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

阪口裕輔[†,☆] 大野和彦[†] 佐々木敬泰[†]
近藤利夫[†] 中島浩[††,☆☆]

我々はメガスケールの並列処理を想定したタスク並列スクリプト言語 MegaScript を開発している．MegaScript は逐次/並列の外部プログラムをタスクとして扱い，これらを並列実行することにより大規模並列性を引き出す．このような実行モデルにおいて，高性能化のためには処理系やタスクプログラムを，対象とするアプリケーションや計算環境に特化する必要がある．一方で，記述の容易さやコードの再利用性を高めるには，計算環境に依存せず，タスク間の独立性を確保することが望ましい．本稿ではこれらを両立させるべく，ユーザレベルでの言語機能の拡張を可能とする枠組みとしてアダプタ機構を提案する．アダプタには，実行システムに特化した拡張用コードを処理系やタスクから独立した形で記述でき，機能の追加や動作の最適化など実行ごとのカスタマイズを簡潔に行える．本機構を MegaScript 処理系へ組み込み，評価を行った結果，高い記述性を保ちながら実用上十分な性能が得られることを確認した．

A User-level Extension Scheme for a Task Parallel Script Language

Yusuke Sakaguchi,[†,☆] Kazuhiko Ohno,[†] Takahiro Sasaki,[†]
Toshio Kondo[†] and Hiroshi Nakashima[††,☆☆]

We are developing a task-parallel script language named *MegaScript* for mega-scale parallel processing. MegaScript regards existing sequential/parallel programs as tasks, and controls them for massively parallel execution. Although MegaScript runtime and task programs should be specialized to the target application and platform to obtain high performance, it is undesirable for portability and reusability. To satisfy these conflicting requirements, we propose a user-level runtime extension scheme named *Adapter*. This scheme enables programmers to extend and optimize behavior of the application without modifying the runtime nor task programs. The evaluation of our implementation achieved both efficient programming and enough performance for practical use.

1. はじめに

ゲノム解析や気象・災害シミュレーションなど膨大な計算処理を要する分野では PFLOPS 以上の計算能力が期待されており，そのため 100 万台規模のプロセッサを用いた並列処理が不可欠である．そこで，我々は汎用メガスケールコンピューティング向けのプログラミング言語としてタスク並列スクリプト言語 MegaScript を提案し[1)]，開発を行っている．

メガスケールの並列性を持つプログラムを，MPI のようにフラットで直接的なプログラミングにより記述するのは非常に困難である．このため MegaScript では，部分問題を対象とする逐次/小規模並列プログラム（タスク）を組み合わせることにより大規模並列性を引き出す 2 階層並列モデルを採用している．並列実行のレベルを 2 階層に分離することにより，メガスケールの並列性の記述を現実的なものとする．

MegaScript の実行モデルでは，アプリケーションの特性は組み合わせるタスクの性質により大きく変化する．また，計算環境は Grid と同様に広域に分散したヘテロなものとなり，利用できる環境ごとにその構成や性能は異なる．これにより，実行時に求められる機能や性能はアプリケーションや計算環境などの実行システムごとに変化し，処理の高性能化には個々のアプリケーションや計算環境に特化した最適化が必要となる．しかし，MegaScript による大規模並列アプリケーションの構築には，利用できる計算機資源の変化

† 三重大学大学院工学研究科情報工学専攻
Mie University
†† 豊橋技術科学大学
Toyohashi University of Technology
☆ 現在，ソニーイーエムシーエス株式会社
Presently with Sony EMCS Corporation
☆☆ 現在，京都大学
Presently with Kyoto University

への対応や 2 階層並列モデルに基づくプログラミングスタイルの観点から，計算環境に依存せず，各タスク間の独立性を保つことが重要である．そのため，アプリケーションや計算環境に依存するようなコードを処理系やタスク内部に直接記述することは，応用プログラムの作成を困難にし，コードの再利用性やシステムの可搬性を低下させる．

そこで本稿では，MegaScript 処理系においてユーザレベルでの機能拡張をサポートする枠組みとしてアダプタ機構を提案する．アダプタにはユーザが任意の拡張用コードを記述でき，これを計算環境の各ローカルホスト上でイベント発生に対しコールバックさせることで処理の挙動を変更する．これにより，ユーザは処理系やタスク内部の詳細を知らなくても，新たな機能の追加や既存機能の動作の最適化のためのコードを簡潔に記述できる．具体的には，アプリケーションに特化した例外処理機構や分散ファイルシステムといった機能の追加，あるいは通信処理など実行システムの構成に左右されやすい機能の最適化などをアダプタを用いて実現できる．

以下，まず 2 章で研究の背景となる MegaScript の概要を述べる．続いて 3 章では今回提案するアダプタ機構の概要と詳細構造を示し，4 章でその実装方法について述べる．5 章では実装したアダプタ機構の評価結果を示し，6 章で関連研究との比較を行う．最後に 7 章でまとめる．

2. タスク並列スクリプト言語 MegaScript

2.1 言語の概要

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

計算のコアな部分は外部プログラムとして用意するため，MegaScript プログラム内には主に並列実行に関する制御情報を記述する．実行制御に要する計算量は全体に対してわずかであるため，実行効率より記述性や拡張性を優先し Ruby [2] をベースとするオブジェクト指向スクリプト言語としている．

