修士論文

題目

大規模ワークフロー用
スケジューラ実装のための
ライブラリ設計

指導教員

大野　和彦　　講師

平成25年度

三重大学大学院 工学研究科 情報工学専攻
コンピュータソフトウェア研究室

鈴木　康平 (412M511)

内容梗概

近年, 大規模計算の需要がますます増大しており，コスト面で優れる大規模分散環境での並列処理に注目が集まっている．大規模計算にはワークフローとして表現できるものが多くあり，我々はワークフロー型並列処理を容易に実現するタスク並列スクリプト言語 MegaScript を開発している．

効率のよいワークフロー実行には個々に適したタスクスケジューラの選択，実装が必要になる．ワークフローシステム上へのスケジューラ実装は，処理系内部の低レベルなコードを考慮したプログラミングが必要となる．また，複数のスケジューラを実装する場合，何度も共通するコードを書かなくてはならない．そこで低レベルな記述の隠蔽とコードの共通化により，このような問題を解決するライブラリを設計・実装した．

本ライブラリを使用してスケジューラを実装し，使用しない場合と記述量，スケジューリング時間を比較した結果，不使用のスケジューラに約 30 %存在した低レベルな記述がなくなり，スケジューリング時間に関しても実用上許容できる範囲に収まった．

三重大学大学院　工学研究科

Abstract

The demand for large-scale computing is increasing. A widely distributed system may be a practical solution for such purpose. Because much large-scale computing are represented as workflows, we are developing a task parallel script language MegaScript for workflow model in a widely distributed environment.

Application-specific implementation of the scheduling algorithm is often needed for efficient workflow execution. When the user implements a scheduler onto a workflow system, the user must have enough knowledge of low-level system code. The user often repeats writing similar code because the scheduling algorithms of the same category require similar features. To solve these problems, we propose a library which hides the low-level system code, and shares the scheduler code in the same category.

We implemented several schedulers using/not using our library, and compared the number of lines of scheduler code and the scheduling time. Conventional scheduler code included about 30% low-level code. However, low-level code was eliminated from the scheduler code using our library. The increased scheduling time was small enough for all schedulers to use our library practically.

三重大学大学院　工学研究科

目 次

図 目 次

表 目 次

1　はじめに

近年，気象予測や遺伝子解析のような分野において大規模計算の需要が高まってきている．これらの計算は主にスーパーコンピュータで行われてきたが，膨大な費用がかかるためコストパフォーマンスに優れた大規模分散環境での並列処理に注目が集まっている．大規模計算にはワークフローとして表現できるものが多くあり，我々はワークフロー型並列処理を容易に実現するタスク並列スクリプト言語 MegaScript [1, 2] を開発している．

大規模ワークフローを効率よく実行するためには，どのタスクをどのホストにどの順序で割り当てるかというタスクスケジューリングが重要である．タスクスケジューリングの研究は盛んに行われているが，各スケジューリングアルゴリズムには向き，不向きがあり万能なものは存在しない．そのため，アプリケーションに適したスケジューラの選択，実装あるいはスケジューリング手法の組み合わせが必要となる．しかしながら，ワークフローシステムにスケジューラを組み込むことはそのシステムの低レベルなコードへの豊富な知識が必要となるため非常に負担が大きい．また複数のスケジューラを実装する場合，ユーザは共通するコードを何度も記述しなくてはならないため煩雑である．さらにデマンドドリブン型動的スケジューラを実装するためには処理系にハードコーディングしなければならず，他のスケジューラと組み合わせることは困難である．

このようなスケジューラ実装の問題を解決するために，我々は MegaScript 上にライブラリを設計･実装した．本ライブラリは処理系へのインタフェースとストラテジクラス群から成る．インタフェースはイベント処理や処理系のリソースへのアクセスのような低レベルなコードを隠蔽する．また，ストラテジクラス群は共通するコードの複数回記述をなくすために，各スケジューラカテゴリ内で共通する処理を提供する．本ライブラリにより処理系の低レベルな要素に関わるコードやスケジューラの各カテゴリ内において共通するコードがなくなり，スケジューラ実装の負担を軽減することができる．また，実装したストラテジクラスを組み合わせることで任意のスケジューリング手法の組み合わせを実現することができる．

以下，2 章では背景である MegaScript，タスクスケジューリング，そして MegaScript 上へのスケジューラ実装について記述する．3 章では他の並列システムと MegaScript を比較し，スケジューラ実装について議論する．4 章ではライブラリの設計の詳細を説明し，5 章でライブラリを使用したスケジューラと使用していないスケジューラの比較評価結果を示す．

三重大学大学院　工学研究科

最後に 6 章でまとめを行う．

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並列処理を記述するためのプログラミング言語であり，ベースの言語として Ruby [3, 4] を用いている．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 MegaScript のコード例を図 2.1 に，図 2.1 のコードにより生成されるワークフローを図 2.2 に示す．MegaScript はワークフローの容易な構築のために API クラスを提供する．API クラスには `Task` クラス，`Stream` クラス，そして `Host` クラスなどがあり，それぞれタスク，ストリーム，ホストを操作，制御するためのクラスである．また，`TaskArray` クラスは `Task` クラスの配列であり，`Scheduler` クラスはスケジューラを表している．ユーザは図 2.1 のように `Task` クラスのオブジェクトや `Stream` クラスのオブジェクトを生成し，`Stream` クラスの `connect()` メソッドを用いて `Task` クラスのオブジェクトをつなぎ合わせることでワークフローを構築できる．本論文では `Task` クラスや `Host` クラスをもとに生成されるオブジェクトを単に `Task` オブジェクト，`Host` オブジェクトと呼ぶこととする．

