Vol.2011-HPC-130 No.60
2011/7/29

適応型手法の改良による大規模な実ワークフローの高速スケジューリング

松本 真樹[†1]　大野 和彦[†1]
佐々木 敬泰[†1]　近藤 利夫[†1]

近年，コストパフォーマンスやスケーラビリティの面から，PC クラスタを利用した大規模並列処理への需要が高まっている．特にワークフロー型の大規模並列処理システムにおいて高いスループットを得るには，静的スケジューリング処理が重要になってくる．しかし，高精度な静的スケジューリング手法は計算コストが非常に高い．一方，実アプリケーションはワークフロー内に類似したサブワークフローを複数持つ場合など，ある程度規則的なワークフローとなることが多い．そこで本論文では，このようなサブワークフローに着目しスケジューリングの計算量を削減する手法を提案する．本手法では，個々のサブワークフローを疑似タスクとして扱い，全体のワークフローや個々のサブワークフローに対して個別に我々が開発した適応型スケジューリング手法を用いる．これにより一回の静的スケジューリング手法で扱うタスク数を減らすことができ，高速なスケジューリングが可能となる．抽象シミュレーションにより本手法を評価した結果，タスク数が 10,000 規模の場合に適応型スケジューリング手法の約 1/100 の時間でスケジューリングを行うことができた．

A Fast Adaptive Scheduling Scheme for Large-scale Workflows

Masaki Matsumoto,[†1] Kazuhiko Ohno,[†1]
Takahiro Sasaki[†1] and Toshio Kondo [†1]

Task scheduling is very important for efficient execution of large-scale workflows. However, scheduling large-scale workflows using existing scheduling schemes is not practical because of the huge computational costs.

To solve this problem, we have proposed an adaptive scheduling scheme with low computational cost. However, many practical workflows are collections of sub-workflows and the scheme may not schedule them efficiently. Therefore, we propose a new scheme that improves adaptive scheduling. In our scheme, sub-workflows are replaced to pseudo tasks and an adaptive scheme schedules the workflow which contains pseudo tasks and each sub-workflows separately. The evaluation using an abstract simulation shows the scheduling time of our scheme is approximately 1/100 compared to the adaptive scheme.

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．特に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ため，PC クラスタを利用することに注目が集まっている．

大規模なワークフロー型の並列処理で高いスループットを得るには，実行単位となる各タスクをどの計算機でどの順番で行うかというタスクスケジューリングが重要になる．そのため様々なタスクスケジューリング手法が提案されている．HEFT[1] や LDBS[2] のような静的スケジューリング手法は，ワークフロー型並列処理全体の実行時間であるスケジューリング長において良い結果を得ることができる．しかし，大規模なワークフローを想定した場合，計算量が膨大になり現実的な時間で解くことが難しい．そこで我々は，スケジューリングを現実的な時間で行うために，適応型スケジューリング手法[3] を提案している．これはワークフロー内のタスクの依存関係に着目し適応的にスケジューリング手法を切り替える方法で，ワークフロー内にパラメータスウィープのような互いに依存関係を持たないタスクの集合を多数含んでいた場合，スケジューリングの計算時間を大幅に削減することができる．しかし BLAST を用いた遺伝子解析のワークフローのように類似性の高いサブワークフローが複数含まれている場合，適応型スケジューリング手法のアルゴリズムでは処理の切り替えを効率的に行うことができず，スケジューリングの計算時間の大幅な削減が行えない可能性がある．

そこで我々は適応型スケジューリング手法を改良し，スケジューリング処理の適応的な切り替えを再帰的に行う手法を提案する．提案手法では，サブワークフローを一個の疑似タスクと見なして適応型スケジューリング手法でスケジューリングを行う．そして疑似タスクをスケジューリングする段階でサブワークフローに展開し，サブワークフロー内のタスクを適応型スケジューリング手法を用いてスケジューリングする．このように再帰的に適応型スケ

