情報処理学会論文誌　プログラミング　Vol. 4　No. 3　55–67 (June 2011)

静的情報を用いた動的再スケジューリングのオーバヘッド削減手法

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

ワークフロー型の大規模並列処理において実行効率を高めるにはタスクスケジューリングが重要であり，様々な手法が提案されている．静的スケジューリング手法は良いスケジューリング長を得ることができるが，計算ホストの負荷等による性能変動が発生した場合，ワークフローの実行効率が低下する可能性がある．一方デマンドドリブン型の動的スケジューリング手法は，ワークフローの依存関係によって良いスケジューリング長を得られない場合がある．そこで，実行環境の性能変動が発生するごとに静的スケジューリング・アルゴリズムを用いて再スケジューリングを行う動的再スケジューリング手法が提案されている．しかし，静的スケジューリング・アルゴリズムの計算コストは非常に高いため，アルゴリズムの実行回数が多い場合，現実的な時間でスケジューリングを行えない．そこで本発表では，スケジューリングの計算コストを削減することを目的とした動的再スケジューリング手法を提案する．従来の手法は性能変動を条件に再スケジューリングを行っているため，無駄な再スケジューリング処理が行われる可能性がある．そこで本手法は，再スケジューリングの実行回数を削減するために，特定タスクの終了を実行条件として用いている．本手法を抽象シミュレーションを用いて評価した結果，タスク数が 1,000 規模のワークフローの場合，5% 程度のワークフローのスケジューリング長の増加で，再スケジューリングの実行回数を約 1/4〜1/100 に減らすことができた．

A Dynamic Rescheduling Scheme for Reducing Scheduling Overhead Using Static Information

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

Task scheduling is very important for efficient execution of large-scale workflows in distributed computing environments. Static scheduling schemes achieve high performance when executing workflows in stable environments. However, the scheduling costs are very high for large-scale workflows and they may perform poorly if the system performance is changed dynamically. Demand-driven scheduling schemes achieve high performance for independent tasks, but they may perform poorly for workflows. Dynamic rescheduling schemes have the advantages of these two types of schemes, because tasks are rescheduled using algorithms from static scheduling schemes when the performance is changed. Thus, better performance can be achieved in workflow applications, although the scheduling costs are increased if the system performance is frequently changed. Therefore, we propose a new dynamic scheduling scheme to reduce the number of reschedulings. The rescheduling trigger of this scheme is based on task terminations. Moreover, the scheme reduces the number of reschedulings compared to a scheme using a simple task termination trigger by checking dependencies between tasks. Evaluation using an abstract simulation demonstrated that the number of reschedulings was reduced to approximately 1/4 to 1/100 of that without, and with a 5% increase of the execution time.

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実行する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，階層型スケジューリング手法[3] や，適応型スケジューリング手法[4] を提案している．しかし，静的スケジューリング手法はワークフローを実行する前に 1 度だけスケジューリングを行うため，他のプロセス等による実行時のホストの計算

†1 三重大学大学院工学研究科
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

性能の変化については対応できない．性能の変化に対応している手法としては，デマンドドリブン型のスケジューリング手法が提案されているが，これはワークフローの依存関係を考慮していないため，静的スケジューリング手法と比較して実行効率が低下する恐れがある．一方，静的スケジューリング手法やデマンドドリブン型のスケジューリング手法に対して，AHEFT [5] や Blythe らの手法[6] のような動的再スケジューリング手法が提案されている．これらの手法は，実行ホストの性能変動が発生した場合，実行ホストの新しい計算性能を用いて再スケジューリングを行う．また再スケジューリングを行う際，ワークフローを考慮する静的スケジューリング手法のアルゴリズムをスケジューリング関数として用いる．したがって，動的再スケジューリング手法はワークフローの依存関係を考慮しつつ実行時のホストの性能の変化について対応できる．しかし，動的再スケジューリング手法の計算コストは静的スケジューリング手法の計算コストより高くなるという問題がある．

そこで我々は，動的再スケジューリング手法の計算コストを削減する手法を提案する．従来の手法は性能変動を条件に再スケジューリングを行っているため，短い時間に連続してホストの計算性能が変化するような環境では，無駄な再スケジューリング処理が行われる可能性がある．そこで本手法では，このような再スケジューリング処理を行わないようにするために，特定の条件を満たすタスクの終了を再スケジューリング処理の実行条件として用いている．そして本手法の効果を確認するために，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 [7)–9)] に実装し，抽象シミュレーションで評価を行った．その結果，タスク数が 1,000 規模のワークフローの場合で，5% 程度のワークフローのスケジューリング長の増加で，再スケジューリングの実行回数が約 1/4〜1/100 に減った．

以下，2 章では背景であるタスクスケジューリングと今回実装に用いた MegaScript の概要について述べる．3 章では従来のスケジューリング手法を紹介し，我々の目的との適合性について議論する．4 章では提案手法を述べ，5 章で抽象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．独立した外部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なるタスクネットワーク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．

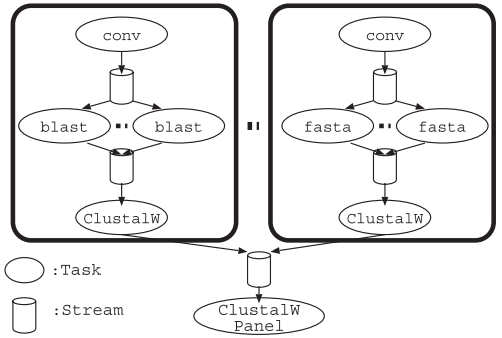


図 1　タスクネットワークの例
Fig. 1　Example of tasknetwork.