2.2　タスクスケジューリング

タスクスケジューリング [5] の目的は実行効率を最大限向上させるために，タスクをどのホストでどの順序で実行するかを決定することである．タスクスケジューリングアルゴリズムは次のような観点で分類できる．

```
p = TaskArray.new(N,
“producer”)
c1 = Task.new(“consumer”)
c2 = Task.new(“consumer”)
c3 = Task.new(“consumer”)
s = Stream.new
s.connect(p, IN)
s.connect(c1, OUT)
s.connect(c2, OUT)
s.connect(c3, OUT)
activate_all
Scheduler.new.schedule
```

図 2.1: MegaScript コード例

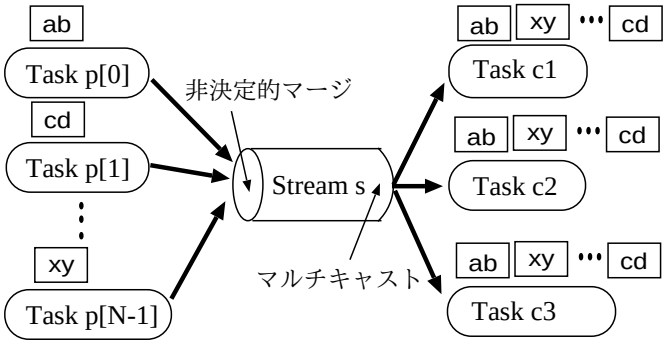

図 2.2: 生成されるワークフロー

- 分類 A(スケジューリングのタイミング)：
  静的スケジューリング / 動的スケジューリング / ハイブリッドスケジューリング

- 分類 B(タスク間依存の有無)：
  独立型タスクスケジューリング / DAG(Directed Acyclic Graph) スケジューリング

静的スケジューリングはワークフロー実行前に全てのタスクをホストに割り当てる．また，動的スケジューリングはワークフロー実行中に発生するイベントに応じてスケジューリングを行う．ハイブリッドスケジューリングは静的スケジューリングと動的スケジューリングを組み合わせた手法である．ハイブリッドスケジューリングでは，ワークフロー実行前に静的スケジューリングアルゴリズムを使ってホストに割り当て，ホスト性能が変動したときにそのアルゴリズムにより再スケジューリングを行う．

独立型タスクスケジューリングは互いに依存関係のないタスクを扱い，タスクの計算量やホストの性能を用いて割当先のホストを決定する．DAG スケジューリングは依存関係のあるタスクに対して適用され，タスクの計算量やホストの性能の他にタスク間の依存関係も考慮する．本論文では議論を簡単にするために，与えられたワークフローは実行中にタスクが新たに追加されたり，削除されたりしないものとする．

2.3 MegaScript ランタイム上へのスケジューラ実装

各スケジューリングアルゴリズムには適するワークフローの形状や実行環境があり，すべてのワークフローに対して単一のスケジューリングアルゴリズムを用いることは最適ではない．そのため，アプリケーションに適したスケジューラの選択，実装が必要になる [6, 7].

MegaScript ではスケジューラがプラグインになっているため，新しいスケジューラを処理系上に容易に組み込むことができる．ユーザは Ruby を用いて MegaScript ランタイム上にスケジューラを実装することができ，スケジューラを組み込む際にコンパイルやランタイムの再構築を行う必要がない．

しかしながら，一からスケジューラを実装することはランタイムの低レベルな要素に対する知識が必要となるため非常に負担が大きい．また，複数のスケジューラを実装する場合，似たような処理を複数回記述しなければならないので煩雑である．さらに，MegaScript ランタイムは Ruby 上でのイベント処理をサポートしていない．そのため，動的あるいはハイブリッドスケジューラを実装する場合，ユーザは大量にあるランタイムコード内の適切な場所にスケジューリング処理をハードコーディングしなくてはならない．これにより，任意のスケジューリングアルゴリズムを組み合わせることも非常に困難になる．我々は以上のような問題を解決するようなライブラリを設計する必要がある．

3 関連研究

MegaScript と同じようなワークフローシステムとして Pegasus [8] や Swift [9] がある．

Pegasus は XML を用いて書かれたユーザの抽象ワークフローを実行ワークフローに変換しグリッドに割り当てる．ランタイムでは DAGMan [10] や Condor [11] が各タスクの実行を管理している．利用可能なホストの性能を考慮して静的マッピングを最適化することはできるが，ユーザの立場から動的スケジューリングを制御することはランタイムの下位層がこれを処理しているため難しい．

Swift ではワークフローを定義するためにスクリプト言語 SwiftScript を用いる．ユーザスクリプトは抽象計算プランに変換され，実行エンジンである Karajan [12] により実行される．そしてスケジューリングは実行