2.1.1 タ ス ク

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

2.1.2 ストリーム

ストリームは，あるタスクの標準出力の内容を他のタスクの標準入力に流し込むための通信路であり，MegaScript におけるタスク間通信を実現する．

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

2.2 API クラス

MegaScript では，タスクやストリームなど並列実行の基盤となる要素を組み込みの API クラスとして実現している．現在の MegaScript では，以下の 4 クラスが提供される．

`Task` タスクを表現・操作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引数など実行に必要な情報を保持し，生成用メソッドを提供する．

`Stream` ストリームを表現・操作するためのクラスである．ストリーム入出力端への接続タスクの情報を保持し，接続・生成用メソッドを提供する．

`Host` 抽象化された実行環境の各ホストを表現するためのクラスである．実行時に自ホストの情報を収集し，専用のメソッドを介して提供する．

`Scheduler` スケジューラを実現するためのベースクラスである．このクラスを継承してメソッドをオーバライドすることで，任意のスケジューリング戦略を実現できる．

2.3 プログラミングモデル

MegaScript では，外部プログラムであるタスクを並行並列に実行し，その間をストリームで接続しあう，タスクネットワークの形状を記述する．例として，生成者/消費者型のアプリケーションは **図 1** に示すようなタスクネットワークとして表現でき，プログラムは **図 2** のように記述できる．また，オブジェクト指向型の言語であることを利用して，API クラスを継承し，メンバ変数の追加やメソッドのオーバライドを行うことで，自由に変更・拡張することができる．

タスクプログラム内における通信処理は標準入出力ライブラリの関数を使用することで記述でき，プログラムの再利用性や記述の容易性を高める．また，タスクネットワーク構築をサポートするため，機能レベル

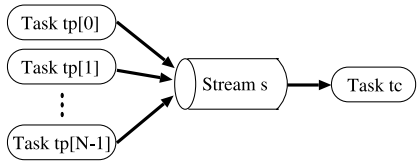


図 1 タスクネットワークの例
Fig. 1 Example of task network.

```
tp = Task.new_array(N, "producer")
tc = Task.new("consumer")
s = Stream.new
s.connect(tp, IN)
s.connect(tc, OUT)
tp.create(1..N)
tc.create
s.create
Scheduler.new.schedule
```

図 2 タスクネットワークの定義例
Fig. 2 Example of task network definition.

ごとに階層化されたクラスライブラリが提供され，並列の知識のないエンドユーザからヘビーユーザまで幅広い層に対応させている[3]．

2.4 ランタイムシステム

ランタイムは API クラス定義などの MegaScript モジュールを組み込んだ Ruby インタプリタとスケジューラからなり，基本的なタスク並列実行機能を提供する[4]．

実行環境では，1 台のホストがマスタ，残りがスレーブとなり，それぞれの上でランタイムプロセスが起動する．マスタはインタプリタ上で MegaScript プログラムを実行し，タスクやストリームなど API クラスのオブジェクトを生成する．スケジューラはインタプリタから適時呼び出され，生成したオブジェクトの配置ホストを決定し，そのホスト上のランタイムプロセスに対して実体の生成を要求する．これに基づき，各スレーブホスト上ではタスクプロセスの生成や通信路の確立など並列実行に必要な処理が開始される．将来的に広域分散環境での実行に対応するため，ランタイムプロセス間の通信には Phoenix[5] を用いている．

3. アダプタ機構

3.1 基 本 概 念

アダプタ機構はアプリケーションや環境依存のコード（アダプタ）を動的に MegaScript 処理系へ組み込むための共通のフレームワークであり，柔軟で拡張性の高い MegaScript の言語機能を実現できる．

ユーザは拡張用のコードをアダプタとして記述でき，これを MegaScript プログラムと組み合わせることでアプリケーションの挙動を拡張できる．ユーザはアダプタを利用することにより，処理系やタスクプログラムを変更せずに，実行システムに特化した機能の拡張や最適化のためのカスタマイズを行える．

3.2 アダプタ機構の設計

アダプタ機構では，以下を実現する必要がある．

(1) アプリケーションや環境依存のコード（アダプタ）を簡単に記述できる．
(2) アダプタを動的に処理系に組み込める．
(3) MegaScript プログラムの実行中に，必要なタイミングでこのアダプタのコードが呼び出される．

(1) について，MegaScript プログラムを作成するユーザは通常，処理系内部の詳細を知る必要がない．また，既存プログラムの外部仕様を把握していればタスクとして利用できるため，その内部仕様までは把握していない場合もある．このようなユーザがアプリケーションの機能拡張やカスタマイズを行うには，追加・変更したい機能に関するコードのみを記述できる仕組みが必要である．また，アダプタの再利用性や保守性，さらに既存の MegaScript 言語仕様との親和性も重要である．