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，またその結果の相互比較を行うタスクネットワークの例（**図 1**）を示す．まず未知の DNA 配列群に対し BLAST [10)] や FASTA [11)] 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，その結果に対して ClustalW [12)] 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）．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 ClustalW Panel [12)] 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，本稿では議論を簡単にするために，ワークフローはあらかじめ与えられており動的に変化しないものとする．したがって本稿で扱うスケジューリング手法は，DAG のタスクスケジューリングの一種になる．一般の DAG の形をとる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．大規模ワークフローを対象とした場合これらの手法は一長一短であるが，実用化においてはそれぞれの利点を両立させ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，また効率良く自動で実行するような

システムの需要は高い．しかし，そのようなシステムを実現するためにはいくつかの特徴を考慮する必要がある．以下は本稿で扱うタスクスケジューリングの特徴である．

(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)　計算資源を独占して利用できるとは限らないので，ワークフローの実行中に他のプロセスによってホストの計算性能が変わる可能性がある．
(4)　MegaScript では各タスクの計算量や通信量をユーザが記述するため，100% 正確とは限らない．
(5)　大規模な並列処理を目的としており，10 万 ∼100 万規模のタスクやホストを扱えることが求められる．
(6)　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合せになる．
(7)　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 *blast* や *fasta* のように入力データや実行時引数を変えて同一プログラムを実行するパラメータスウィープを用いることが多く，これら実行は互いに依存関係がない．MegaScript のワークフローではこのようなパラメータスウィープを含んでいることが多い．

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)，(5) の特徴により計算コストは非常に大きくなる．しかし，(6)，(7) の特徴により複雑な DAG の記述が難しいため，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．また，(3) や (4) の特徴から厳密なスケジューリングは求められていない．しかし (3) を考慮した DAG のタスクスケジューラは厳密なスケジューリングを行うことに着目しているものが多くそれらの計算コストは高い．そのため，(2)，(5) の特徴を考慮した場合，スケジューリングの計算時間は非現実的になり，実システムで用いることは難しい．したがって，スケジューリングの計算コストが小さく，動的な性能の変化に対してスケジューリング結果を補正するようなスケジューリング手法が必要になる．

3. 関 連 研 究

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）[13] や Generalized Dynamic Level（GDL）[14]，Best Imaginary Level（BIL）[15] 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[16),17] 等もある．レベルスケジューリング以外の手法としては，Modified Critical Path（MCP）[18] や Dynamic Critical Path（DCP）[19] のようなワークフロー中のクリティカルパスに注目した手法も多く提案されている．

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，MegaScript が想定するような実行環境ではホストの計算性能やホスト間の通信性能が頻繁に変化する．このため，静的スケジューリング手法で良いスケジューリング長を得ても実際の実行時間と一致するとは限らず，高い実行効率を得ることが難しい．また大規模ワークフローを想定した場合，静的スケジューリング手法は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．

3.2 要求駆動型のスケジューリング手法

タスク間に依存関係がないものを対象とした手法に，マスタワーカ型の単純なスケジューリング手法がある．これらは各ワーカホストがアイドル状態となるたびにマスタホストへタスクを要求するため，実行環境における計算性能や通信性能の変化に柔軟に対応できる．最も単純な手法としては，アイドル状態になったマシンにタスクを割り当てる Opportunistic Load Balancing（OLB）[20] や，タスクの実行時間が短いホストへ割り当てる User Directed Assignment（UDA）[20] がある．また，各タスクの計算量やホストの計算性能等を考慮する手法としては，Min-min Heuristic [21] や Greedy 法[22] 等がある．これらに対しランダムスチール（RS）法[23] 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．

しかしワークフローを扱う場合，これらの手法は大域的な依存関係を考慮できないため，静的スケジューリング手法と比較して実行効率が低下する恐れがある．

3.3 DAG の動的スケジューリング手法

ワークフローを考慮した手法として，AHEFT [5] や DCP-G [24]，Blythe らの手法[6] 等のような動的再スケジューリング手法が提案されている．これらは性能変動が発生するような

実行環境においても，静的スケジューリング手法のような良いスケジューリング長を得ることができる．これらの手法は初めに，静的スケジューリング手法のような最適解に近いスケジューリング結果を求める．その後，そのスケジューリング結果に従って各タスクを実行していく．そして，ホストの計算性能や通信性能に大きな変化が発生した場合，未実行のタスクに対して性能変化を考慮したうえで再度スケジューリングを行う．そして残りの未実行タスクはこの再スケジューリングの結果に従って実行される．これにより，ワークフローの依存関係を考慮しつつ，性能の変化に対応してワークフローを実行することができる．しかしこのような再スケジューリングを行う手法は，デマンドドリブン型の単純な動的スケジューリングと比較してスケジューリングにかかる計算コストが大きくなり，これがワークフローの実行において大きなオーバヘッドとなる恐れがある．

一方で，同一ワークフローを複数回実行することを想定した手法[25)–27)]等も提案されており，少ない計算コストで高い性能を得ることができる．これらは，前回までのスケジューリング結果を参考にスケジューリングを行う．したがって，ワークフローの実行回数が増えるほど，より最適なスケジューリング結果が得られる．しかし，文献 25) は均質なホストを想定しており，文献 26)，27) は非均質なホストに対応しているが約 50 タスクまでのワークフローしかスケジューリングを行うことができない．

我々のスケジューラは DAG を扱うことができ，性能変動に対応できる必要がある．したがって，本手法は動的再スケジューリング手法をベースとし，再スケジューリングの実行回数を抑えることでスケジューリングの計算コストを削減する手法を提案する．また，1 回あたりの再スケジューリング処理の計算コストも削減するために，適応型スケジューリング手法[4)]を再スケジューリング処理に用いた．

4. 提案手法

4.1 概　　要

3.1 節で紹介した静的スケジューリング手法は，ホストの性能変動が起きないような安定した環境では最適解に近いスケジューリング結果を得ることができる．しかし，大規模ワークフローを想定した場合，その計算コストは非常に高い．また，ワークフローの大規模化にともないその実行時間も長くなるため，長時間にわたってホストの性能変動が起きないことを望むことは難しい．したがって，大規模ワークフローではホストの性能変動が発生することでスケジューリング長が悪化する恐れがある．