時に実行エンジンにより行われるので，スケジューラの振る舞いを制御するためにはKarajanを直接拡張しなくてはならない．

これらのワークフローシステムと比較すると，MegaScriptはユーザスクリプトを実行時に直接解釈する点とスケジューラをユーザレベルで実装できるという点で優位性がある．すなわち，ユーザは追加インストールやシステムの再構築をすることなくスケジューラを容易に置き換えたり，拡張したりすることができる．

4　ライブラリの設計

4.1　概要

本ライブラリの目的は任意のカテゴリのスケジューリングアルゴリズムの実装を可能にし，その負担を軽減することである．この目的を達成するためには2.3節で述べた問題を解決する必要がある．そのために我々は`Interface`クラスとストラテジクラス群を設計した．

`Interface`クラスは2.2節の分類Aの観点からスケジューラ実装をサポートする．このクラスの役割は処理系からのイベント発生通知を隠蔽し，スケジューリングに必要なイベントハンドラをコールバックメソッドとして提供することである．ユーザは`Interface`クラスを継承し，必要なイベントハンドラをオーバライドすることで任意のスケジューラを容易に実装することができる．すなわち，`Interface`クラスを用いることによりユーザはスケジューリング処理を処理系にハードコーディングする必要がなくなり，実装の負担が軽減される．また，イベントハンドラごとに異なるスケジューリング手法を実装することができるので，任意のスケジューラを組み合わせることも容易になる．

ストラテジクラス群は2.2節の分類Bの観点からコードの共通化を図る．ストラテジクラス群はいくつかのクラスを持っており，各クラスは分類B内のそれぞれにおいて共通の処理をメソッドとして持つ．例えば，独立型タスクスケジューリングではあるホスト上で最短で終了するタスクの取得，DAGスケジューリングではあるタスクの先行タスク群の取得などが共通の処理として挙げられる．ユーザは実装したいスケジューリング手法に適したストラテジクラスを継承し，その手法固有の処理のみを記述する．これにより容易にスケジューリング手法が実装できると同

時に，同じ分類の手法を複数実装する場合でもその分類において固有の処理を複数回記述するという煩雑さがなくなる．

本ライブラリを使用することにより，次のような利点がある．

- 処理系の低レベルな要素に対する知識がなくても任意のスケジューラを実装することができる
- 同じ分類に属する手法を複数実装する場合でも，その分類において固有の処理を複数回記述することがなくなる
- 実装された手法は複数のスケジューラで再利用でき，任意の手法同士を組み合わせることもできる

一方で次のような欠点もある．

- スケジューラクラスとストラテジクラスの 2 つのクラスを実装しなければならない
- 本ライブラリは継承やコンポジットを使用しているため，メソッド呼び出しのオーバヘッドによりスケジューリング時間が増加してしまう

しかしながら，我々は欠点に比べて利点の方が大きいと考えている．

4.2 Interface クラス

MegaScript 上にスケジューラを実装する際，次のような問題が発生する．

イベント処理
: スケジューリングに必要なイベント処理を扱うためには処理系の低レベルなコードの理解と修正をしなくてはならない．

スケジューラ再実装
: 我々が MegaScript 処理系をアップグレードした場合，ユーザはその仕様に沿ってスケジューラを再実装しなければならない．

スケジューリング手法の組み合わせ
: ユーザは動的スケジューラを実装する際に処理系へのハードコーディングをしなくてはならないため，任意の手法を組み合わせることは非常に困難である．

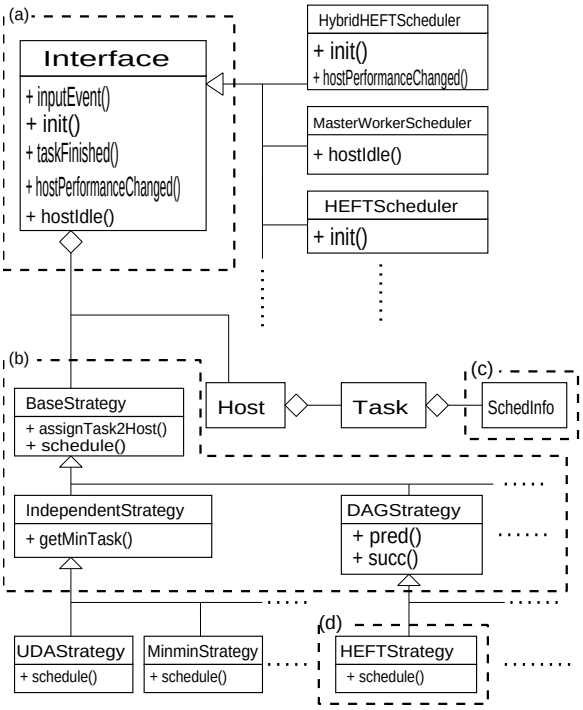

図 4.3: 本ライブラリのクラス図

これらの問題を解決するために，我々は `Interface` クラス (図 4.3(a)) をユーザに提供する．図 4.4 に `Interface` クラスの疑似コードを示す．`Interface` クラスは `inputEvent()` メソッドを持つ．MegaScript 処理系はワークフロー実行中に何らかのシステムイベントが発生したときに `inputEvent()` メソッドを呼び出す．そして，このメソッドは発生したイベントに関連のあるイベントハンドラを呼び出す．

