのクラスを用意して、階層的なワークフローの縮約を実現する。図 2.5 のサブワークフローをオブジェクト化し、階層構造を縮約した結果を図 2.7 に示す。

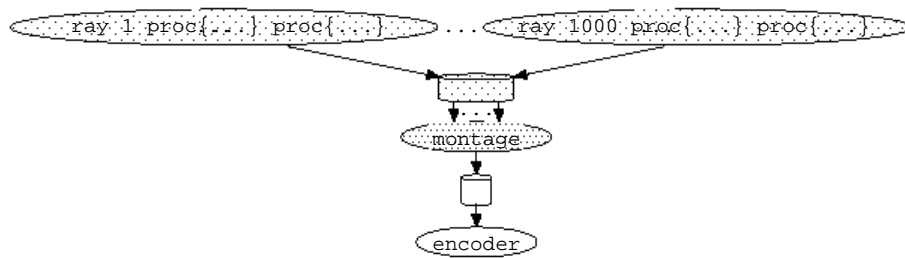


図 2.6: にタスク配列の縮約を適用した結果

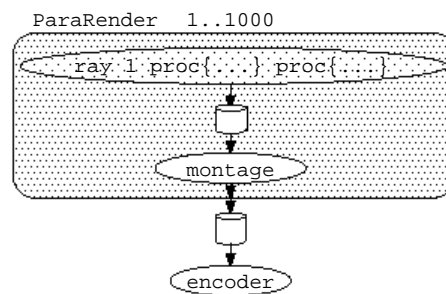


図 2.7: 階層型ワークフローの縮約

### 3 階層構造の縮約手法の設計

#### 3.1 TaskNet クラス

2.4 節で述べたように、階層構造を縮約するために、サブワークフローを記述するクラスが必要になる。そこで、API クラス `TaskNet` を新たに用意する。しかし、サブワークフローの構造をオブジェクトの属性として定義することが難しい。また、サブワークフローを定義する方法が従来の記法と大きく異なれば、仕様が複雑になりユーザへの負担が大きくなる。そこで、従来の記法をサブワークフロー構造の定義でも使えるようにするために、`TaskNet` クラスに `struct` メソッドを用意する。ユーザは `TaskNet` を継承したサブクラス `MyTaskNet` を定義し、`struct` メソッドを再定義する。そして、ユーザは従来の記法でその再定義したメソッドにサブワークフロー構造を記述する。MegaScript 処理系は `MyTaskNet` クラスのオブジェクトを生成するたびにこのメソッドを呼び出すことで、目的のサブワークフローを生成する。

サブワークフローの内部のタスクと外部のストリームを接続するために TaskNet クラスに connect メソッドを用意した。再定義した struct メソッド内でタスクを引数として connect メソッドを呼び出すことで、そのタスクは外部のストリームとの接続を宣言することができる。struct メソッド内のコードはこの connect メソッドを記述する点を除き、ユーザは従来の記法でサブワークフローを定義することができる。また、struct メソッドは引数をとることもでき、サブワークフロー内の構造を記述する際にこの引数を用いることが可能である。

タスクの繰り返しをタスク配列として記述できるように、サブワークフローの繰り返しも TaskNet の配列として記述することができる。TaskNet クラスのオブジェクトを擬似的にタスクオブジェクトとして扱い、タスク配列で処理できるようにする。実際のコードでは、タスクの繰り返しをタスク配列として記述する場合、TaskArray クラスの new メソッドの第 2 引数に実行ファイル名 (文字列) を与える。一方、サブワークフローの繰り返しを TaskNet の配列として記述する場合、TaskArray クラスの new メソッドの第 2 引数に *MyTaskNet* を与える。第 3 引数以降は *MyTaskNet* の new メソッドに渡され、最終的に struct メソッドの引数として与えられる。

TaskNet を用いることで、図 2.5 のワークフローは図 3.8 のように記述できる。図 2.5 は 2.4 節で説明したように CG レンダリングを行うワークフローである。実際にレンダリングを行う ray タスクは実行においてフレーム番号が必要になる。そこで、パラメータ化したフレーム番号を TaskArray クラスの new メソッドの第 3 引数に与える (11 行目)。これにより各 ParaRender オブジェクトの生成時に struct メソッドが呼ばれ、引数として各フレーム番号が与えられる。この引数を使うことで、ray プログラムの実行時引数にフレーム番号を与えることが可能になる。

このように TaskNet を用いることで、サブワークフローの部品化が容易になり、階層型並列性を持つワークフローを簡潔に記述する事が可能になる。また、このクラスの配列を縮約することでメモリ効率が向上する。

## 3.2 階層構造の縮約

サブワークフローの縮約手法について述べる。3.1 節で述べたように、MegaScript プログラムはサブワークフローを定義するために TaskNet を継承したクラスを宣言する必要がある。本節ではそのクラスを *MyTaskNet*

```

1. class ParaRender < TaskNet
2.   def struct(frame)
3.     ray = TaskArray.new(6, "ray", frame,
4.       proc{|i| i*100+1}, proc{|i| i*100+100})
5.     montage = Task.new("montage")
6.     s = Stream.new
7.     s.connect(ray, IN)
8.     s.connect(montage, OUT)
9.     self.connect(montage, OUT)
10.  end
11. end
12. render = TaskArray.new(1000, ParaRender, 1..1000)
13. encoder = Task.new("encoder")
14. s = Stream.new
15. s.connect(render, IN)
16. s.connect(encoder, OUT)

```

図 3.8: TaskNet を用いた MegaScript コード

とする．そして，サブワークフロー構造を定義するためには，`struct` メソッドを再定義しそのメソッド内にサブワークフロー構造を記述する．そして，MegaScript 処理系がこの `struct` メソッドを呼び出したときに，`struct` メソッド内のオブジェクトが初めて生成される．現状の MegaScript プログラムは *MyTaskNet* の外から `struct` メソッド内のオブジェクトにアクセスできない仕様になっている．これは MegaScript プログラムが複雑になることを防ぐためである．このことから `struct` メソッド内で生成されるオブジェクトの存在はユーザから隠蔽される．従って，最初は与えられた引数だけを保持しておき，ワークフロー実行中に必要になったサブワークフローのみを適宜生成することでメモリ使用量を削減できる．

しかし，`struct` メソッドを実行するまで，サブワークフロー内のタスクやストリームの情報やサブワークフローの構造を取得することができない．MegaScript ではスケジューリングを行う際，タスクの情報やサブワークフローの構造を必要とする．そのため，`TaskArray` の要素である *MyTaskNet* オブジェクトを全て縮約すると，適切なタスクスケジューリングが行えない．そこで，*MyTaskNet* オブジェクトを 1 個だけ生成する．他のサブワークフローの構造もこの *MyTaskNet* オブジェクトが表しているサブワークフローと同種構造である．従ってこの *MyTaskNet* オブジェクトからサブワークフロー構造を取得することでスケジューリングが行