一方，3.2 節で紹介した要求駆動型の単純なスケジューリング手法はホストの性能変動に対応しており，スケジューリングの計算コストも低い．これらの手法はパラメータスウィープのような依存関係のないタスク群に対しては高い性能が得られるが，ワークフローに対しては，タスク間の依存関係によって静的スケジューリング手法ほど良いスケジューリング結果を得られない可能性がある．

ホストの性能変動に対応しており，かつワークフローの依存関係を考慮したスケジューリングを行う手法として，3.3 節で紹介した動的再スケジューリング手法がある．しかし，スケジューリングの計算コストは静的スケジューリング手法より高い．たとえば，HEFT[1)] の計算オーダは，$O(t^2m)$（ワークフロー中のタスク数を t，実行ホスト数を m とした場合）になるのに対し，性能変動が発生するごとに HEFT のスケジューリングを呼び出す AHEFT[5)] の計算オーダは，$O(t^2mp)$（性能変動の発生回数を p とした場合）となる．大規模ワークフローを想定した場合，多くのホストが必要になることや長時間ホストを使用することから性能変動の発生回数は増大してしまい，スケジューリングの計算コストは大きくなってしまう．この問題に対して，従来の動的再スケジューリング手法はスケジューリングアルゴリズムの改良に焦点を当てており，再スケジューリング処理の実行回数について考慮されていない．そこで我々は，再スケジューリング処理の実行回数を大幅に減らすことで，小さい計算コストで良いスケジューリング長を得る手法を提案する．

4.2 実行モデル

本節では提案手法の実行モデルについて述べる．**図 2** は 4 台のホストからなる環境の実行モデルである．本手法の実行モデルでは，各計算ホストでランタイムが起動しており（図 2 の `Runtime`），またランタイムとは別に 1 個のスケジューラが起動している（図 2 の `Scheduler`）．そして，スケジューラは内部に保持しているワークフローの情報 (1) と実行環境の情報 (2) からスケジューリングを行う．各タスクの計算量や通信量，タスク間の依存関係といった情報は，ワークフローの情報として扱う．各計算ホストの計算性能や互いの通信性能については，実行環境の情報として扱う．

またランタイムとスケジューラは多対 1 通信を行っており（図 2 の (a)），これによってタスクの実行やホストの性能変動に関する情報を通信している．たとえば，**図 3** の (a) ホストでタスク T_k を実行する場合，最初にランタイムはスケジューラからタスク T_k の実行開始命令を受け取る（図 3 の (1)）．その後，ただちにホスト (a) はタスク T_k の実行プロセスを起動させる（図 3 の (2)）．またランタイムは計算ホストの性能変動を検知することができ，その情報をスケジューラに通知する．たとえば，図 3 の (b) のホストの性能が変化した場合，図 3 の (3) のようにホストのランタイムからスケジューラに通知する．

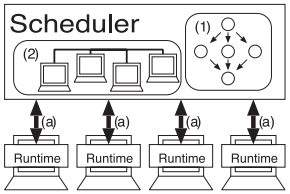


図 2　実行モデルの例
Fig. 2　Example of environment model.

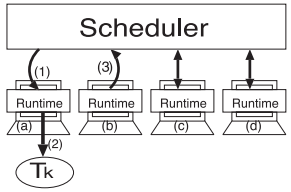


図 3　タスクの起動通知や性能変動通知
Fig. 3　Task execution and performance changes.

4.3 定　　義

本節では提案手法に必要な各種定義を行う．ワークフローに含まれるタスクを $T_1, T_2, \cdots, T_n$ とする．また，実行環境として用いるホストを $H_1, H_2, \cdots, H_m$ とする．*IsPerformanceChanged(message)* 関数は，ランタイムから受信したメッセージ *message* がホストの計算性能の変化を通知するものであった場合真を返す関数である．*GetChangedPerformance(message)* 関数は，ホストの計算性能の変化を通知するメッセージ *message* を引数とし，ホストの変化した後の計算性能を返す関数である．*IsTaskFinished(message)* 関数は，ランタイムから受信したメッセージ *message* がタスクの実行完了を通知するものであった場合真を返す関数である．*GetTaskFinished(message)* 関数は，タスクの実行完了を通知するメッセージ *message* を引数とし，実行が完了したタスクを返す．

また，我々が想定するワークフローでは 2.2 節 (7) で示したようなタスクの集合が多く含まれており，本手法ではこのタスクの集合に着目しスケジューリングのオーバヘッドの削減を行っている．そこで，以下のすべてを満たすタスクの集合を**独立型タスク群**と定義する．

(1)　独立型タスク群を構成するタスクは互いに依存関係を持たない．
(2)　独立型タスク群を構成するタスクに対して，直接依存するタスクの集合はすべて同じである．
(3)　独立型タスク群を構成するタスクに対して，直接依存されるタスクの集合はすべて同じである．

図 4 は独立型タスク群の例である．これは物理シミュレーションを行い，その結果を視覚化するワークフローである．このワークフローには 3 個の独立型タスク群（(i)，(ii)，(iii)）が含まれている．(i) と (ii) のタスクは，(1) と (3) の条件を満たしているが，(i) に属するタスクと (ii) に属するタスクでは依存するタスクが異なるため，(2) の条件を満たさない．したがって (i) と (ii) は異なる独立型タスク群となる．

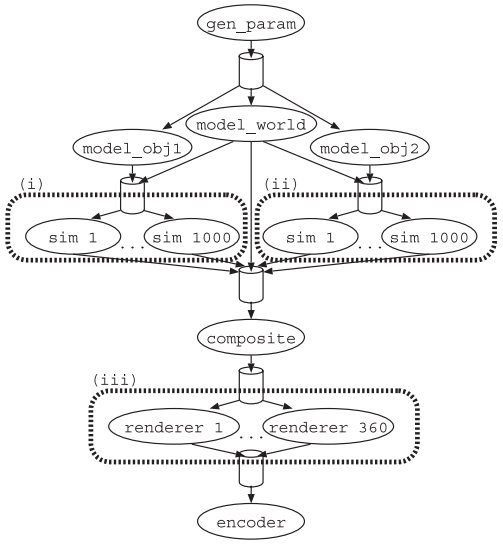


図 4　独立型タスク群の例
Fig. 4　Example of independent task sets.