†1 三重大学大学院 工学研究科
Mie University

ジューリング手法を実行することで，一度の適応型スケジューリング手法の実行で扱うタスク数を減らし，またサブワークフローを疑似タスクに見なし効果的に処理の切り替えを発生させることで，計算時間の削減を行う．そして，このような大規模なワークフロー型の並列処理を目的とするタスク並列スクリプト言語 MegaScript[4)–6)] に本手法を実装し抽象シミュレーションで評価を行った結果，10,000 タスク規模でスケジューリングの計算時間を 1/100 程度に削減できた．

以下，2 章では背景である MegaScript とスケジューリングについて述べる．3 章では従来のスケジューリング手法を説明し，我々の目的との適合性について議論する．4 章では提案手法の詳細を述べ，5 章でシミュレーションによる提案手法の評価結果を示す．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

MegaScript は 2 階層並列モデルの上位層を記述する言語である．逐次や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行・並列に実行させることで並列処理を行う．また，ストリームと呼ばれる通信路を介することで，各タスク間のデータ通信を行う．計算の中心となる各タスクはネイティブプログラムとして用意するため，MegaScript プログラムでは，タスクやストリームから成るワークフローと呼ばれるワークフローやタスクの実行に必要な情報等を記述する．MegaScript ではこれらの情報を解析し，スケジューリング結果に従って各タスクを指定された計算機で実行する．

ここで未知の病原菌の DNA 配列群を既知の病原菌のデータと比較し特定を行い，またそれらの相互比較を行う場合のワークフローの例（**図 1**）を示す．まず未知の DNA 配列群に対し $BLAST$[7)] や $FASTA$[8)] を使い既知のシーケンスデータベースとの相同性検索を行い，その結果を利用して $ClustalW$[9)] でマルチプルアライメントを行い樹形図を得る (図 1 の太枠内)．これを複数のシーケンスデータベースで行い，シーケンスデータベースごとに得られた樹形図を $ClustalWPanel$[9)] に読み込ませ相互比較を行う．

2.2 タスクスケジューリング

一般のワークフローは Directed Acyclic Graph(DAG) で表現ができるが，MegaScript のようにタスクの動的生成が可能なシステムもある．しかし，本論文では議論を簡単にするために，ワークフローは予め与えられており動的に変化しないものとする．従って本論文で扱うスケジューリング手法は，DAG のオフラインスケジューリングの一種になる．一般の

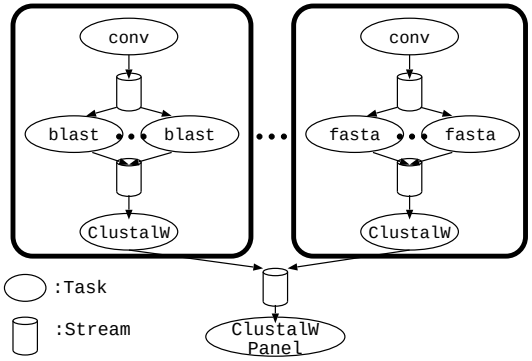

図 1　ワークフローの例
Fig. 1　Workflow Example

DAG の形を取るオフラインタスクスケジューリングとして，3 章で述べるように，様々な従来手法が提案されている．これらの手法は良いスケジューリング長を得ることができるが，大規模なタスク数やホスト数を想定した並列処理においては，計算コストが非常に高くなり現実的な時間で解くのは難しい．したがって，スケジューリング長を維持しつつ計算コストを大幅に削減することが必須である．

ワークフロー型の大規模並列処理を容易に実行でき，また効率よく自動で実行するようなシステムの需要は高い．しかし本論文で扱うタスク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) 一般のワークフローでは MegaScript のようにネイティブプログラムを扱うことも多く，タスクの分割やタスク実行中のマイグレーションは行えない．
(2) 実行環境として複数のホストを想定しており，計算性能は非均質である．
(3) MegaScript では各タスクの計算量や通信量をユーザーが記述するため，100% 正確とは限らない．
(4) 大規模な並列処理を目的としており，10 万 ~100 万あるいはそれ以上の規模のタスクを扱えることが求められる．
(5) 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，ユーザが明示的にワークフローを記述する．このためデータの分散・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み合わせになる．
(6) 科学技術の分野では，解析やシミュレーションを行うために，図 1 の太枠内のように