える．この手法は配列毎に 1 つのサブワークフローを生成するだけでよい．従って配列の大きさによらず定数オーダーのメモリ使用量で表現することができる．

## 4 縮約手法の実装

### 4.1 実装上の問題

タスクオブジェクトはタスクプロセスを生成し実行するときに必要なとなる。そのため、縮約されたタスク群はタスクプロセス生成時に展開しなければならない。MegaScript 処理系のメモリ使用量を少なくするためには同時に実行するタスクプロセス数を制御して、大部分のタスク群を縮約した状態で管理できるようにする必要がある。

しかし、従来の MegaScript 処理系はタスク間のデータ通信を行うために一度に全てのタスクプロセスを生成する。既存のストリームは入力側の接続タスクプロセスより読み込んだデータを宛先のタスクプロセスに次々と転送する。もし宛先のタスクプロセスがまだ生成されていなければ、転送されてきたデータを受け取るタスクプロセスがないためにデータを破棄するしかない。そのため、ストリームの入力側のタスクプロセスと出力側のタスクプロセスは同時に生成されていなければならない。一般的に、ワークフロー型並列処理では各タスクは上位のタスクに依存していることが多い。そのため、最上位のタスクプロセスが生成された段階で次々と下位のタスクプロセスも生成され、結果全てのタスクプロセスが生成されることになる。このことから、スレーブホスト上では縮約されているオブジェクトを一度に全て展開してしまう問題がある。そこで、データを受信するタスクプロセスが生成されていなくても、正常にタスク間通信ができるデータ通信機構の手法を提案する。この手法により、縮約手法を有効利用することができるため、大規模ワークフローの実行が可能になる。

### 4.2 データ通信機構

基本方針として、送信された出力データをバッファに格納し、再度の転送要求にも応えることができるようにする。上記のデータ通信機構を実現するためには、次のことを考える必要がある。

- (1) ストリームの入出力端をいつ生成するか
- (2) どのホストがバッファ機能のあるストリーム入出力端を持つか
- (3) 縮約されているタスク/ストリームをどう扱うか

一対多，多対一，多対多それぞれのタスク間通信を例を用いて説明する．(3)の問題については，4.2.4 節で述べる．

#### 4.2.1 一対多のタスク間通信

一対多のタスク間通信の構造である図 4.9 を例に説明する．(1)の問題について，データ送信元であるタスク  $a$  のプロセスを生成するときに，その通信路であるストリームを生成する．(2)の問題について，代表出力端では，データの再度の転送要求に応えるために出力データを保持するバッファが必要となる．一方，他の出力端にもバッファを持たせることは，利点と欠点がある．利点は，通信回数を減らせる可能性があることである．図 4.10 のタスク  $b[k]$  (但し， $0 \leq k \leq i-1$ ) のプロセス生成が遅れたときを例に考える．全出力端にバッファを設け，代表出力端のミラーを出力端のバッファに持たせる．この事によって，タスク  $b[k]$  はホスト 1 にデータ転送を依頼しなくても，自ホスト内のバッファから取ってくることで通信処理を行わずに解決できる．欠点は，複数ホストにバッファを持たせる，メモリ資源を余分にとってしまうことである．そこで，遅れてタスクプロセスを生成し出力データを要求した場合，代表出力端のバッファにあるデータを転送するときにそのホストの出力端にバッファを生成し，代表出力端にあるバッファ内のデータをコピーする．このようにすることで，二度目以降は代表出力端のあるホストにバッファデータの転送要求を出さずとも，自ホスト内のバッファからデータを転送することで要求を満たすことができる．この処理によって，実行速度の向上と処理系で占有するメモリ量の削減の両立をはかる．

以上のことをまとめると，一対多のタスク間通信の処理は以下の通りである．

- (i) タスク  $a$  のタスクプロセスを生成するとき，ストリームを生成する
- (ii) 出力端バッファのデータ転送要求が発生したとき，そのホストの出力端にバッファを生成する
- (iii) データ出力側のタスクプロセスがデータを出力する度に，代表出力端に転送する
- (iv) 代表出力端を経由して受信タスクがあるホストに転送する

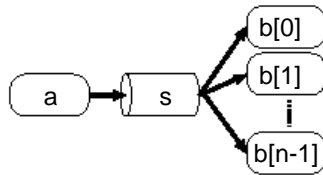


図 4.9: 一対多のワークフロー

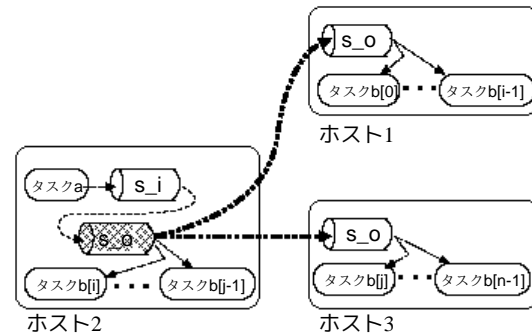


図 4.10: 一対多の従来実装手法

図 4.11 の例を用いて，処理の流れを説明する．各ホストにタスクを配置する前に，スケジューリング結果から受信タスクがもっとも多く配置されるホスト 2 に生成される出力端を代表出力端に決定する．そして，実際にタスクを配置した場合，図 4.11 のようになる．最初に，タスク  $a$  のプロセスが生成されるとする．(i) よりストリームの入出力端を生成する．このときに代表出力端は，バッファの生成を併せて行う．(iii) よりタスク  $a$  の出力は，代表出力端に集められ，バッファに格納される．次に，ホスト 2 のタスク  $b[2]$  とホスト 3 のタスク  $b[5]$  のプロセスが生成されるとする．(iv) よりタスク  $b[2]$  には代表出力端に流れた出力データを転送する．また，出力データは代表出力端を経由してホスト 3 の出力端に転送される．(ii) よりホスト 3 の出力端にバッファを生成し，バッファ内に出力データを格納する．そして，バッファ内のデータをタスク  $b[5]$  に転送する．タスク  $a$ ， $b[2]$ ， $b[5]$  のプロセスの終了後，それぞれのホストにあるタスク  $b[0]$ ， $b[3]$ ， $b[6]$  のプロセスが生成されるとする．タスク  $b[0]$ ， $b[3]$ ， $b[6]$  は自ホスト内のバッファにタスク  $a$  の出力データが保持されているため，そのバッファからタスク  $a$  の出力データが転送される．自ホストにあるバッファ内のデータを転送することで，ホスト間の通信処理を出来る限り行わずに出力データの転送ができる．次に，タスク  $b[0]$ ， $b[3]$ ， $b[5]$  のプロセスの終了後，タスク  $b[1]$ ， $b[4]$  のプロセスが生成されるとする．タスク  $b[1]$  とタスク  $b[4]$  も同様の処理で自ホスト内のバッファから出力データが転送される．以上の操作により，タスク  $a$  の出力データがすべてのタスク  $b$  に転送されることになる．