Interface クラスにはスケジューリングに必要なイベントハンドラを用意している．ワークフロー実行中にはタスクの終了，ホストの性能変動，タスク間通信の発生などさまざまなシステムイベントが発生する．しかしながら，これらのイベントが全てスケジューリングに用られるわけではないので，スケジューリングによく使われるイベントへのハンドラのみを提供する．例えばタスクの終了やホストの性能変動は，多くの動的 / ハイブリッドアルゴリズムで使われるためこれらへのイベントハンドラを用意する．一方で，タスク間通信の発生はスケジューリングアルゴリズムでは使われる可能性が極めて低いので，このイベントに対するハンドラは用意しない．さらに，ほとんどのスケジューラにおいて初期スケジューリングが行われるので，ワークフローが実行可能状態になったことを示す *init* イベントを通知できるようにして，それに対するイベントハンドラを用意する．

ここで，本ライブラリを使用した場合のハイブリッドスケジューラ実装についてコードを用いて説明する．ハイブリッドスケジューラ (HybridHEFTScheduler) のコード例を図 4.5 に示す．MegaScript 処理系上にスケジューラを実装する場合，SchedulerLib::Interface を継承する．‘class A < B’ は A が B を継承していることを表す構文である．また，オーバライドされているメソッドを下線で示す．ハイブリッドスケジューラはワークフロー実行前に静的アルゴリズムにより初期スケジューリングを行い，ホスト性能が変動するたびにその静的アルゴリズムにより再スケジューリングを行う．HybridHEFTScheduler はその静的アルゴリズムとして HEFT [13] を用いてスケジューリングを行う．そのため，HEFT アルゴリズムを実装した HEFTStrategy をスケジューリングアルゴリズムとして登録する (3 行目)．HEFTStrategy は次節で説明するストラテジクラス群を用いている．HybridHEFTScheduler は初期スケジューリングとホスト性能が変動したときにスケジューリングを行うため，init() メソッドと hostPerformanceChanged() メソッドをオーバライドしその中で HEFT アルゴリズムを呼び出すようにする (5〜7，8〜10 行目)．

このようにユーザは必要なイベントハンドラをオーバライドし，そこにスケジューリング処理を記述することにより，処理系へのハードコーディングをせずにスケジューラを実装できる．

4.3 ストラテジクラス群

スケジューリング手法を実装する際に次のような問題がある．

共通するコードの複数回記述
: 手法の各分類内で共通の処理が存在するため，ユーザが類似した手法を実装する場合に共通するコードを何度も書かなくてはならない．また，これらの処理内には処理系の低レベルな要素を扱う部分も存在するため実装が困難である．

変数・メソッドの競合
: 手法によってタスクに必要な属性が異なるため，ユーザが Task クラスにメンバ変数として属性を追加したい場合がある．例えばタスクの優先度を与える手法では Task クラスに優先度を表す変数@rank が必要になる．Ruby では既に定義されたクラスに対して後からメンバ変数やメソッドを追加できる特性があり，ユーザはこの特性を用いて Task クラスに@rank を容易に追加できる．しかしながら，ユーザにより追加された変数が，Task クラスに既にある変数や我々が今後 Task クラスの内部仕様を変更する際に追加する変数と競合する可能性がある．

共通するコードの複数回記述を解決するためにストラテジクラス群 (図 4.3 (b)) を提供する．まず，どのスケジューラにも共通するような処理を持つ BaseStrategy クラスを設計した．例えば，タスク T_i をホスト H_p 上に割り当てる処理はどのスケジューリングアルゴリズムにも必要である．そのため，この処理を assignTask2Host(T_i，H_p) メソッドとして BaseStrategy クラスに実装した．また，この処理では Task オブジェクトを Host オブジェクトの持つキューに格納する．Host クラスには Task オブジェクトを格納するキューがメンバ変数として与えられており，このキューを操作するためには Host クラスの内部実装の知識が不可欠である．しかしながら，assignTask2Host メソッドに Task オブジェクトと Host オブジェクトを与えれば，内部実装への知識なしにタスクをホストに割り当てることができる．

また，分類 B における各スケジューリング手法実装のためのクラスを BaseStrategy のサブクラスとして提供する．本論文では独立型タスクスケジューラと DAG スケジューラのための IndependentStrategy クラスと DAGStrategy クラスを例に説明する．各スケジューリング手法の分

表 4.1: ストラテジクラス群と主なメソッド

ストラテジクラス	主なメソッド	機能
BaseStrategy	assignTask2Host(T_i, H_p)	タスク T_i をホスト H_p に割り当てる
	getExecutionTime(T_i, H_p)	H_p 上での T_i の予想される実行時間を取得する
IndependentStrategy	getMinTask(H_p)	H_p 上で最短で終了するタスクを取得する
	getIndependentTasks(*workflow*)	ワークフロー内の独立型タスクの集合を取得する
DAGStrategy	pred(T_i)	T_i の先行タスク群を取得する
	succ(T_i)	T_i の後続タスク群を取得する
	getBottomTasks()	最下位タスクを取得する