我々が想定するワークフローには複数の独立型タスク群が含まれており，それぞれを $I_1, I_2, \cdots, I_l$ とする．また，*GetContainedITS(task)* 関数は，タスク *task* が含まれる独立型タスク群 I_i を返す関数で，タスク *task* がどの独立型タスク群にも含まれない場合，無効値 `nil` を返す．*IsAllTaskFinished*(I_i) 関数は独立型タスク群 I_i を引数とし，独立型タスク群 I_i に含まれるすべてのタスクの実行が終了している場合に真を返す．

4.4 動的再スケジューリング・アルゴリズム

図 5 に動的再スケジューリングのアルゴリズムを示す．初めに *Schedule()* 手続きを呼び出すことで，各タスクの初期のスケジューリングを行う（1 行目）．*Schedule()* 手続きには HEFT [1] や適応型スケジューリング手法[4] のような静的スケジューリング手法のスケジューリング関数を用いることができる．本手法ではスケジューリングの計算コストが軽い適応型スケジューリング手法を用いた．適応型スケジューリング手法の詳細は文献 4) を参照していただきたい．

続く 2 行目から 12 行目では，すべてのタスクの実行が完了するまで以下の処理を繰り返す．

```
 1.Schedule()
 2.while (DAG is not finished) do
 3.   任意のホストからメッセージを受信するまで待機
 4.   message = 受信したメッセージ
 5.   H_i = 受信元ランタイムのホスト
 6.   if IsPerformanceChanged(message) = true then
 7.     ホスト H_i の計算性能 = GetChangedPerformance(message)
 8.   end
 9.   if Trigger() = true then
10.     Schedule()
11.   end
12.end
```

図 5　動的再スケジューリングのアルゴリズム
Fig. 5　Algorithm of dynamic rescheduling scheme.

(1)　ランタイムからメッセージの受信を待つ（3–5 行目）．
(2)　受信したメッセージに従ってスケジューラが保持するデータを更新する（6–8 行目）．
(3)　再スケジューリングを行うかチェックし，必要に応じて再スケジューリングを行う（9–11 行目）．

3 行目では各ホストで動いているランタイムのいずれかからメッセージを受信するまでスケジューラは待機する．その後，いずれかのランタイムからメッセージを受信した場合，受信したメッセージを *message* に，メッセージを送信したランタイムが動いているホストを H_i に格納する（4–5 行目）．6–8 行目では，*message* がホスト H_i の性能変動を通知するメッセージであった場合，変動後の計算性能で再スケジューリング処理を行うために，スケジューラが保持しているホスト H_i の計算性能の情報を更新する．最後に，9–11 行目で必要に応じて再スケジューリング処理を行う．*Trigger()* は再スケジューリングを実行するか否かを判断する関数であり，

4.5 *Trigger()* 関数

本稿では 3 種類の *Trigger()* 関数について説明する．*TriggerPerformanceChanged()* 関数は AHEFT [5] や Blythe らの手法[6] のような従来の動的再スケジューリング手法で用いられており，ホストの性能変動を再スケジューリングの条件としている．*TriggerTaskFinished()* 関数はタスクの実行終了を再スケジューリングの条件としている．*TriggerITSFinished()* 関数はワークフローに含まれる独立型タスク群に着目し，特定タスクの実行終了を再スケジューリングの条件としている．

```
1.function TriggerPerformanceChanged(message)
2.  if IsPerformanceChanged(message) = true then
3.    return true
4.  else
5.    return false
6.  end
7.end function
```

図 6　*TriggerPerformanceChanged()* のアルゴリズム
Fig. 6　Algorithm of *TriggerPerformanceChanged()*.

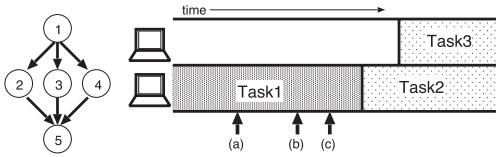


図 7　*TriggerPerformanceChanged()* の問題点
Fig. 7　Problem of *TriggerPerformanceChanged()*.

4.5.1 *TriggerPerformanceChanged()* 関数

従来の動的再スケジューリング手法は，ホストの性能変動が発生したときに再スケジューリング処理を行う．これはクリティカルパス上のタスクを実行するホストの計算性能が低下した場合，ワークフローの実行時間が大幅に延びてしまう可能性があるので，実行時間の悪化を抑えるために再スケジューリングを行う．*TriggerPerformanceChanged()* 関数のアルゴリズムを**図 6** に示す．*TriggerPerformanceChanged()* 関数がホストの性能変動の発生時に真を返すことで，スケジューラは再スケジューリング処理を行う．

しかし，ホストの性能変動が頻繁に発生する環境において，再スケジューリングの結果が利用されない場合がある．この問題について**図 7** を用いて説明する．図 7 の左図のワークフローを，右図の 2 台のホストで実行するとする．(a)，(b)，(c) のタイミングで性能変動が発生したとすると，各タイミングで再スケジューリング処理が行われる．そして Task2–4 は (c) の再スケジューリング結果に従って実行される．この場合，(a) と (b) のタイミングで計算されたスケジューリング結果は利用されないため，この処理がオーバヘッドとなる．*TriggerPerformanceChanged()* 関数はホストの性能変動が頻繁に発生する環境において，このような無駄な再スケジューリング処理が大量に発生してしまうという問題がある．

```
 1.flag = false
 2.function TriggerTaskFinished(message)
 3.   if IsTaskFinished(message) = true and flag = true then
 4.     flag = false
 5.     return true
 6.   else if IsPerformanceChanged(message) = true then
 7.     flag = true
 8.     return false
 9.   else
10.     return false
11.   end
12.end function
```

図 8　*TriggerTaskFinished()* のアルゴリズム
Fig. 8　Algorithm of *TriggerTaskFinished()*.