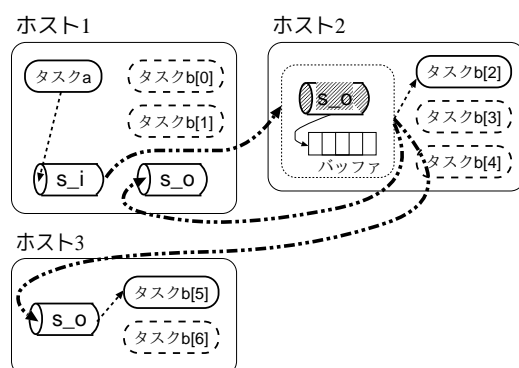


図 4.11: 一対多のタスク間通信の例

#### 4.2.2 多対一のタスク間通信

多対一のタスク間通信の構造を図 4.12 を例に説明する。(1)の問題について、データ送信タスクであるタスク  $a$  のプロセスのいずれかを生成するときに、ストリーム入出力端を生成する。図 4.12 ではタスク  $a$  が複数ある。そこで、ストリームに入出力端の生成フラグを設定し、同じストリーム入出力端を複数生成しないように制御する。例えば  $a[0]$  が最初にプロセス生成をする場合、ストリーム入出力端の生成を行い、ストリーム入出力端のフラグを生成済に変更する。 $a[1]$  から  $a[n-1]$  のタスクプロセスを生成しても、ストリーム入出力端のフラグが生成済となっているため、ストリーム入出力端を生成しない。(2)の問題について、ストリーム出力端が一つしかないため、必然的にタスク  $b$  が配置されるホストの出力端が代表出力端となる。以上より、多対一のタスク間通信の処理は、一対多のタスク間通信の処理の (i) に複数のストリームを生成しない操作を追加するだけで良い。

図 4.13 を例に、処理の流れを説明する。各ホストにタスクを配置する前に、スケジューリング結果から受信タスクが配置されるホスト 2 のストリーム出力端を代表出力端に決定する。そして、実際にタスクを配置する。最初に、各ホストに配置されているタスク  $a[0]$ ,  $a[3]$ ,  $a[5]$  のプロセスが生成されるとする。(i) より、ストリームを生成する。併せて代表出力端はバッファの生成を行う。(iii) より、タスク  $a[0]$ ,  $a[3]$ ,  $a[5]$  の出力データは、代表出力端に転送し、バッファに格納される。この例では受信タスクであるタスク  $b$  のプロセスが生成されていない。そこで (iv) より、代表出力端からタスク  $b$  へのデータ転送は行われない。タスク  $a[0]$ ,  $a[3]$ ,

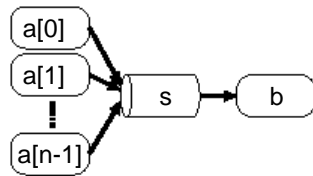


図 4.12: 多対一のワークフロー

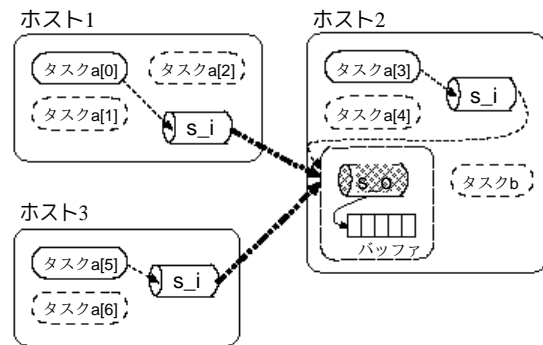


図 4.13: 多対一のタスク間通信の例

$a[5]$  のプロセスの終了後，タスク  $a[1]$ ， $a[4]$ ， $a[6]$  のプロセスが生成されるとする．これらのタスクの出力結果も同様に代表出力端のバッファに格納される．タスク  $a[1]$ ， $a[4]$ ， $a[6]$  のプロセスの終了後，タスク  $a[2]$ ， $b$  のプロセスが生成されるとする．タスク  $b$  がタスク  $a$  群の出力結果の転送を要求する．そこで，代表出力端のバッファのデータをタスク  $b$  に転送する．そして，タスク  $a[2]$  の出力データも同様に代表出力端に転送し，バッファに格納後，タスク  $b$  に転送する．以上の操作により，タスク  $a$  群の出力データがすべてのタスク  $b$  に転送されることになる．

#### 4.2.3 多対多のタスク間通信

多対多のタスク間通信の構造は図 4.14 である．この構造は一对多と多対一を組み合わせたものであり，それぞれのタスク間通信と同じ処理で多対多のデータ通信ができる．

図 4.15 を例に説明をする．まず，代表出力端となるホストを決定する．今回の例ではどのホストも受信タスク  $b$  の配置される数が同じであるため，ホスト 1 の出力端を代表出力端とする．最初にタスク  $a[0]$ ， $a[1]$ ， $a[2]$  のプロセスが生成されるとする．その時点でストリーム入出力端を生成する．タスク  $a[0]$ ， $a[1]$ ， $a[2]$  のそれぞれの出力データは，代表出力端に集められバッファに格納される．タスク  $a[0]$ ， $a[1]$ ， $a[2]$  のプロセス終了後，タスク  $b[0]$ ， $b[2]$ ， $b[4]$  のプロセスが生成されるとする．タスク  $b[0]$  の実行ホストは代表出力端のあるホストと同じであることから，代表出力端のバッファに格納されている出力データが転送される．一方，タスク  $b[2]$ ， $b[4]$  は代表出力端と異なるホストで実行されるため，ホスト 2 とホスト 3 のストリーム出力端に出力データを転送する．このときに，ホス

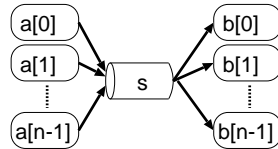


図 4.14: 多対多のワークフロー

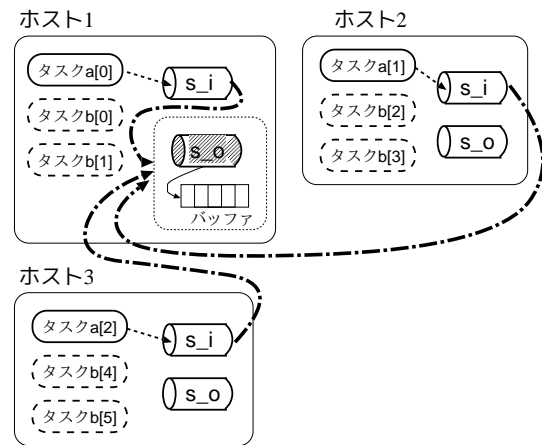


図 4.15: 多対多のタスク間通信の例

ト 2 とホスト 3 はそれぞれバッファを生成し，出力データをバッファに格納する．各ホストにあるバッファのデータをタスク  $b[2]$ ， $b[4]$  に転送する．タスク  $b[0]$ ， $b[2]$ ， $b[4]$  のプロセス終了後，タスク  $b[1]$ ， $b[3]$ ， $b[5]$  のプロセスも同様の手順で生成し，データを転送する．以上の操作でタスク  $a$  群の出力データをすべてのタスク  $b$  群に転送することができる．