類内でそれぞれ共通の処理を持つ．例えば，ホスト H_p 上で最も実行時間の短いタスクを取得する処理は独立型タスクスケジューリングではよく行われるが，DAG スケジューリングでは依存関係の制約がありこの処理は行われない．よって，この処理を getMinTask(H_p) メソッドとして IndependentStrategy クラスに実装する．一方，タスク T_i の先行タスク群あるいは後続タスク群を取得する処理は DAG スケジューリングではよく行われるが，独立型タスクスケジューリングでは扱うタスク間に依存関係がないためこれらの処理は必要ない．したがって，これらの処理をそれぞれ pred(T_i) メソッド，succ(T_i) メソッドとして DAGStrategy クラスに実装する．また，ワークフロー内の独立型タスクの集合を取得したり，DAG グラフのタスク間依存解析をしたりするために Task クラスや Stream クラス内のメンバ変数を使用する．この部分を IndependentStrategy クラスや DAGStrategy クラスのメソッドの中に隠蔽することで，内部仕様への知識が不要になる．

表 4.1 にストラテジクラス群と主なメソッドを示す．ユーザは実装したいスケジューリング手法に適したストラテジクラスを継承し，その固有の処理のみを記述することで容易に手法を実装できる．また，類似する手法を複数実装する場合にその分類において共通する処理を複数回実装する必要がなくなる．

変数・メソッドの競合を解決するために，我々はスケジューラのために追加した属性を保持する SchedInfo クラス (図 4.3 (c)) を提供する．各 Task オブジェクトはメンバ変数として SchedInfo オブジェクトを持つ．これにより Task オブジェクトが生成される時に自動的に SchedInfo オブジェクトも生成され，対応づけられる．すなわち，ユーザは膨大な数の Task オブジェクトに手動で SchedInfo オブジェクトを対応づける必

要がない．SchedInfo クラスはデフォルトではメンバ変数，メソッドを持たず，ユーザがスケジューラに必要なメンバ変数やメソッドを追加拡張する．前述したように Ruby は動的拡張をサポートしており，定義済みのクラスに後からメンバ変数やメソッドを追加することができる．図 4.6 に MegaScript 処理系内の Task クラスの定義を，図 4.7 にユーザスクリプト内での Task クラスの動的拡張の例を示す．図 4.6 のような Task クラスに対して，ユーザは図 4.7 のように Task クラスを拡張し，メンバ変数@rank と setRank(r) メソッドを追加できる．この場合，我々開発者側が Task クラスに仕様変更等の理由により，後からメンバ変数@rank を追加するとメンバ変数名の競合が発生してしまう．そこで，SchedInfo クラスにより間接的に Task クラスを拡張することでこの問題を解決する．図 4.8 に SchedInfo クラスの拡張例を示す．ユーザは図 4.8 のようなコードを実装したストラテジクラス内に記述することで，SchedInfo クラスを用いた Task クラスの間接的な拡張が可能になる．また，Task クラスに既に存在する変数や内部仕様の変更により Task クラスに追加される変数と競合することがなくなる．

ここで本ライブラリを使用して実装したストラテジクラスと使用せずに実装した同等のものを比較する．図 4.9 に本ライブラリ不使用の HEFTStrategy クラスの疑似コードを示す．MegaScript 処理系は生成された Task，Stream オブジェクトをマスタホストオブジェクトの schedule_queue に格納する．そのため，そのキューを参照する必要がある (4 行目)．HEFTStrategy はまず参照したキューの各オブジェクトを Task オブジェクトと Stream オブジェクトに分けて最も下位の Task オブジェクトを検索する (7〜16 行目)．task_queue, stream_queue，bottomtask_queue はそれぞれタスク，ストリーム，最下位タスクを格納するためのキューである．そしてタスクの優先度が最下位タスクから最上位タスクに向かって計算され (17〜26 行目)，この処理は全てのタスクの計算が終了するまで繰り返される．predtask_queue はあるタスクの先行タスクを格納するためのキューである．最後に全タスクは優先度の降順でソートされ，HEFT アルゴリズムに従ってホストに割り当てられる．低レベルな知識が必要なコードを実線で，DAG スケジューリング手法において共通するコードを波線で示す．

図 4.10 に本ライブラリを使用した HEFTStrategy クラス (図 4.3 (d)) の疑似コードを示す．図 4.10 には図 4.9 に存在した実線部，波線部のようなコードが出てこない．すなわち，ユーザは低レベルな仕様に関わるコードや各分類内で共通するコードを記述する必要がなくなるので，ス

ケジューリング手法実装の負担は軽減される．

5　評価

5.1　評価方法

本ライブラリの有用性を示すために，本ライブラリを使用したスケジューラと使用していないスケジューラを実装し評価した．3 つの静的スケジューラ (UDA [14]，Min-min [15]，HEFT [13])，2 つのデマンドドリブン型動的スケジューラ (マスタ・ワーカ，タスクスティール [16]) そして 2 つのハイブリッドスケジューラを実装した．2 つのハイブリッドスケジューラは初期スケジューリング時とホスト性能変動の発生時に静的スケジューリングアルゴリズム (UDA，HEFT) を用いる．