4.5.2　*TriggerTaskFinished()* 関数

4.5.1 項で述べたように，*TriggerPerformanceChanged()* はホストの性能変動が頻繁に発生する環境で大量のオーバヘッドが発生してしまう可能性がある．このオーバヘッドを減らすために *TriggerTaskFinished()* 関数を提案する．*TriggerTaskFinished()* 関数のアルゴリズムを**図 8** に示す．*TriggerTaskFinished()* 関数を用いた場合，ワークフローの各タスクが終了した時点で再スケジューリングを行う．ただし，前回の再スケジューリングから今回の再スケジューリングの間に，いずれのホストも性能変動が発生していない場合，再スケジューリングを行う必要がないので再スケジューリング処理を行わない．これはグローバル変数である *flag* 変数を用いて制御する．*flag* 変数は前回の再スケジューリングから性能変動が発生した場合に真がセットされる（7 行目）．これにより，4.5.1 項で述べたような無駄な再スケジューリング処理を実行せず，またワーストケース（性能変動がめったに発生しない環境）でも，*TriggerPerformanceChanged()* 関数を用いた場合と同じ再スケジューリング回数になる．

しかし，*TriggerTaskFinished()* 関数を用いた場合，あまり効果のない再スケジューリング処理が行われる可能性がある．これを**図 9** を用いて説明する．これは図 4 の (i) と `composite` タスクの実行状況を表している．2.2 節 (7) で述べたように，我々の想定するワークフローでは独立型タスク群が含まれるが，*TriggerTaskFinished()* 関数は独立型タスク群の有無にかかわらず各タスクの実行完了時に再スケジューリング処理を行う（(a)，(b)，(c)，(d) のタイミング）．しかし，`'sim 1000'` の後続のタスクである `'composite'` や `'renderer'`，

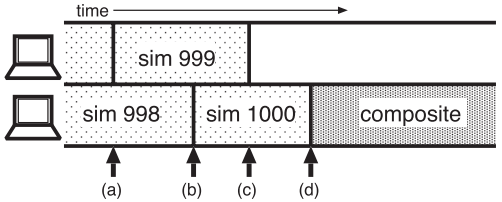


図 9　*TriggerTaskFinished()* の問題点
Fig. 9　Problem of *TriggerTaskFinished()*.

`'encoder'` は (d) のスケジューリング結果に従って実行され，(a)，(b)，(c) のスケジューリング結果はこれらのタスクの実行には利用されない．したがって，(a)，(b)，(c) であまり効果のない再スケジューリング処理が実行される可能性がある．

4.5.3　*TriggerITSFinished()* 関数

TriggerPerformanceChanged() 関数や *TriggerTaskFinished()* 関数で無駄な再スケジューリング処理を削減するために *TriggerITSFinished()* 関数を提案する．**図 10** に *TriggerITSFinished()* 関数のアルゴリズムを示す．*flag* 変数は *TriggerTaskFinished()* 関数のときと同様に，前回の再スケジューリングから今回の再スケジューリングの間にホストの性能変動が発生したか否かを記憶している．*TriggerITSFinished()* 関数は，以下のいずれかを満たしたとき，再スケジューリングを行うために真を返す．ただし，*TriggerTaskFinished()* 関数と同等の理由から，*flag* 変数が偽のときは再スケジューリングを行わない（11 行目・19 行目）．

(1)　いずれの独立型タスク群にも含まれていないタスク T_i が実行完了した場合
(2)　タスク T_i が含まれる独立型タスク群 I_j において，すべてのタスクが実行完了した場合

(1) の条件を満たすか否かを 6–12 行目で，(2) の条件を満たすか否かを 13–21 行目で調べている．また，*flag* 変数の制御を 23–25 行目で行っている．これらの処理により，*TriggerPerformanceChanged()* 関数や *TriggerTaskFinished()* 関数で実行される無駄な再スケジューリング処理を減らしている．

実際に各 *Trigger()* 関数において，再スケジューリングの実行回数がどの程度削減されているか例を用いて説明する．**図 11** は図 4 の独立型タスク群 (i) の実行完了前後の様子である．図 11 の (a) でホストの性能変動が発生したとする．*TriggerPerformanceChanged()* 関数を用いた場合，各 (a) のタイミングで再スケジューリングが行われるので，図 11 の時

```
 1.flag = false
 2.function TriggerTaskFinished(message)
 3.  if IsTaskFinished(message) = true then
 4.    T_i = GetFinishedTask(message)
 5.    I_j = GetContainedITS(T_i)
 6.    if I_j = nil then
 7.      if flag = true then
 8.        flag = false
 9.        return true
10.      else
11.        return false
12.      end
13.    else
14.      if IsAllTaskFinished(I_i) = true then
15.        if flag = true then
16.          flag = false
17.          return true
18.        else
19.          return false
20.        end
21.      end
22.    end
23.  else if IsPerformanceChanged(message) = true then
24.    flag = true
25.    return false
26.  else
27.    return false
28.  end
29.end function
```

図 10　*TriggerITSFinished()* のアルゴリズム
Fig. 10　Algorithm of *TriggerITSFinished()*.