#### 4.2.4 タスクの縮約

縮約されたタスク群はオブジェクトの転送時間および必要メモリ量を削減するために，各ホストに縮約状態のままタスクを転送することが望ましい．しかし，縮約されたタスク群はスケジューリングの結果によって複数のホストに分割して配置される可能性がある．例えば図 4.16 の縮約状態のタスク群が，タスク  $[0]$  からタスク  $[i-1]$  をホスト 1 に，タスク  $[i]$  からタスク  $[j-1]$  がホスト 2 に，タスク  $[j]$  からタスク  $[n-1]$  がホスト 3 にスケジューリングされるとする．このスケジューリング結果を満たしつつ縮約状態を展開せずにタスク転送を行うために，縮約されたタスク群を指定位置で分割する機能を追加する．例では，タスク  $[0]$  からタスク  $[i-1]$  の縮約表現のタスク群に，タスク  $[i]$  からタスク  $[j-1]$  の縮約状態のタスク群に，タスク  $[j]$  からタスク  $[n-1]$  の縮約状態のタスク群に分割している．それぞれのタスクオブジェクトは縮約状態を維持しつつ，各ホストに転送する．各ホストで縮約されたタスクは必要とされるタスクのみ展開しタスクプロセスを生成するようにすれば，タスクオブ

ジェットの転送時間および必要メモリ量を減らすことが出来る．

タスク群が縮約されていた場合の一对多のタスク間通信を，図 4.17 を例に説明する．スケジューリング結果から，受信タスクであるタスク  $b$  が最も多く配置されるホスト 1 を代表出力端とし，各ホストに縮約されたタスク  $b$  群を配置する．最初に，タスク  $a, b[i], b[j]$  のプロセスが生成されるとする．タスク  $b[i]$  と  $b[j]$  は縮約表現の一部となっているため，部分展開が必要となる．また，部分展開した後，親である縮約状態のタスク  $b$  群からタスク間の接続情報を持っているストリームを調べ，出力端の接続先をタスク  $b[i]$  に変更する．タスク  $b[j]$  に関しても出力端の接続先を変更する．その後，タスク  $b[i]$  とタスク  $b[j]$  のプロセスを生成する．タスク  $b[i]$  とタスク  $b[j]$  がタスク  $a$  の出力データを要求するため，ホスト 2 とホスト 3 のストリーム出力端にバッファを生成し，代表出力端のバッファに格納されている出力データをホスト 2 とホスト 3 のバッファに転送する．そして，自ホスト内にあるバッファのデータをタスク  $b[i]$  とタスク  $b[j]$  に転送する．タスク  $a$  とタスク  $b[i]$ ，タスク  $b[j]$  のプロセスが終了したら，タスク  $b[0]$ ，タスク  $b[i+1]$ ，タスク  $b[j+1]$  がプロセス生成されるとして，これらのタスクを部分展開し，ストリームの接続先を変更し，プロセス生成を行う．タスク  $a$  の出力データを自ホスト内にあるバッファから，タスク  $b[0]$  とタスク  $b[i+1]$ ，タスク  $b[j+1]$  に転送する．タスク  $b[0]$  とタスク  $b[i+1]$ ，タスク  $b[j+1]$  のプロセスが終了したら，残りのタスクも順次展開して，自ホスト内のバッファにあるタスク  $a$  の出力データを渡すことで，すべてのタスク  $b$  にタスク  $a$  の出力データを転送することができる．

多対一，多対多のタスク間通信も同様に必要時に展開・生成することで非縮約の転送と同じ処理でタスク間の通信ができる．

#### 4.2.5 ストリームの縮約

MegaScript で記述したワークフローには，図 4.18 のような接続関係がよく出現する．このワークフローを縮約すると，図 4.19 のように表現でき，タスク群とストリーム群は縮約状態のまま接続関係を保持することができる．また，タスク  $a$  群とタスク  $b$  群をスケジューリングすると，縮約状態のまま纏めて転送することによるオブジェクトの転送時間とスケジューリングコストの削減の観点からタスク  $a[k:l]$  群とタスク  $b[k:l]$  群（但し， $0 \leq k \leq n, k \leq l \leq n$ ）は同じホストに配置される．そこで，縮約状態のストリーム  $s[0:n]$  群を分割してストリーム  $s[k:l]$  群を生成し，同じホストに配置し，そして，タスク間通信の処理で，ストリームを部分

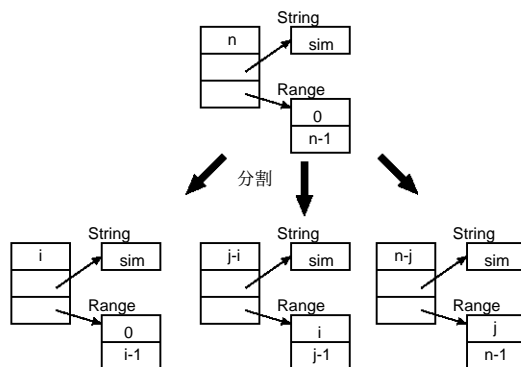


図 4.16: 縮約タスクの分割

展開する．そのストリームの入出力端に接続するタスクを設定し，ストリーム入出力端を生成する．タスク群も適宜，部分展開しプロセスを生成する．ストリームの入出力端から自ホスト内にあるタスク  $b$  へデータを転送する．

図 4.20 を例に説明する．スケジューリング結果に従いタスク  $a$  群とタスク  $b$  群を分割する．縮約されたタスク群を各ホストに転送する．タスク  $a[0 : i - 1]$  群とタスク  $b[0 : i - 1]$  群，及びストリーム  $s[0 : i - 1]$  群をホスト 1 に，タスク  $a[i : j - 1]$  群とタスク  $b[i : j - 1]$  群，及びストリーム  $s[i : j - 1]$  群をホスト 2 に，タスク  $a[j : n - 1]$  群とタスク  $b[j : n - 1]$  群，及びストリーム  $s[j : n - 1]$  をホスト 3 に配置したとする．最初にホスト 1 はタスク  $a[0]$  のプロセスが生成されるとする．タスク  $a[0 : i - 1]$  群から部分展開しタスク  $a[0]$  のプロセスを生成する．そして，そのプロセスで必要とされるストリーム  $s[0]$  のオブジェクトを部分展開し，ストリーム入出力端を生成する．タスク  $a[0]$  の出力データを出力端  $s_o[0]$  のバッファに格納する．ホスト 1 は次に，タスク  $b[0]$  のプロセスが生成されるとする．タスク  $b[0 : i - 1]$  群から部分展開しタスク  $b[0]$  のプロセスを生成する．出力端のバッファから出力データをタスク  $b[0]$  のプロセスに転送する．順次，タスク  $a[0 : i - 1]$  群，タスク  $b[0 : i - 1]$  群，ストリーム  $s[0 : i - 1]$  を展開していき，データ転送を行う．ホスト 2，3 もホスト 1 の処理と同様にタスク  $a$  群とタスク  $b$  群，及びストリーム  $s$  群を順次部分展開して，ストリーム入出力端を生成し，各タスク  $a$  の出力データをタスク  $b$  に転送する．