本ライブラリを用いることによりスケジューラの記述量を削減できることが予想される一方，継承やコンポジットを利用しているためメソッド呼び出しのオーバヘッドによりスケジューリング時間が増加する可能性が高い．そこで実装したスケジューラの記述量とスケジューリング時間の評価を行った．スケジューリング時間の計測には AMD Athlon CPU (3.0GHz)，2GB のメモリを搭載した PC を使用した．静的 / ハイブリッドスケジューラは遺伝子解析で用いられる Epigenomics workflow [17] を 32 台のホストに割り当てる時間を計測した．動的スケジューラは独立型タスクの集合に対し，タスク数を閾値にした場合とタスクの合計計算量を閾値にした場合のスケジューリング時間を計測した．

5.2　スケジューラコードの記述量

表 5.2〜 5.4 はそれぞれ静的，動的，ハイブリッドスケジューラの記述量を示す．また，表の依存部分は処理系の内部仕様に関する低レベルなコード記述の行数である．

表 5.2 から，本ライブラリを使用することにより静的スケジューラの記述量が削減されていることがわかる．これはストラテジクラス群によりコードの共通化がなされているため，その分類内で共通するコードを記述する必要がなくなったことが理由である．また，依存部分が不使用のコード全体の約 30 %を占めている．そのため，ユーザは処理系の内部仕様を知らなければこれらのスケジューラを実装することができない．ま

た，内部仕様が変更された場合にこの依存部分を再実装しなくてはならない．しかしながら，Interface クラスやストラテジクラス群が依存部分を隠蔽するため，本ライブラリを使用することでこの部分はなくなる．

一方，表 5.3 から，動的スケジューラでは本ライブラリを使用することにより記述量が増加していることがわかる．これは動的スケジューラの記述量自体が少なく，共通するコードや隠蔽できるコードがほとんどないためである．しかしながら，依存部分に関しては処理系にハードコーディングしなければならず，他のスケジューリング手法と組み合わせることが非常に困難である．本ライブラリを使用することで動的スケジューラを処理系の内部仕様と完全に分離できるので，任意の手法と組み合わせることも容易になる．

さらに，表 5.4 から，静的スケジューラと同様にハイブリッドスケジューラの記述量が削減されていることがわかる．また，表 5.2 の静的 UDA，HEFT とハイブリッド UDA，HEFT とを比べると，本ライブラリを使用することでほとんど記述量が増加しないことが示される．すなわち，ユーザは静的スケジューラを実装する負担とほぼ同等の負担でハイブリッドスケジューラを実装することができる．さらに，静的スケジューラとデマンドドリブン型動的スケジューラを組み合わせるようなハイブリッドスケジューラを実装する場合でも，デマンドドリブン型動的スケジューラで増加する記述量は他の部分で増加する記述量より少ないので全体をみると削減されるといえる．

表 5.2: 静的スケジューラコードの記述量 (行数)

	UDA	Minmin	HEFT
不使用	108	133	194
依存部分	34	38	53
使用	65	93	146
依存部分	0	0	0

表 5.3: デマンドドリブン型動的スケジューラコードの記述量 (行数)

	MasterWorker	WorkStealing
不使用	12	43
依存部分	4	15
使用	18	57
依存部分	0	0

5.3 スケジューリング時間

表 5.5 に静的スケジューラのスケジューリング時間を示す．タスク数を変化させて計測を行った結果，本ライブラリを使用することでスケジューリング時間が増加した．UDA と HEFT では 1 %ほどの増加であるので無視できる範囲だといえる．Min-min では 20 %〜30 %ほど増加してしまっ

表 5.4: ハイブリッドスケジューラコードの記述量 (行数)

	HybridUDA	HybridHEFT
不使用	126	213
依存部分	49	67
使用	73	153
依存部分	0	0

ている．これは `MinMinStrategy` が他のストラテジクラスに比べてスーパークラスのメソッド呼び出しが多く，そのオーバーヘッドが原因であると考えられる．しかしながら，この増加はワークフロー全体の実行時間と比較すれば実用上許容できる範囲だと考える．

動的スケジューラのスケジューリング時間に関しては，あるホストのキューにある独立型タスクの集合を分割して，別のホストに転送することを想定した．そのとき一度に送るタスク数を閾値により決定することにし，閾値としてタスク数とタスクの合計計算量の 2 通りの場合で計測を行った．各タスクの計算量については 100 で固定し，タスクの合計計算量での計測ではこの値を足していくことにより一度に転送するタスク数を決定するようにした．表 5.6， 5.7 はそれぞれタスク数とタスクの合計計算量を閾値にしたスケジューリング時間を示す．表 5.6 から，本ライブラリを使用したことにより増加していることがわかるが，差が数マイクロ秒と非常に小さく，タスクの転送時間 (表 5.8 参照) やタスクの実行時間と比べれば無視できる範囲である．また表 5.7 から，増加する時間が数マイクロ秒と非常に小さいこと，表 5.6， 5.7 を比較するとタスクの合計計算量を求める処理のオーバヘッドがほとんどないことがわかる．この結果からデマンドドリブン型動的スケジューラのスケジューリング時間の増加は，本ライブラリを使用してもほとんどなく，転送前に何らかの計算を行ってもそのオーバヘッドは通信時間などと比べ非常に小さく無視できる範囲といえる．