間内で 10 回実行される．次に *TriggerTaskFinished()* 関数を用いた場合について説明する．*TriggerTaskFinished()* 関数では (b)，(c)，(d) のタイミングは再スケジューリング処理実行の候補になるが，2 回目の (b) と 3 回目の (b) の間で性能変動は発生していないため，3 回目の (b) のタイミングでは再スケジューリングは実行されない．したがって，この時間内では 5 回の再スケジューリングが実行される．最後に *TriggerITSFinished()* 関数を用いた場合について説明する．*TriggerITSFinished()* 関数では，(c) のタイミングで `sim` の独立

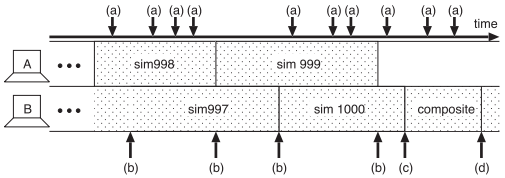


図 11　各 *Trigger()* 関数の実行例
Fig. 11　Example of each *Trigger()*.

型タスク群に含まれるすべてのタスクが実行完了となり，このときに再スケジューリングが実行される．また，タスク `composite` はいずれの独立型タスク群にも含まれていないため，タスク `composite` の実行が完了した (d) のタイミングでも再スケジューリングが実行される．したがって，この時間内では 2 回の再スケジューリングが実行される．

また，図 4 のワークフローを用いて，*TriggerTaskFinished()* 関数と *TriggerITSFinished()* 関数を用いた場合の再スケジューリングの実行回数を比較する．このとき，ホストの性能変動は頻繁に発生しているものとし，各タスクが実行完了する間に性能変動が 1 回以上発生しているものとする．*TriggerTaskFinished()* 関数を用いた場合，ワークフロー内にタスク数は 2,366 個存在するため，2,366 回のスケジューリング処理が行われる．これに対して *TriggerITSFinished()* 関数を用いた場合，9 回のスケジューリング処理が実行される．したがって，スケジューリング回数を約 1/250 に減らせる．

5. 評　　価

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った．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ものとした．

- 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 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．

表 1　タスクネットワークの生成条件
Table 1　Attributes of task networks.

タスク数	約 1,000
独立型タスク群の大きさ	100
タスクの計算量	100–200
タスクの基本通信量	100–200

- 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．

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，ランダムなタスクネットワークを生成した．

(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タスクネットワークを生成する．

(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，タスクネットワーク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，タスクネットワーク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，タスクネットワーク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．

(3)　タスクの総数が設定値以下であれば，(2) から繰り返す．

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，

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

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タスクネットワークとなり，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[10] や AMBER[28] 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．タスクネットワーク全体で実行する独立型タスク群の集合の数は，2.2 節で述べたようにユーザがワークフローを記述するため，パラメータサーベイのパラメータの個数と比較すると小さいと考えられる．したがって今回は独立型タスク群の集合の数に比べそのパラメータの個数を十分に大きい値にした．また数百の DNA 情報の解析を行える megablast が注目を集めており，シーケンスデータベースも数十におよぶことから，タスクネットワーク全体のタスク数を 1,000 程度と想定した．

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

また計算ホストの性能変動をシミュレーションするために，疑似タスクを計算ホストの半数で発生させ，ホストの計算性能は疑似タスクの個数に反比例するようにした．以下は各ホストの計算性能を求める式である．

$$\text{ホストの計算性能} = \frac{\text{ホストの基本計算性能}}{1 + dummy\_task(H_i)} \tag{2}$$

ホストの基本計算性能はスケジューラ等で用いた計算ホストの計算性能と同様で，あらかじめ実行環境情報として与えられたホストの計算性能である．また，$dummy\_task(H_i)$ 関数は計算ホスト H_i に存在している疑似タスク数を返す関数である．疑似タスクの生成頻度は単位時間に平均 λ 回のポアソン分布に従っている．一方，疑似タスクの消滅は時間間隔が平均 $1/\lambda$ の指数分布に従っている．

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

5.3.1　性能変動の頻度

計算ホストの性能変動の頻度による各条件の効果の違いを明らかにするために，スケジューリング長と再スケジューリングの呼び出し回数による比較評価を行った．**図 12** は各条件でスケジューリングを行ったときのスケジューリング長を，再スケジューリングを行わない場

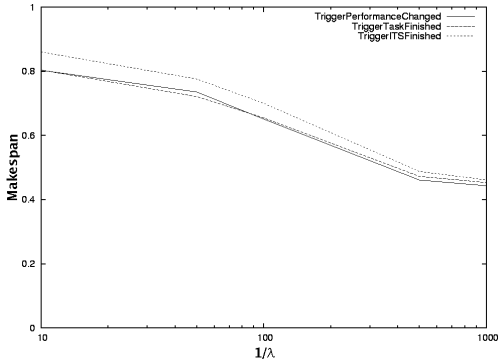


図 12　スケジューリング長（性能変動）
Fig. 12　Relationship of makespan and frequency of performance changes.

表 3　静的スケジューリングの実行回数（性能変動）
Table 3　Relationship of the number of rescheduling execution and frequency of performance changes.

$1/\lambda$	10	50	100	500	1,000
TriggerPerformanceChanged()	8,167	2,003	1,013	246	148
TriggerTaskFinished()	823	573	411	136	72
TriggerITSFinished()	33	33	32	32	26

合のスケジューリング長を 1 として正規化した値を示したグラフである．また各条件を用いた場合の再スケジューリングの実行回数を**表 3** に示す．計算コストと通信コストの比である CCR 値は 1.0 とした．

図 12 の評価結果から，すべての条件において $1/\lambda$ が大きいほど再スケジューリングを行わないときに対するワークフローの実行時間比が小さくなった．計算性能の変動の頻度が小さいほど計算ホストの性能低下の偏りが大きくなり，静的スケジューリングの性能は大きく低下する．一方，動的再スケジューリング手法では再スケジューリングを行うことで，実行効率の改善がされたと考えられる．