以上の処理を行うことで，タスクやストリームが縮約されていても，適

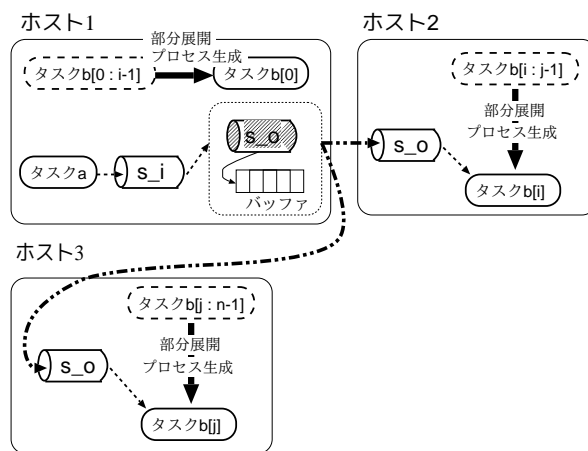


図 4.17: 縮約状態の一对多のタスク間通信の例

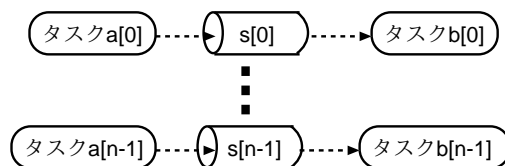


図 4.18: 一对一のワークフロー例

切にタスク間の通信ができる．

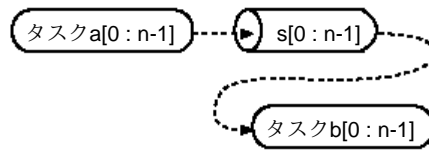


図 4.19: 縮約した一対一のワークフロー例

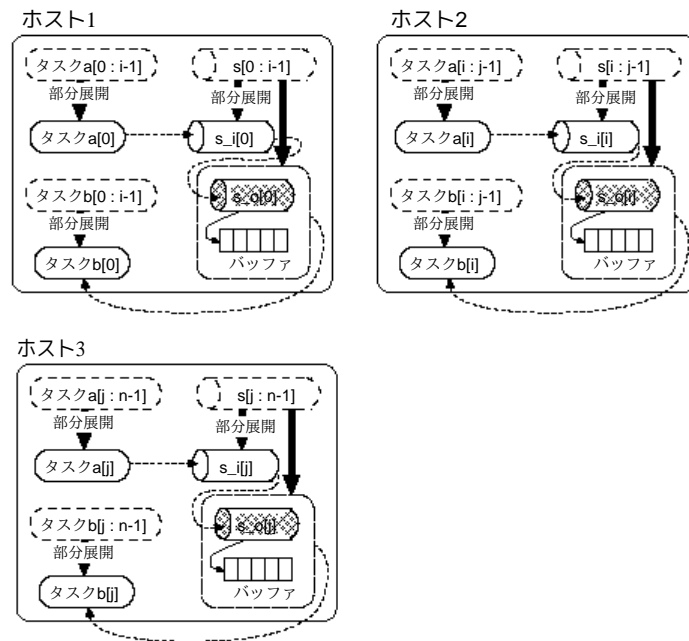


図 4.20: (縮約あり) 一対一のタスク間通信の例

## 5 評価

本評価では、CPU に Intel®Xeon®Processor X3330(2.66GHz , L2 cache 6MByte) , 2GByte のメモリを搭載した PC クラスタを利用した。また、ネットワーク環境は GigabitEthernet で評価を行った。

### 5.1 実ワークフローのメモリ使用量

縮約表現の効果を評価するために、実ワークフロー内で利用されるアプリケーション (表 5.1) のタスク配列に対し、必要なメモリ量を測定し

表 5.1: 評価アプリケーション

|            |  |
|------------|--|
| povray     | 1 フレーム/タスクで $N$ フレームの 3DCG 画像をレンダリングする |
| blastn     | 1 クエリ/タスクでヌクレオチド DB から $N$ 個のクエリの探索を行う |
| mProjectPP | 1 画像/タスクで $N$ 個の画像を指定サイズにスケーリングする      |

表 5.2: タスク配列の必要メモリ量 (bytes)

| タスク数    | povray     |      | blastn     |      | mProjectPP |      |
|---------|------------|------|------------|------|------------|------|
|         | 従来手法       | 提案手法 | 従来手法       | 提案手法 | 従来手法       | 提案手法 |
| 100     | 25,820     | 582  | 20,420     | 454  | 21,620     | 484  |
| 1,000   | 258,020    | 583  | 204,020    | 454  | 216,020    | 484  |
| 10,000  | 2,580,020  | 584  | 2,040,020  | 454  | 2,160,020  | 484  |
| 100,000 | 25,800,020 | 585  | 20,400,020 | 454  | 21,600,020 | 484  |

た．各アプリケーションについて，タスク配列 1 個で表されるパラメータスイープ型のワークフローを MegaScript で記述し，これを従来の処理系と提案手法を実装した処理系で実行したときに生成されるオブジェクトの大きさの合計を求めてメモリ使用量とした．

結果を表 5.2 に示す．従来手法ではタスク配列がすべて非縮約表現になるため，必要メモリ量はタスク数  $N$  にほぼ比例し，10 万タスクで 20MByte を超える．一方，タスク配列の縮約手法ではタスク配列が完全縮約表現になるため，タスク数によらず 600 バイト以下になる．

また，階層構造の縮約手法の効果を評価するために，実アプリケーションのワークフローに CG レンダリング (図 2.5)，天文画像を合成する Montage(図 5.22(a))，遺伝子解析を行う Epigenomics(図 5.22(b)) を用いてメモリ使用量を測定した．