ハイブリッドスケジューラのスケジューリング時間は，タスク数を 1,004 で固定し，ホストの性能変動の回数を 100，500，1,000 と変化させその合計時間を計測した．また，性能変動が発生するたびに全タスクを再スケジューリングするワーストケースを評価した．表 5.9 はハイブリッドスケジューラの合計スケジューリング時間を示す．静的や動的スケジューラと同様に，本ライブラリを使用することでスケジューリング時間は増加

表 5.5: 静的スケジューラのスケジューリング時間 (秒)

	UDA		Minmin		HEFT	
タスク数	不使用	使用	不使用	使用	不使用	使用
1,004	2.02	2.04	6.60	8.69	0.83	0.84
5,004	64.71	64.90	194.78	245.34	22.28	22.56
10,004	309.65	311.13	896.21	1074.71	94.57	94.97

表 5.6: タスク数を閾値にした場合のスケジューリング時間 (ミリ秒)

閾値	不使用	使用
1	0.0018	0.0029
2	0.0023	0.0034
4	0.0029	0.0040
8	0.0045	0.0059
16	0.0090	0.010

表 5.7: タスク合計計算量を閾値にした場合のスケジューリング時間 (ミリ秒)

閾値	不使用	使用
100	0.0019	0.0033
200	0.0026	0.0041
400	0.0036	0.0056
800	0.0062	0.0086
1,600	0.012	0.015

した．しかしながら，UDA，HEFT どちらでも数%の増加であり実用上許容できる範囲であるといえる．

6 終わりに

本論文では MegaScript 処理系上へのスケジューラ実装の負担を軽減するライブラリの設計・実装について述べた．本ライブラリは `Interface` クラスとストラテジクラス群から成り立つ．`Interface` クラスは処理系のイベント関連部分の低レベルコードを隠蔽し，スケジューリングに必要なイベントハンドラを提供する．`Interface` クラスを継承し，必要なイベントハンドラをオーバライドすることにより，ユーザは処理系への知識なしで動的およびハイブリッドスケジューラを実装することができる．ストラテジクラス群はスケジューラの各分類内で共通する処理をまとめたいくつかのクラスから構成され，ユーザは適したストラテジクラスを継承することで容易にスケジューリング手法を実装できる．また，類似した手法を複数実装する場合に，ユーザは共通するコードを複数回記

表 5.8: ホスト間のタスク転送時間 (ミリ秒)

タスク数	タスク転送時間
10	0.36
100	2.93
1,000	28.14
5,000	2164.71
10,000	3321.35
100,000	4023.30

表 5.9: ハイブリッドスケジューラの合計スケジューリング時間 (秒)

	DynamicUDA		DynamicHEFT	
性能変動回数	不使用	使用	不使用	使用
100	203.62	204.20	81.60	83.73
500	1,023.23	1,023.40	413.38	418.87
1,000	2,043.06	2,044.26	828.33	838.67

述しなくてもよくなる．

本ライブラリを使用したスケジューラを実装し，使用していない場合と記述量，スケジューリング時間を比較した．記述量に関しては不使用のスケジューラに約 30 %含まれていた処理系への知識が必要となるコードがなくなった．動的スケジューラではコード量自体が少なく共通化・隠蔽できる部分がほとんどないため，記述量が増えてしまった．しかしながら，処理系へのハードコーディングが必要なくなったため，任意のスケジューリング手法を組み合わせることが容易になった．スケジューリング時間に関しては静的 / ハイブリッドスケジューラではほとんどのアルゴリズムにおいて数%の増加に収まった．Min-min のみ 20〜30 %ほど増加してしまったがワークフロー全体の実行時間に比べれば十分小さく許容できる範囲だといえる．動的スケジューラでも同様にオーバヘッドが発生しスケジューリング時間が増加したが，タスクの転送時間や実行時間と比べると十分小さく実用上問題がない範囲であった．

本論文では Epigenomics Workflow のみを扱ったが，他のワークフローでも評価することを考えている．また，実環境上で動的 / ハイブリッドスケジューラを評価することも今後の課題である．

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山田俊行講師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] 大塚 保紀 他. タスク並列スクリプト言語 megascript の構想. In **先進的計算基盤システムシンポジウム** *SACSIS2003*, pages 73–76, 2003.

[2] 阪口 裕輔 他. タスク並列スクリプト言語処理系におけるユーザレベル機能拡張機構. **情報処理学会論文誌 コンピューティングシステム**, 47(SIG 12):296–307, 2006.

[3] Yukihiro Matsumoto. *Ruby in a Nutshell*. O'Reilly Media, Inc., 2001.

[4] Dave Thomas et. al. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley Professional, 2000.

[5] 須田礼仁. ヘテロ並列計算環境のためのタスクスケジューリング手法のサーベイ. **情報処理学会論文誌 コンピューティングシステム**, 47(SIG 18):92–114, 2006.

[6] 松本真樹 他. 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**, 2(1):1–17, 2009.

[7] 松本真樹 他. 静的情報を用いた動的再スケジューリングのオーバーヘッド削減手法. **情報処理学会論文誌 プログラミング**, 4(3):55–67, 2011.