類似性の高いサブワークフローを多数含んだワークフローを用いることが多い．

ワークフロー型の大規模並列処理では DAG のタスクスケジューリングが必要になり，(2)，(4) の特徴により計算コストは非常に大きくなる．しかし，(5)，(6) の特徴により複雑な DAG になる可能性が低く，直並列グラフに近い単純なワークフローになることが多いと考えられる．従って DAG を前提とするスケジューリング手法を用いる必要はあるが，このような (5)，(6) の特徴を定式化し処理を簡略化することで，スケジューリングの計算コストの削減が期待できる．適応型スケジューリング手法では，互いに依存関係が無く先行・後続タスクが共通するタスクの集合に着目している．これはワークフロー中に多数存在すると考えられ，本論文ではこれらのタスクの集合を**独立型タスク群**と呼ぶ．

2.3 適応型スケジューリング手法

我々は 2.2 節であげた特徴を考慮した適応型スケジューリング手法を提案し，スケジューリング時間の大幅な削減を達成している [3]．本節では適応型スケジューリング手法について説明する．

適応型スケジューリング手法では，各独立型タスク群を疑似タスクに縮退し，従来の DAG のタスクスケジューリング手法を用いてスケジューリングを行う．2.2 節で述べたように，独立型タスク群は互いに依存関係は無い．したがって，この独立型タスク群をスケジューリングする際，DAG タスクスケジューリング手法ではなく，独立タスクスケジューリング手法を利用することができる．本論文の適応型スケジューリング手法では，DAG のタスクスケジューリングとして HEFT[1] を，独立タスクスケジューリング手法として簡略化した MinMin ヒューリスティック法を用いた．適応型スケジューリング手法の詳細は 3) を参照して頂きたい．

しかし，適応型スケジューリング手法はパラメータスウィープのような独立型タスク群しか効率的なスケジューリングが行えない．そのため，図 1 の太枠内のようなサブワークフローが多数存在している場合，従来の計算量の大きいタスクスケジューリング手法でスケジューリングが行われてしまい，計算時間が膨大になってしまう問題がある．

3. 関 連 研 究

本章では，既存のスケジューリング手法について概観し，提案手法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

DAG の静的スケジューリング手法は多くの研究がされており，その計算コストは高いが良いスケジューリング長を得ることができる．特にレベルスケジューリングに基づく手法は多く提案されており，古典的な手法では Mapping Heuristic(MH)[10] や Generalized Dynamic Level(GDL)[11]，Best Imaginary Level(BIL)[12] がある．また比較的引用回数の多い手法として，各タスクの実行時間の平均値を用いてレベルを定義する Heterogenous Earliest-Finish-Time(HEFT)[1] や Levelized Duplication Based Scheduling(LDBS)[2] がある．またメタヒューリスティックでレベルを最適化する手法[13),14)] 等もある．

しかし大規模ワークフロー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，スケジューリングにかかる計算コストが高くなってしまい，現実時間でスケジューリングを行うことが困難である．従って，2.3 節で紹介した適応型スケジューリング手法のように計算時間を削減することが必須となる．

3.2 独立タスクスケジューリング

スケジューリング対象のタスクを，実行完了時間が最小となるホストに順次配置していく手法に，Min-Min ヒューリスティック法[15] がある．これはタスクの実行時間が計算量や実行環境に比例しない *unrelated* な問題に対応した手法である．複数の独立タスクスケジューリング手法と比較した結果，スケジューリングにかかる計算コストが低く，比較的良いスケジューリング結果を導き出している[16]．

より計算コストが低いスケジューリング手法としては OLB や UDA があるが[17]，これらのスケジューリング長は Min-Min ヒューリスティックと比較すると良くない[16]．遺伝的アルゴリズムを使ったスケジューリング手法[18] では，Min-Min ヒューリスティックと比較すると良いスケジューリング結果を得られたが，計算コストが非常に高い結果になっている[16]．