結果を表 5.3 に示す．CG レンダリングと Epigenomics については完全縮約表現が可能である．このため，API オブジェクト数・必要メモリ量のいずれもワークフローの規模によらず，前者が 10 個前後，後者が 2–3KB 程度と非常に大きな削減効果が得られた．一方，Montage については個数が入力画像数のオーダのタスク群が 3 組あり，うち 2 組の間の入出力関係が不規則性を持つ．今回はこれが完全にランダムであるとして評価を行ったため，従来手法に対し，オブジェクト数で 4 割，必要メモリ量で 3

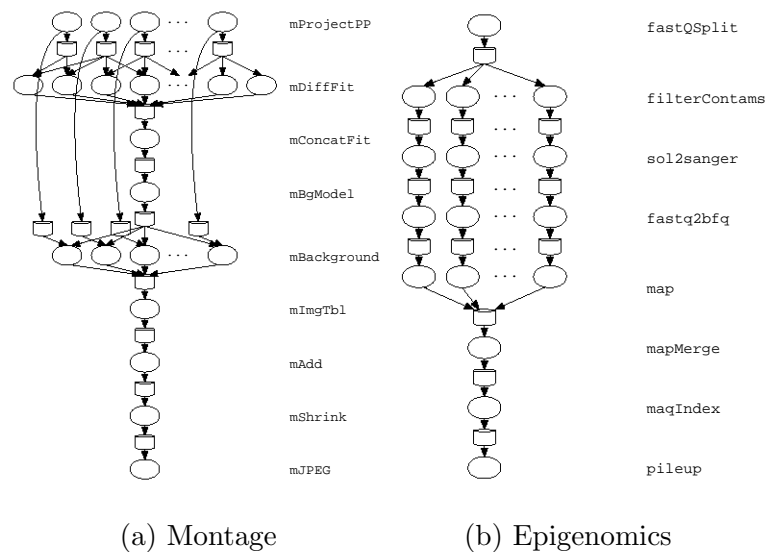


図 5.21: 評価に用いた実ワークフロー

割の削減にとどまった．しかし，入力画像数が増えると，規則性がない部分の設定が非常に手間になる．そのため，入力画像群にある程度の規則性を持たせるようになることが期待できる．

## 5.2 実行時間の評価

提案した通信機構の性能を評価するために MegaScript の実行時間を測定した．本実験は大量のタスクを実行するときに消費するメモリ量が与える実行速度への影響を調査するために，タスク間通信を少なくした．本評価に用いたワークフロー構造を図 5.22 に示す．output\_program は 3 バイトの文字列を出力するだけのプログラムである．また，array\_foreach は output\_program からデータを受け取り，そのデータを破棄してから巨大な配列 (10, 20, 30MByte のメモリを消費する配列) の各要素を順次参照するプログラムである．評価結果を表 5.4 に示す．また，評価結果のグラフを図 5.23，図 5.24，図 5.25 に示す．

提案手法の実行速度は従来手法よりも 1 割程度低下した．提案手法は現在実行しているタスクプロセスの終了を待ってから次のタスクプロセスを生成している．このオーバーヘッドがあるため，従来手法よりも実行速度が低下してしまったと考えられる．しかし，提案手法ではタスク

表 5.3: 実アプリケーションのワークフローの縮約結果

|               | API オブジェクト数        |         |         | 必要メモリ量 (bytes) |            |         |
|---------------|--------------------|---------|---------|----------------|------------|---------|
|               | 従来手法               | 提案手法    | 削減率 (%) | 従来手法           | 提案手法       | 削減率 (%) |
| <b>フレーム数</b>  | <b>CG レンダリング</b>   |         |         |                |            |         |
| 100           | 1,003              | 7       | 99.30   | 222,159        | 1,957      | 99.12   |
| 1,000         | 10,003             | 7       | 99.93   | 2,218,359      | 1,957      | 99.91   |
| 10,000        | 100,003            | 7       | 99.99   | 22,180,359     | 1,957      | 99.99   |
| 100,000       | 1,000,003          | 7       | 100.00  | 221,800,359    | 1,957      | 100.00  |
| <b>入力画像数</b>  | <b>Montage</b>     |         |         |                |            |         |
| 100           | 518                | 318     | 38.61   | 119,400        | 82,293     | 31.08   |
| 1,000         | 5,018              | 3,018   | 39.86   | 1,172,400      | 788,793    | 32.72   |
| 10,000        | 50,018             | 30,018  | 39.99   | 1,1702,400     | 7,853,793  | 32.89   |
| 100,000       | 500,018            | 300,018 | 40.00   | 117,002,400    | 78,503,793 | 32.90   |
| <b>データ分割数</b> | <b>Epigenomics</b> |         |         |                |            |         |
| 100           | 809                | 17      | 97.90   | 150,143        | 3,261      | 97.83   |
| 1,000         | 8,009              | 17      | 99.79   | 1,488,443      | 3,261      | 99.78   |
| 10,000        | 80,009             | 17      | 99.98   | 14,871,443     | 3,261      | 99.98   |
| 100,000       | 800,009            | 17      | 100.00  | 148,701,443    | 3,261      | 100.00  |

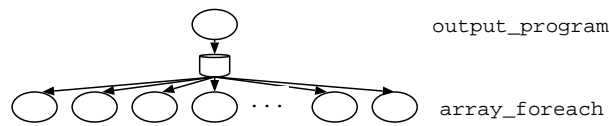


図 5.22: 一対  $N$  のワークフロー構造

数が増えても安定して実行することができたのに対して，従来手法はタスク数が増えて全てのタスクプロセスに十分なメモリを割り当てることができなくなると実行が困難になった．例えば，配列のメモリ使用量が 10MByte， $N$  が 800 では実行時間が急増した．また， $N$  が 1000 になると実行できなくなった．以上より，大規模ワークフローの実行は提案手法の方が適していることが分かる．今後，同時に実行するタスクプロセスを最適化し，提案手法のオーバーヘッドを低減する必要がある．

本実験では，提案手法に縮約手法を利用した場合と利用しない場合で実行時間に大きな違いは見られなかった．これはタスク数が少なく MegaScript 処理系のメモリ使用量が問題にならなかったためである．タスク数が多くなり MegaScript 処理系のメモリ使用量が大きくなると縮約手法の効果が期待できる．

| 表 5.4: 実行時間 (秒) |                   |         |
|-----------------|-------------------|---------|
|                 | 従来手法              | 提案手法    |
| $N$             | 配列のメモリ使用量:10MByte |         |
| 100             | 427.1             | 434.9   |
| 200             | 855.6             | 862.7   |
| 400             | 1779.9            | 1878.9  |
| 600             | 2919.0            | 2944.3  |
| 800             | 9543.1            | 3996.8  |
| 1000            | -                 | 5069.9  |
| $N$             | 配列のメモリ使用量:20MByte |         |
| 100             | 851.5             | 867.3   |
| 200             | 1780.4            | 1872.4  |
| 400             | 6774.8            | 4000.5  |
| 600             | -                 | 6133.4  |
| 800             | -                 | 8256.6  |
| 1000            | -                 | 10369.9 |
| $N$             | 配列のメモリ使用量:30MByte |         |
| 100             | 1284.9            | 1378.0  |
| 200             | 2680.3            | 2932.4  |
| 400             | -                 | 6147.6  |
| 600             | -                 | 9324.1  |
| 800             | -                 | 12509.0 |
| 1000            | -                 | 15697.4 |