[8] E.Deelman et. al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *13th IOS Press Scientific Programming*, pages 219–237, 2005.

[9] Y.Zhao et. al. Swift: Fast, reliable, loosely coupled parallel computation. *IEEE International Workshop on Scientific Workflows*, 2007.

[10] http://research.cs.wisc.edu/condor/dagman/. Condor High Throughput Computing.

[11] http://research.cs.wisc.edu/condor/overview. Condor High Throughput Computing.

[12] G.Laszewski et. al. Java cog kit workflow. *Workflows for eScience*, pages 340–356, 2007.

[13] H.Topcuoglu et. al. A high-performance mapping algorithm for heterogeneous computing system. *IPPS/SPDP Workshop on Heterogeneous Computing*, pages 3–14, 1999.

[14] R.Armstrong et. al. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. *7th IEEE Heterogenous Computing Workshop*, pages 87–97, 1998.

[15] Min-You Wu et. al. Segmented min-min: a static mapping algorithm for meta-tasks on heterogeneous computing systems. In *Heterogeneous Computing Workshop, 2000*, pages 375–385, 2000.

[16] Rob V. van Nieuwpoort et. al. Efficient load balancing for wide-area divide-and-conquer applications. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and Practices of Parallel Programming(PPoPP'01)*, pages 34–43, 2001.

[17] S. Bharathi et. al. Characterization of scientific workflows. In *Proceedings of the 3rd Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, 2008.

```
01.  class Interface
02.    def inputEvent(event)
03.      if event is init
04.        init()
05.      else if event is taskFinished
06.        taskFinished(host)
07.      else if event is hostPerformanceChanged
08.        hostPerformanceChanged(hostID)
09.      else if event is hostIdle
10.        hostIdle(hostID)
.            .
.            .
.            .
21.      end
22.    end
23.    def init()
24.    end
25.    def taskFinished(host)
26.    end
27.    def hostPerformanceChanged(hostID)
28.    end
29.    def hostIdle(hostID)
30.    end
.            .
.            .
.            .
41.  end
```

図 4.4: Interface クラスの疑似コード

```
01. class HybridHEFTScheduler < SchedulerLib::Interface
02.   def initialize()
03.     @scheduling_algorithm
            = new HEFTStrategy()
04.   end
05.   def init()
06.     @scheduling_algorithm.schedule()
07.   end
08.   def hostPerformanceChanged(hostID)
09.     @scheduling_algorithm.schedule()
10.   end
11. end
```

図 4.5: HybridHEFTScheduler のコード

```
01.   class Task
02.    @computation_cost = 100              #タスクの計算量
03.    @args = Array.new                  #タスクへの引数
04.    @stdins = Array.new      #タスクの入力側ストリーム群
05.    @stdouts = Array.new      #タスクの出力側ストリーム群
 .       .
 .       .
 .       .
30.   end
```

図 4.6: 処理系内の Task クラスの定義

```
01.  class Task
02.   @rank = 0
03.    def setRank(r)
04.      @rank = r
05.    end
06.   end
```

図 4.7: Task クラス動的拡張の例

```
01.  class SchedulerLib::SchedInfo
02.   @rank = 0
03.    def setRank(r)
04.      @rank = r
05.    end
06.   end
```

図 4.8: SchedInfo クラスの拡張例

```
01. class HEFTStrategy
02.   def initialize()
03.     initialize all queues                #全てのキューを初期化する
04.     @schedule_queue=master_host.schedule_queue
05.   end
06.   def schedule
07.     @schedule_queue.each{|obj|
08.       if obj.class  == Task
09.         @task_queue.push(obj)
10.           if obj.stdouts.size == 0      #あるタスクの出力側ス
                                            #トリームがない場合
11.             @bottomtask_queue.push(obj)  #そのタスクは最下
                                            #位のタスクである
12.         end
13.       elsif obj.class  == Stream
14.         @stream_queue.push(obj)
15.       end
16.     }
17.     @bottomtask_queue.each{|task1|
18.       task1.stdins.each{|stream|    #入力側ストリーム群検索
19.         stream.in_tasks.each{|task2|   #入力タスク群検索
20.           computeCost(task2)
21.           @predtask_queue.push(task2)  #そのタスクは先行
                                            #タスクである
22.         }
23.       }
24.     }
25.     @bottomtask_queue = @predtask_queue
26.     @predtask_queue = []
27.     Sort in descending order of task cost
28.     Mapping(all tasks)
29.   end
30. end
```

図 4.9: 本ライブラリ不使用の HEFTStrategy クラスコード

```
01. class HEFTStrategy < SchedulerLib::DAGStrategy
02.   def schedule
03.  @bottomtask_queue = getBottomTasks()
04.     @bottomtask_queue.each{|task1|
05.       @predtask_queue = pred(task1)
06.       @predtask_queue.each{|task2|
07.          computeCost(task2)
08.       }
09.     }
10.    @bottomtask_queue = @predtask_queue
11.     @predtask_queue = []
12.     Sort in descending order of task cost
13.     Mapping(all tasks)
14.   end
15. end
```

図 4.10: 本ライブラリ使用の HEFTStrategy クラスコード