各条件の静的スケジューリングの実行回数について，表 3 の評価結果から，*TriggerPerformanceChanged()* 関数や *TriggerTaskFinished()* 関数は性能変動の頻度が高いほど実行回数は多くなったが，*TriggerITSFinished()* 関数の実行回数には大きな違いがなかった．また，*TriggerTaskFinished()* 関数は性能変動の頻度が高くても，*TriggerPerformanceChanged()* 関数ほど静的スケジューリングの実行回数が増加することはなかった．これは，*TriggerTaskFinished()* 関数や *TriggerITSFinished()* 関数がワークフローの情報を用いて，一定回数以上の再スケジューリングを行わないようにしているためである．また，*TriggerITSFinished()* 関数は独立型タスク群にも着目しているため，*TriggerTaskFinished()* 関数と比較して実行回数が大幅に少ない結果となった．一方スケジューリング長の増加は図 12 の結果から *TriggerITSFinished()* 関数と *TriggerPerformanceChanged()* 関数・*TriggerTaskFinished()* 関数を比較して約 5% に収まっている．したがって，*TriggerITSFinished()* 関数は約 5% のワークフローの実行効率の低下で，再スケジューリングの実行回数を大幅に減らすことができるといえる．

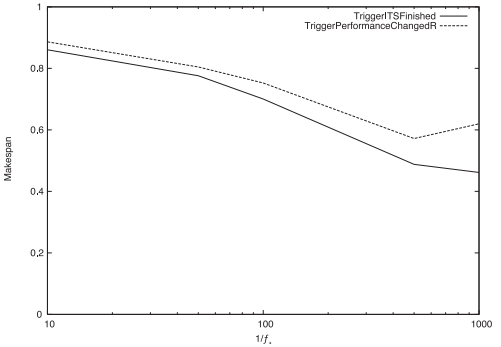


図 13　スケジューリング長の比較
Fig. 13　Makespans of reduced rescheduling executions.

5.3.2　再スケジューリング実行回数の単純な削減手法との比較

単純に再スケジューリング実行回数を削減した場合と提案手法による再スケジューリング実行回数を削減した場合の比較を行った．**図 13** は図 12 と同様に，各条件でスケジューリングを行ったときのスケジューリング長を，再スケジューリングを行わない場合のスケジューリング長を 1 として正規化した値を示したグラフである．*TriggerPerformanceChangedR()* 関数は単純に再スケジューリング実行回数を削減した条件で，ホストの性能変動が N 回発生するごとに再スケジューリングを実行する．この N は 5.3.1 項の評価結果を参考に，*TriggerPerformanceChangedR()* 関数が *TriggerITSFinished()* 関数の再スケジューリング実行回数と同程度になるように設定した．また，*TriggerPerformanceChangedR()* 関数と *TriggerITSFinished()* 関数の再スケジューリング実行回数を**表 4** に示す．

表 4　静的スケジューリングの実行回数の比較
Table 4　The number of static scheduling execution of reduced rescheduling executions.

$1/\lambda$	10	50	100	500	1,000
TriggerPerformanceChangedR()	31	35	35	37	27
TriggerITSFinished()	33	33	32	32	26

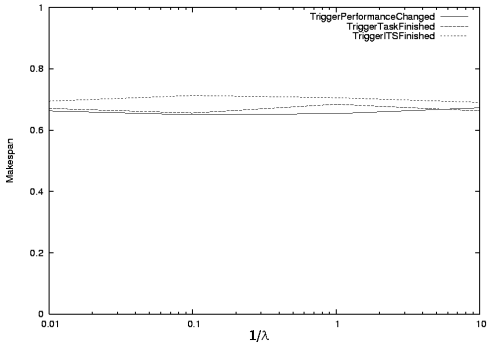


図 14　スケジューリング長（CCR）
Fig. 14　Relationship of makespan and CCR.

図 13 からすべての条件において，*TriggerPerformanceChangedR()* 関数は *TriggerITSFinished()* 関数よりスケジューリング長が悪化したことが分かる．*TriggerITSFinished()* 関数は効果の高いと思われる時点で再スケジューリングを行う．しかし，*TriggerPerformanceChangedR()* 関数はホストの性能変動の発生のみを条件としており，再スケジューリングの効果の高さについては考慮していない．したがって，同程度の再スケジューリングの実行回数においても *TriggerITSFinished()* 関数ほど良いスケジューリング長が得られなかったと思われる．

5.3.3　CCR 値

CCR 値の違いによる各条件の効果の違いについて評価を行った．**図 14** は図 12 と同様に，各条件でスケジューリングを行ったときのスケジューリング長を，再スケジューリングを行わない場合のスケジューリング長を 1 として正規化した値を示したグラフである．また各条件を用いた場合の再スケジューリングの実行回数を**表 5** に示す．ホストの性能変動の頻度は $1/\lambda = 100$ とした．

表 5 からすべての CCR 値において，*TriggerTaskFinished()* 関数と *TriggerITSFinished()* 関数で再スケジューリングの実行回数に約 10 倍の差があった．*TriggerTaskFinished()* 関数では独立型タスク群の各タスクが終了するごとに再スケジューリングが行われる．一方，*TriggerITSFinished()* 関数では独立型タスク群のすべてのタスクが終了した時点で 1 度だけ再スケジューリングが行われるので，大幅に呼び出し回数が減った．したがって *TriggerITSFinished()* 関数は *TriggerTaskFinished()* 関数より再スケジューリングの計算コストが小さくなると思われる．図 14 から *TriggerITSFinished()* 関数を用いたときのスケジューリング長は *TriggerPerformanceChanged()* 関数や *TriggerTaskFinished()* 関数と比較して 5% 程度増加したが，CCR 値による各条件の性能差は確認できなかった．したがって，CCR 値による違いにかかわらず，*TriggerITSFinished()* 関数は約 5% のワークフローの実行効率の低下で，再スケジューリングの実行回数を大幅に減らすことができるといえる．

表 5　静的スケジューリングの実行回数（CCR）
Table 5　Relationship of the number of static scheduling execution and CCR.