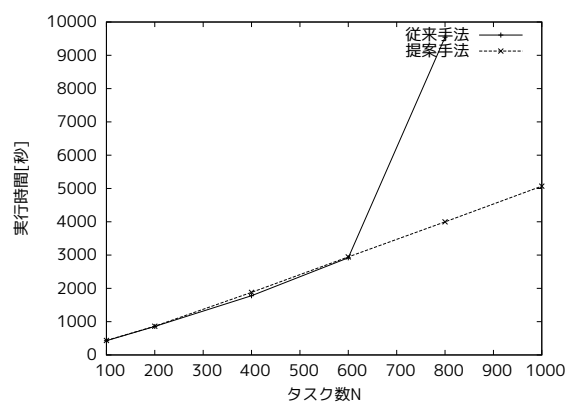


図 5.23: 10MByte の配列

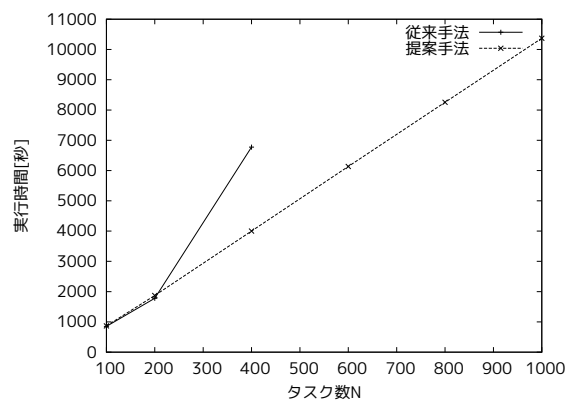


図 5.24: 20MByte の配列

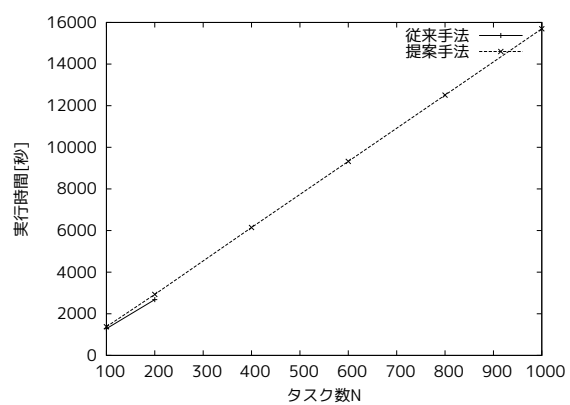


図 5.25: 30MByte の配列

## 6 関連研究

Pegasus, Swift, GXP Make など、大規模なワークフローの実行に用いた例が報告されている。Pegasus ではユーザが DAG として記述した抽象ワークフローから、実資源へのマッピングやデータ転送などを追加した実ワークフローを生成する。実行時には DAGMan や Condor-G を用いて、依存解決の判定や個々のタスク投入を行う。ワークフローが大量の細粒度タスクを含む場合、個々のタスク投入オーバーヘッドが無視できないほど大きくなるため、複数のプログラム実行を静的に 1 タスクにまとめて粒度を上げるタスククラスタリングの機能が提供されており、DAG 上のレベル (根からの距離) やユーザが指定するラベルを基準とするクラスタリングをサポートしている。また *condor glidein*[14] の機能を使ったオーバレイスケジューリングも提案されている。これは、タスクとして投入された glidein プロセスが実行を開始した後、これと直接通信して目的のタスク群を実行させる手法である。Condor-G などのタスクディスパッチャをバイパスすることにより個々のタスクの投入手続きを省略できると同時に、目的のタスク群のスケジューリングを上位層で制御できる。タスククラスタリングやオーバレイスケジューリングは、80 万タスクからなる Cybershake ワークフローの実行で高い効果が得られたことが報告されている [17]。

Swift は MegaScript と同様にワークフロー記述を目的とするスクリプト言語であり、foreach 構文と配列を組み合わせることで並列タスクの集合を容易に記述できる。Swift プログラムは Karajan[15] プログラムに変換され、最終的に Globus toolkit[16] や Condor-G などを用いて実行される。このため実行時の機構は Pegasus に類似しており、タスククラスタリングや coaster と呼ばれるオーバレイスケジューリングの機能がサポートされている。また、機能を単純化したディスパッチャをオーバレイスケジューリング方式で動作させる Falkon も提案・実装され、タスク投入オーバーヘッドを大幅に削減して高いスループットとスケーラビリティを得ている [18]。しかし、Pegasus や Swift のタスククラスタリングはまとめたタスク群の粒度を動的に変更できない。そのため、ホストの性能変動に併せたタスク単位の動的負荷分散ができない。また、オーバレイスケジューリングは上位層で扱うタスク数を減らすことができない。一方、MegaScript は処理系の判断で縮約されたタスク群の粒度を変更できるため、動的な性能変動にも柔軟に対応できる。そして、個々のタスクの実行直前まで縮約することが可能なため上位層から下位層の全処理過

程で縮約の効果が得られる．

GXP Make は Makefile の書式によりワークフローを記述し，GNU make によりタスクの依存関係や実行可能判定・タスクの並列発行を行う．make が発行したタスクはディスパッチャへと送られ，各ホスト上の GXP デモンにより実行される．make が発行するタスクを処理系が直接実行するため，Pegasus や Swift のように一度 XML に変換してミドルウェアに実行させる方式と比べるとオーバーヘッドが小さく，100 万タスク規模の実験で Swift+Falkon に匹敵するスループットを実現している．

これらの既存システムの手法のうち，クラスタリングについては，適切な粒度になるようユーザが閾値やラベルを決定したりチューニングを行う作業が必要となる．また，静的にクラスタリングを行うため，見かけ上のタスク数を減らし大域スケジューリングのコストを削減できる効果がある．一方で，実行時のホスト数に応じて粒度を動的に変更することはできない．一方，オーバレイスケジューリングは動的な粒度調整が可能であるが，上位層で扱うタスク数は変わらないためスケジューリングのコストは削減できない．

さらに，いずれの手法も処理系の一部で扱うタスク数を減らすものであり，ワークフローを表す情報自体を縮約する効果はない．このためタスク数が膨大になると実行に支障を来すケースがあり，上記の 80 万タスクからなる Cybershake ワークフローでは XML ファイルが 1GB に達しクラスタリング処理が困難になったことが報告されている [17]．また，Falkon で 200 万タスクを実行する実験ではディスパッチャの Java ヒープサイズを 1.5GB に設定しており [18]，より大規模なワークフローではタスクの情報を保持するためのメモリ消費量が問題となる．