ワークフロー型の大規模並列処理のタスクスケジューリングは，各タスク間の依存関係を考慮する必要があるが，図 1 の太枠内のサブワークフロー間は互いに依存関係を持たないので，上記の手法を適用させることができれば，より効率的なスケジューリングが行える．

4. 提 案 手 法

4.1 概　　要

3.1 節で紹介した従来の DAG のタスクスケジューリング手法はタスク間の依存関係を考慮しつつスケジューリングを行うため，良いスケジューリング長を得ることができる．しかし，我々が想定するような大規模なワークフローで従来の DAG のタスクスケジューリング手法を用いた場合，計算量が膨大となり現実時間で解くことが難しい．そこで我々はワーク

フロー内の独立型タスク群に着目した適応型スケジューリング手法を提案し，計算時間の大幅な削減を行うことができている．しかし，適応型スケジューリング手法は独立型タスク群を含んでいる箇所しか効果が得られない．従って，図 1 のようなサブワークフローが多数含まれているワークフローでは，*conv* や *ClustalW* のようなタスクを一度にスケジューリングしてしまうため，計算量の大幅な削減を見込めないという問題がある．

そこで提案手法では初めに，ワークフロー中のサブワークフローを疑似タスクに置き換える．そして置き換えたワークフローに対して適応型スケジューリング手法でスケジューリングを行う．その際に，疑似タスクをホストへスケジューリングする時点でサブワークフローへ展開し，サブワークフロー内の各タスクを適応型スケジューリング手法でスケジューリングする．このように提案手法では，適応型スケジューリング手法を再帰的に実行することでスケジューリングを行う．また，図 1 に示すような単純なサブワークフローだけでなく，より複雑な任意のサブワークフローに対しても提案手法を用いることが可能である．

4.2 サブワークフローの検出

MegaScript ではサブワークフローを効率よく記述するために TaskNet クラス[19] が用意されている．従ってスケジューラは MegaScript 処理系から TaskNet インスタンスの情報を取得することでワークフロー内のサブワークフローを検出することができる．また実ワークフローでは図 1 の遺伝子解析ワークフローのように，入力データを変えてサブワークフローを実行することが多い．従って，このサブワークフローを一個の疑似タスクと見なすとパラメータスウィープのような形となる．そこで MegaScript ではこのような形を容易に記述できるようにするために，サブワークフローを一個の疑似タスクとし，それらを配列として記述できるようにしている[19]．

4.3 アルゴリズム

図 2–4 は提案手法のアルゴリズムである．提案手法の最上位関数を**図 2** に示す．提案手法ではワークフロー内のサブワークフローや独立型タスク群を全て疑似タスクへと置換する．MegaScript では 4.2 節で説明したように TaskNet クラスが用意されており，ユーザーがワークフロー内のサブワークフローを明示的に記述することができる．また MegaScript 処理系の内部でも TaskNet を用いたワークフローの管理がされている．*SubWorkflowToPseudoTask()* 関数ではこれを利用することで，ワークフロー内のサブワークフローを疑似タスクへと置き換える．また，ワークフロー内の独立型タスク群を *TaskArrayToPseudoTask()* 関数を用いることで疑似タスクへと置き換える．MegaScript では TaskNet も 4.2 節で説明したように配列として扱えるため，*TaskArrayToPseudoTask()* 関数ではサブワークフロー

```
1.function Schedule(workflow)
2.   SubWorkflowToPseudoTask()
3.   TaskArrayToPseudoTask()
4.   DagSchedule(workflow)
5.end
```

図 2 提案手法のアルゴリズム
Fig. 2 Algorithm of Proposed Scheduling Scheme

の配列も一個の疑似タスクへと置き換える．最後に置き換えた後のワークフローを引数に *DagSchedule()* 関数の呼び出しを行うことで，全タスクのスケジューリングを行う．