CCR	0.01	0.1	1.0	10.0
TriggerPerformanceChanged()	1,065	1,086	1,064	1,055
TriggerTaskFinished()	428	421	429	424
TriggerITSFinished()	35	32	35	33

6. おわりに

本稿では，計算ホストの性能変動やワークフローの実行状態を条件とし静的スケジューリングを実行する動的再スケジューリング手法を提案した．また，静的スケジューリングを実行する条件について各 *Trigger()* 関数を実装し評価を行った．抽象シミュレーションの結果から，計算ホストの性能変動が頻繁に発生する場合，*TriggerITSFinished()* 関数は再スケジューリング実行回数が大幅に少ないにもかかわらず，ワークフローのスケジューリング長は *TriggerTaskFinished()* 関数や *TriggerPerformanceChanged()* 関数を用いた場合と比較して 5% 程度の増加にとどまった．

今後は，ワークフローのスケジューリング長の増加を抑えつつ，再スケジューリングの実行回数を減らす *Trigger()* 関数の開発をしていく．また，より大規模なホスト数でも対応できるような手法の開発を行っていく．評価においては，ネットワークの挙動をより現実的なモデルにする等，精度を高めたシミュレーション評価を行うとともに，広域分散環境で実アプリケーションを用いた評価を行う必要がある．

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

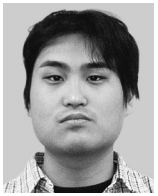
21013025,「タスクと実行環境の高精度モデルに基づくスケーラブルなタスクスケジューリング技術」）による．

参 考 文 献

1) Topcuoglu, H., Hariri, S. and Wu, M.-Y.: A high-performance mapping algorithm for heterogeneous computing system, *IPPS/SPDP Workshop on Heterogeneous Computing*, pp.3–14 (1999).
2) Dogan, A. and Ozguner, R.: 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：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，情報処理学会論文誌：プログラミング，Vol.2, No.1, pp.1–17 (2008).
4) 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).
5) Yu, Z. and Shi, W.: An adaptive rescheduling strategy for grid workflow applications, *Parallel and Distributed Processing Symposium, International*, Vol.0, pp.1–8 (2007).
6) Blythe, J., Jain, S., Deelman, E., Gil, Y., Vahi, K., Mandal, A. and Kennedy, K.: Task scheduling strategies for workflow-based applications in grids, *5th IEEE International Symposium on Cluster Computing and the Grid*, Vol.2, pp.759–767 (2005).
7) 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).
8) 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).
9) 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：コンピューティングシステム，Vol.47, No.SIG 12(ACS 15), pp.296–307 (2006).
10) 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).
11) Lipman, D.J. and Pearson, W.R.: Rapid and sensitive protein similarity searches, *Science*, Vol.227, pp.1435–1441 (1985).
12) Thompson, J.D., Higgins, D.G. and Gibson, T.J.: 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).
13) 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).
14) Sih., G.C. and Lee, E.A.: Dynamic-level scheduling for heterogeneous processor networks, *Proc. IEEE Symposium on Parallel and Distributed Processing* (1990).
15) Oh, H. and Ha, S.: A static scheduling heuristic for heterogeneous processors, *Euro-Par'96 Parallel Processing* (1996).
16) Boeres, C., Rios, E. and Ochi, L.S.: Hybrid evolutionary static scheduling for heterogeneous systems, *The 2005 IEEE Congress on Evolutionary Computation* (2005).
17) Kim, G.H. and Lee, C.S.G.: Genetic reinforcement learning for scheduling heterogeneous machines, *1996 IEEE International Conference on Robotics and Automation* (1996).
18) you 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).
19) Kwok, Y.-K. 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).
20) Armstrong, R., Hensgen, D. and Kidd, T.: 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).
21) 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).
22) Armstrong, R., Hensgen, D. and Kidd, T.: The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions, *7th IEEE Heterogeneous Computing Workshop* (*HCW'98*) (1998).
23) Blumofe, R.D. and Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing, *35th Annual Symposium on Foundations of Computer Science* (*FOCS'94*), pp.34–43 (1994).
24) Rahman, M., Venugopal, S. and Buyya, R.: A dynamic critical path algorithm for scheduling scientific workflow applications on global grids, *Proc. 3rd IEEE International Conference on e-Science and Grid Computing* (2007).
25) Beaumont, O., Legrand, A., Marchal, L. and Robert, Y.: Steady-State Scheduling Of Task Graphs On Heterogeneous Computing Platforms, *Int. J. Foundations of*

Computer Science, Vol.16, No.2, pp.163–194 (2003).
26) Gallet, M., Marchal, L., Vivien, F. and Lyon, U.D.: Allocating series of workflows on computing grids, *ICPADS. IEEE Computer Society* (2008).
27) Gallet, M., Marchal, L. and Vivien, F.: Efficient Scheduling of Task Graph Collections on Heterogeneous Resources, *IEEE International Symposium on Parallel and Distributed Processing* (2009).
28) The Amber Molecular Dynamics Package. http://ambermd.org/

(平成 22 年 12 月 17 日受付)
(平成 23 年 3 月 29 日採録)

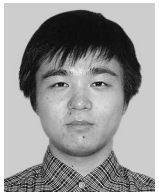


松本　真樹（学生会員）

2007 年三重大学工学部情報工学科卒業．2010 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．現在，同大学院工学研究科情報工学専攻博士後期課程在学中．広域分散環境における大規模並列処理に興味を持つ．

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）．

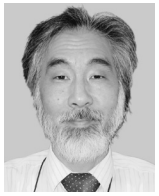


佐々木敬泰（正会員）

1998 年広島市立大学情報工学科卒業．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．

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．