GXP Make については，パラメータスイープ型の類似タスクを MegaScript のように簡潔に書くことができ，Makefile の大きさ自体は MegaScript 同様，一般的な実ワークフローではタスク数に依存しない．しかし実行時には類似タスクについても個々にタスクとして発行されるため，ディスパッチャに送信されるデータの総量はタスク数に比例する．また，GXP Make では GNU make の `-j` オプションにより，並列に発行するタスク数  $P$  を制御するが，発行するタスク毎に子プロセスが生成されタスク完了まで維持されることにより， $P$  の値の上限が制約される．make の性質上，依存元が解決し実行可能になったタスクしか発行されないから，まだ実行可能でないワークフロー下流のタスクや，実行可能であっても発行上限  $P$  を超えるタスクについては，ディスパッチャはその存在を知ること

ができない．したがって，ワークフローの規模が大きくなってもディスプレイのメモリ消費量は増えないが，ワークフロー全体に対する静的スケジューリングを行うことはできない．MegaScript では最初にワークフロー構造全体を構築するため，縮約によりメモリ消費量やスケジューリングコストの問題が解決できれば，大規模なワークフローに対しても静的スケジューリングが可能である．

## 7 おわりに

多くの科学分野において，ワークフローが実用的な大規模並列計算の手段として利用されるようになってきており，広域分散環境上で効率よく実行する手法が必要とされている．しかし，単純な実装ではワークフローの規模に応じて，ワークフロー構造の保持に必要とされるメモリ使用量が大きくなり，実行可能なワークフローの規模がマスターホストのメモリ量などで制約される．この問題を解決するために縮約表現を提案している．

大規模ワークフローを効率的に実行するためには，スレーブホストでもランタイムのメモリ使用量を抑える必要がある．しかし，従来の実装手法ではタスク間通信を行う際，一度に大量のタスクプロセスを生成してしまう．そこで，本論文で提案する通信機構を実装することで縮約されたタスク群から少数のタスクプロセスを生成できるようになる．MegaScript 処理系に本手法を実装し評価した結果，少ないスレーブホスト数でも大規模ワークフローを正常に実行することができた．

提案手法の一对多のタスク間通信で，代表出力端の受信処理に負荷が集中する問題がある．この問題を解決するために，入力端にもバッファを設けて同ホスト内のタスクのメッセージデータを集めて，パッキングし纏めて送信することで受信の負荷を減らすことができる．大規模並列処理を行う実アプリケーションで性能評価を行うことが今後の課題となる．

## 謝辞

本研究を行うにあたり，御指導，御助言頂きました大野和彦講師，並びに多くの助言を頂きました近藤利夫教授，佐々木敬泰助教に深く感謝致します．また，様々な局面にてお世話になりました研究室の皆様にも心より感謝いたします．

## 参考文献

- [1] Kenjiro Taura, Takuya Matsuzaki, Makoto Miwa, Yoshikazu Kamoshida, Daisaku Yokoyama, Nan Dun, Takeshi Shibata, Choi Sung Jun and Jun'ichi Tsujii. Design and Implementation of GXP Make – A Workflow System Based on Make. eScience, IEEE International Conference on, 214-221, 2010.
- [2] Ewa Deelman, Gurmeet Singh, Mei-Hui Sua, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berri-man, John Good, Anastasia Laity, Joseph C. Jacob and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Scientific Programming. 219-237, 2005.
- [3] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz and Ian Foster. Swift: A language for distributed parallel scripting. Parallel Computing. 2011.
- [4] E.Deelman, D.Gannon, M.Shields, and I.Taylor. Workflows and e-science: An overview of workflow system features and capabilities. Future Gener. Comput. Syst., 25:528-540, 2009.
- [5] Y.Gil, E.Deelman, M.Ellisman, T.Fahringer, G.Fox, D.Gannon, C.Goble, M.Livny, L.Moreau, and J.Myers. Examining the challenges of scientific workflows. Computer, 40:24-32, 2007.
- [6] 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, 73-76, May 2003.
- [7] 西里 一史, 大野 和彦, 中島 浩. タスク並列スクリプト言語 MegaScript 向けランタイムシステム. In 情報研報 2004-HPC-99, pages 7-12, July, 2004.
- [8] 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手法. 先進的計算基盤システムシンポジウム SACSIS2004, 135-136, May 2004.

- [9] 大野 和彦, 張 鉄群, 佐々木 敬泰, 近藤 利夫, 中島 浩, 配列の縮退表現による大規模並列タスクネットワークの実装効率化, 情報処理学会第 70 回全国大会講演論文集, pages 5-105 - 5-106, March, 2008.
- [10] 三田 明宏, 佐々木 敬泰, 大野 和彦, 近藤 利夫, 中島 浩大規模タスクネットワークの縮約表現による実装効率化, 平成 22 年度電気関係学会東海支部連合大会, D5-7, October, 2010.
- [11] Kazuhiko Ohno, Akihiro Mita, Masaki Matsumoto, Takahiro Sasaki, Toshio Kondo, Hiroshi Nakashima. Efficient Implementation of Large-scale Workflows based on Array Contraction. Proceedings of the 22th IASTED International Conference on Parallel and Distributed Computing and Systems –PDCS 2010–, 153-162, November 2010.
- [12] 三田 明宏, 仲 貴幸, 松本 真樹, 大野 和彦, 佐々木 敬泰, 近藤 利夫 MegaScript における大規模ワークフローの縮約機構の設計, 情処研報 2011-HPC-130, July 2011.
- [13] まつもとゆきひろ and 石塚圭樹. オブジェクト指向スクリプト言語 Ruby. ASCII, 1999.
- [14] Condor Glide-in. <http://www.cs.wisc.edu/condor/glidein/>.
- [15] G.Laszewski, M.Hategan, and D.Kodeboyina. Java CoG kit workflow. In Workflows for e-Science, pages 340-356. Springer London, 2007.
- [16] I.Foster and C.Kesselman. Globus: A metacomputing infrastructure toolkit. J.Supercomputer Applications, 11:115-128, 1997
- [17] S.Callaghan, P.Maechling, E.Deelman, K.Vahi, G.Mehta, G.Juve, K.Milner, R.Graves, E.Field, D.Okaya, D.Gunter, K.Beattie, and T.Jordan. Reducing time-to-solution using distributed high-throughput mega-workflows - experiences from SCEC CyberShake. eScience, IEEE International Conference on, 0:151-158, 2008.
- [18] I.Raicu, Y.Zhao, C.Dumitrescu, I.Foster, and M.Wilde. Falkon: a fast and light-weight task execution framework. In Proceedings of

the 2007 ACM/IEEE conference on Supercomputing, SC '07, pages  
43:1-43:12, 2007