DagSchedule(workflow) 関数は依存関係を含んだ DAG のスケジューリング関数であり，ワークフロー *workflow* のスケジューリングを行う．本論文では適応型スケジューリング手法と同様に HEFT[1] を使っている．**図 3** は *DagSchedule()* 関数のアルゴリズムである．*DagSchedule()* 関数は最初に *workflow* に含まれる各タスクのランクを計算している．これは各タスクの平均計算時間をベースに，ワークフローの先頭に近いタスクほど高い値になるように計算されている．算出方法の詳細については HEFT[1] を参照して頂きたい．そして，このランクの値が高いタスクから順番にスケジューリングを行う (5–18 行目)．6 行目で未スケジューリングのタスクの中でランクの高いタスク T_i を取得する．この時，タスク T_i の種類によって処理が異なる．タスク T_i が疑似タスクでなく通常のタスクの場合（14 行目），*Mapping(task)* 関数によって *task* の実行完了時刻が一番早いホストへと割り当てられる．タスク T_i がサブワークフローを置換した疑似タスクの場合，サブワークフローを引数に *DagSchedule()* 関数を再帰的に呼び出すことで，各タスクに対して DAG のタスクスケジューリングを行う（11–12 行目）．タスク T_i が独立型タスク群を置換した疑似タスクの場合，適応型スケジューリング手法と同様に，この疑似タスクの各タスクに対して独立タスクスケジューリングを用いてスケジューリングを行う（8–9 行目）．*IndepSchedule(tasks)* 関数は独立タスクスケジューリングを *tasks* に対して行う関数である．

図 4 は *IndepSchedule(tasks)* 関数のアルゴリズムである．*IndepSchedule(tasks)* 関数では，*tasks* の各要素がサブワークフローを置換した疑似タスクであるか否かによって処理が異なる．最初に各要素がサブワークフローを置換した疑似タスクであるか否かを判定し，各要素が疑似タスクの場合 *isSubWorkflow* に true を，通常のタスクである場合 false を設定している（2–6 行目）．その後，適応型スケジューリング手法で用いている簡略化した MinMin ヒューリスティック法と同様に，計算量の小さいタスクから順にスケジューリングを行って

```
 1.function DagSchedule(workflow)
 2.  foreach(タスク T_i in workflow)
 3.    タスク T_i のランクを算出
 4.  各タスクのランクに従って降順にソートし list に格納
 5.  while (list.length > 0) do
 6.    タスク T_i = list.first
 7.    if タスク T_i が独立型タスク群 I_j を置換した疑似タスク then
 8.      tasks = 独立型タスク群 I_j に含まれる全てのタスク
 9.      IndepSchedule(tasks)
10.    else if タスク T_i がサブワークフロー N_k を置換した疑似タスク then
11.      subworkflow = サブワークフロー N_k
12.      DagSchedule(subworkflow)
13.    else
14.      Mapping(タスク T_i)
15.    end
16.    list = list - タスク T_i
17.  end
18.end
```

図 3 *DagSchedule()* 関数のアルゴリズム
Fig. 3 Algorithm of *DagSchedule()* function

```
 1.function IndepSchedule(tasks)
 2.  if tasks がサブワークフローを置換した疑似タスクの独立型タスク群 then
 3.    isSubWorkflow = true
 4.  else
 5.    isSubWorkflow = false
 6.  end
 7.  tasks の各タスクの計算コストに従って昇順にソートし list に格納
 8.  while (list.length > 0) do
 9.    タスク T_i = list.first
10.    if isSubWorkflow = true then
11.      workflow = 疑似タスク T_i を展開したサブワークフロー
12.      DagSchedule(subworkflow)
13.    else
14.      Mapping(タスク T_i)
15.    end
16.    list = list - タスク T_i
17.  end
18.end
```

図 4 *IndepSchedule()* 関数のアルゴリズム
Fig. 4 Algorithm of *IndepSchedule()* function

いく（7–17 行目）．タスク T_i が通常のタスクの場合，*Mapping(task)* 関数を用いてホストへ割り当てを行う（14 行目）．そして，タスク T_i がサブワークフローを置換した疑似タスクの場合，サブワークフローを引数に *DagSchedule()* 関数を再帰的に呼び出すことで，各タスクに対して DAG のタスクスケジューリングを行う（11–12 行目）．

このように *DagSchedule()* 関数や *IndepSchedule()* 関数を再帰的に呼び出す．*DagSchedule()* 関数の計算オーダーは扱うタスク数とホスト数をそれぞれ t と m とすると $O(t^2m)$ で，その計算量は扱うタスク数の二乗に比例する．従って，サブワークフローや独立型タスク群を疑似タスクで表すことで，*DagSchedule()* 関数の計算量が大幅に削減される．また，サブワークフローの配列は個々のサブワークフローを疑似タスクとして扱い *IndepSchedule()* 関数を用いてスケジューリングを行う．*IndepSchedule()* 関数の計算オーダーは配列のサイズを n とした場合 $O(nm)$ になり，サブワークフローの配列全体を *DagSchedule()* 関数を用いた場合と比較して計算量の大幅な削減を行うことができる．以上の理由から，提案手法の計算量は適応型スケジューリング手法と比較して非常に小さくなる．

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を行わず，以下の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．各タスクの計算時間はそのタスクの計算量に従い，通信時間はそのタスクの通信量に従う．

- 各タスクの通信は，そのタスクの実行終了時に行う．
- 各ホストでは一度に一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全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．

本評価では，CPU に Intel®Xeon®Processor X3330 (2.66GHz，L2 cache 6MB)，メモリに 2GB を搭載した環境でスケジューリングを行った．

提案手法の効果を確認するために，スケジューリング時間とスケジューリング性能の二つの評価を行った．ここでスケジューリング時間とは，スケジューリング処理の完了までにかかる実行時間を示す．スケジューリング性能とは，適応型スケジューリング手法のスケジューリング長を 1 として正規化した場合の，提案手法のスケジューリング長の増加率を示す．

5.2 シミュレーション条件

本評価では以下の手順により，ランダムなワークフローを生成した．

(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ワークフローを生成する．

(2) 以下のいずれかの操作を等確率で適用する．

連結 ランダムにタスクまたは独立型タスク群 T_i を選択する．ただし，ワークフローの先頭のタスクは除外する．新しくタスク T_j とストリーム 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．

コントロール並列 ランダムにタスクまたは独立型タスク群 T_i を選択する．ただし，ワークフローの先頭と末尾のタスクは除外する．新しくタスク 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．

データ並列 ランダムにタスク T_i を選択する．ただし，ワークフローの先頭と末尾のタスクは除外する．新しく独立型タスク群 T_j を生成し T_i と置き換える．この時，独立型タスク群の大きさは各評価の条件に従う．

サブワークフロー化 ランダムに独立型タスク群 T_i を選択する．独立型タスク群 T_i をサブワークフローの配列に置換する．新しくタスク 2 個をストリームで接続したワークフローを生成し，これをサブワークフローの構造として保存する．

(3) タスクの総数が設定値以下であれば，(2) から繰り返す．

提案手法ではサブワークフロー内にサブワークフローを含むような，サブワークフローのネストしたワークフローに対してもスケジューリングが可能である．しかし，本実験ではサブワークフローのネストが無いワークフローを用いた．

各タスクの計算量や通信量について，**表 1** の範囲でランダムに選んだ値を利用した．タスクの通信量については各評価で用いる CCR 値 (Communication to Computation Ratio) を使い，

表 1 ワークフローの生成条件
Table 1 Workflow Attributes

タスクの計算量	100 - 200
タスクの基本通信量	100 - 200

表 2 非均質環境の生成条件
Table 2 Attributes of Heterogeneous Environments

ホスト間通信性能	100
ホストの計算性能	1.0 - 5.0
ホストの台数	32

$$\text{タスクの通信量} = \text{タスクの基本通信量} * \text{CCR 値} \quad (1)$$

とする．これにより CCR 値が大きい場合，全体的にタスクの計算量より通信量が大きいワークフローとなり，CCR 値が小さい場合はその逆となる．

実行環境については**表 2** の条件に従ってランダムに生成した．BLAST[7] や AMBER[20] の実行時間が通常数百秒以内であり，その出力結果は数百 MB 程度であることが多い．一般的に各クラスタ内については，ギガビット・イーサネット等の高速回線で接続しており，数百 MB のデータ転送時間は数秒程度である．そこで上で設定した計算・通信量から，計算・通信の時間比が BLAST や AMBER の場合に近くなるように設定した．

それぞれの評価で示すデータは，同じ条件でランダムに生成した 40 種類のデータセットに対しそれぞれシミュレーションを行った平均値である．今後，より広い範囲におけるデータセットで評価を行い，様々な実アプリケーションや実行環境における効果を確認していく必要がある．

5.3 評 価 結 果

提案手法のスケーラビリティによる効果を明らかにするために，スケジューリング性能とスケジューリング時間による比較評価を行った．**表 3** は，100，1,000，10,000 タスクからなるワークフローをスケジューリングした結果である．表中の適応型・提案手法はそれぞれの手法におけるスケジューリング時間を，増加率はスケジューリング性能を示したものである．また，CCR 値は 1.0 とした．

表 3 の結果から，タスク数にかかわらず全ての結果で提案手法のほうがスケジューリング時間が短い結果となった．またタスク数が増加するほど，スケジューリング時間の削減率は大きくなっている．これは 4 章で説明したように，提案手法ではサブワークフローを一個の疑似タスクとして扱うことで各関数内で扱うタスク数を減らすことができたためだと

表 3 スケーラビリティの評価
Table 3 Performance with the number of tasks

タスク数	100	1,000	10,000
適応型 (秒)	0.049	2.481	211.440
提案手法 (秒)	0.011	0.114	1.782
増加率	1.371	1.255	1.117

表 4 CCR 値の評価
Table 4 Performance with CCR

CCR 値	0.01	0.1	1	10
適応型 (秒)	249.580	269.161	211.440	214.514
提案手法 (秒)	1.834	1.612	1.782	1.765
増加率	1.063	1.099	1.085	1.117

考えられる．またスケジューリングの増加率に関しても，タスク数が 100 の時は 30%程度増加したが，タスク数が増加するほど増加率を抑えることができ，タスク数が 10,000 の時は 10%程度の結果となった．これはタスク数が小さいほど，個々のタスクのスケジューリング結果によってスケジューリング長が大きく影響を受けるためであると考えられる．以上から，ワークフローのタスク数が多いほど提案手法は効果があると言える．

また，提案手法の CCR 値による効果を明らかにするために，スケジューリング性能とスケジューリング時間による比較評価を行った．**表 4** は，10,000 タスクからなるワークフローをスケジューリングした結果である．表中の適応型・提案手法はそれぞれの手法におけるスケジューリング時間を，増加率はスケジューリング性能を示したものである．

表 4 の結果から，CCR 値にかかわらず提案手法のスケジューリング時間は 1/150 程度になった．またスケジューリング性能に関しても，スケジューリング長の増加率を 10%程度に抑えることができる結果となった．以上から，個々のタスクの計算時間が支配的になるワークフローとタスク間のデータ通信時間が支配的になるワークフローの両方において，提案手法は効果があると言える．

6. おわりに

本論文ではサブワークフローを多く含むワークフローでも高速なスケジューリングが可能な手法を提案した．これはサブワークフローや独立型タスク群を一個の疑似タスクに置換することで計算規模を小さくし，スケジューリング時間を大幅に削減する手法である．提案手法を抽象シミュレーションによる評価を行った結果，10,000 タスクのワークフローではスケジューリング長の増加率を 10%程度に抑えつつ，スケジューリング時間を 1/100 程度に削減することができた．また，ワークフロー内のタスク数が多いほど，スケジューリング長の増加率を抑えつつスケジューリング時間を大幅に削減することができた．

今後はスケジューリング時間を維持しつつ，よりスケジューリング長の増加を抑える手法を考える必要がある．また，マルチクラスタのような広域分散環境への対応や，オンラインスケジューリングとの併用などについても研究して行く必要がある．さらに，より現実的なモデルを用いたシミュレーション評価や，実際の環境を用いた実評価を行っていく必要がある．

参 考 文 献

1) H.Topcuoglu, S.Hariri and Wu, M.-Y.: A high-performance mapping algorithm for heterogeneous computing system, *IPPS/SPDP Workshop on Heterogeneous Computing*, pp.3–14 (1999).
2) A.Dogan and R.Ozguner: LDBS:a duplication based scheduling algorithm for heterogeneous computing systems, *Proc. Int'l Conf. Par. Proc.*, pp.352–359 (2002).
3) 松本真樹，片野聡，佐々木敬泰，大野和彦，近藤利夫，中島浩：非均質環境における適応型スケジューリング手法の提案と評価，電子情報通信学会論文誌，Vol.J93-D, No.6, pp.693–704 (2010).
4) 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.SIG 12 (ACS 11), pp.181–193 (2005).
5) 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).
6) 阪口祐輔，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ーレベル機能拡張機構，情報処理学会論文誌， Vol.47, No.SIG 12(ACS 15), pp.296–307 (2006).
7) Altschul, S., Gish, W., Miller, W., Myers, E. and Lipman, D.: Basic local alignment search tool, *Journal of Molecular Biology*, Vol.215, pp.403–410 (1990).
8) D.J.Lipman and W.R.Pearson: Rapid and sensitive protein similarity searches., *Science*, Vol.227, pp.1435–1441 (1985).
9) J.D.Thompson, D.G.Higgins and T.J.Gibson: CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice, *Nucleic Acids Reseach*, Vol.22, pp.4673–4680 (1994).

10) El-Rewini, H. and Lewis, T.: Scheduling parallel program tasks onto arbitrary target machines, *Journal of Parallel and Distributed Computing*, Vol.9, No.2, pp. 138–153 (1990).
11) Sih., G.C. and Lee, E.A.: Dynamic-level scheduling for heterogeneous processor networks, *Proceedings of IEEE Symposium on Parallel and Distributed Processing* (1990).
12) Oh, H. and Ha, S.: A static scheduling heuristic for heterogeneous processors, *Euro-Par'96 Parallel Processing* (1996).
13) Boeres, C., Rios, E. and Ochi, L.S.: Hybrid evolutionary static scheduling for heterogeneous systems, *The 2005 IEEE Congress on Evolutionary Computation* (2005).
14) Kim, G.H. and Lee, C. S.G.: Genetic reinforcement learning for scheduling heterogeneous machines, *1996 IEEE International Conference on Robotics and Automation* (1996).
15) Wu, M.-Y., Shu, W. and Zhang, H.: Segmented min-min: a static mapping algorithm for meta-tasks onheterogeneous computing systems, *Heterogeneous Computing Workshop, 2000*, pp.375–385 (2000).
16) et. al., T.B.: A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems., *IPPS/SPDP Workshop on Heterogeneous Computing*, pp.15–29 (1999).
17) R.Armstrong, D.Hensgen and T.Kidd: The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions., *7th IEEE Heterogenous Computing Workshop*, pp.87–97 (1998).
18) M.Srinivas and L.M.Patnaik.: Mapping and scheduling heterogeneous task graphs using genetic algorithms., *5th IEEE Heterogenous Computing Workshop*, pp.86–97 (1996).
19) Ohno, K., Mita, A., Matsumoto, M., Sasaki, T., Kondo, T. and Nakashima, H.: Efficient Implementation of Large-scale Workflows based on Array Contraction., *Proceedings of the 22th IASTED International Conference on Parallel and Distributed Computing and Systems*, pp.153–162 (2010).
20) http://ambermd.org/: The Amber Molecular Dynamics Package.