(2) について，組み込み時に処理系やタスクプログラムの再構築が必要になるようでは，アダプタの開発や利用が非常に煩雑になる．動的な組み込みが可能であれば，容易にアダプタを利用できるだけでなく，実行環境に適したアダプタを自動選択し使用する，といった環境適応性の高いアプリケーションも実現できる．

(3) については，組み込んだアダプタの起動タイミングを MegaScript/タスクプログラムで指定するのは非常に困難であり，処理系が自動的に呼び出す必要がある．さらに，この呼び出しのオーバヘッドが大きければ，実行性能が低下してしまう．

以下，それぞれの設計について詳しく論じる．

3.2.1 アダプタのコード記述

MegaScript のアプリケーションは種類の異なる複数のタスクから構成されており，拡張時にこれらのタスクを統一的に扱えるような仕組みが求められる．ここで，MegaScript の実行単位であるタスクはそれぞれが独立した外部プログラムであり，拡張用コードをあらかじめ埋め込んでおくことは困難である．そのため，処理実行時に必要に応じて拡張用コードを割り込ませるような手法を考える．

MegaScript では，タスクの実行や通信など並列実行に関する処理をランタイムにより制御している．そのため，ランタイムはプロセスの生成や通信の発生など大まかな処理の流れを把握することができる．そこ

で，このような処理の流れの変化をイベントとしてとらえ，イベント発生時にランタイム上で対応するアダプタを実行することにより，拡張コードを割り込ませる仕組みをとる．これによりユーザは，あるイベントに対して変えたい挙動の内容だけを記述すればよく，処理系の他の部分やタスクの内部を意識する必要がない．また，拡張コードはランタイム側で実行されるため，タスクの記述言語を問わず 1 種類の言語で書ける．

この拡張コードの記述言語として，様々なアプリケーションや計算環境に柔軟に対応できるよう記述性や拡張性を優先して Ruby を選択した．また，同じ言語をベースとする MegaScript のプログラム内でアダプタを記述するようにした．これにより，アダプタ機構は MegaScript の言語仕様の自然な拡張になっている．

また，様々な状況でのアダプタの利用が想定されるため，アダプタ自体の拡張性を高めたり，より高度な機能を記述できたりすることが求められる．そのため，アダプタの構造はクラスベースとし，ユーザは任意のメソッドを追加したりメンバとして内部状態を持たせたりできるようにした．アダプタをクラスとして提供することで，継承による機能の追加や一部機能の書き換えなどが容易にでき，アダプタの記述性や拡張性を高める．さらに，ランタイムの内部機能を利用できるような専用の API を提供し，タスクやストリームの操作などランタイム内部でしかできなかった処理も記述可能としている．

3.2.2 処理系への組み込み

アダプタの基本機能はベースクラスとして処理系に組み込んでおき，ユーザはこれを継承して拡張コードをスクリプト言語で記述する．これにより，イベント発生時の拡張コード起動といった基本機能は処理系内部に効率良く実装できる．一方で，ユーザが定義するアダプタごとの機能はスクリプトの一部として動的に組み込まれ，処理系を再構築する必要はない．このため，アダプタ作成時にトライ&エラーを繰り返すのも容易である．

3.2.3 アダプタの実行

ランタイムでは通信の発生やプロセスの終了などランタイムにより検知できる動作の変化をイベントの発生としてとらえ，アダプタコードを実行する．このようにイベント発生時に対応するコードを実行する方式としては，`try`〜`catch` 型の例外処理ブロックを記述する方式や，コールバックを用いる方式などがある．

前者は同種イベントであっても文脈に応じて異なるブロックで例外を処理できるため，プロセスの状態に応じた処理を行いやすい．このため C++ をはじめとする逐次言語で広く使われており，Ruby にも同種の機能が用意されている．しかし MegaScript でのプロセス生成や通信発生などのイベントはスケジューラやタスクにより引き起こされ MegaScript プログラムの文脈とは非同期に発生する．このため，このモデルでは扱いにくい．また，分散環境中の事象をマスタホスト上の MegaScript プログラムにおけるイベントとして扱うには，イベント発生ごとにホスト間通信が必要であり，ホスト数増加とともにオーバヘッドが非常に大きくなる．

一方，コールバックではあらかじめ登録しておいたコードがイベント発生時に呼び出されるため，同種イベントに対して文脈に応じた処理を行うことが難しい．しかしアダプタで考慮すべきなのは MegaScript プログラムではなくホストやタスクの状態であるため，問題とはならない．また，多くのアダプタでは自身のローカル変数や特定のタスク，ホストなどの状態のみを参照し，大域的な情報は参照しないと考えられる．この場合，ホスト内でコールバック処理を行うことで，ホスト間通信が不要となる．そこで我々は，イベントが発生したホスト上でローカルなコールバックを行う方式を採用した．

コールバックに指定できるイベントは，通信の発生やプロセスの終了以外にも様々なものを用意し，ユーザが実現する機能に応じて選択できるようにする．ユーザはこれらのイベントに対してアダプタを登録することで，イベント発生時にアダプタがコールバックされ，処理の内容を拡張できる．アダプタの登録されていないイベントに対しては通常どおりの処理が行われる．

MegaScript による並列実行では，分散配置された各ホストに対しタスクやストリームのオブジェクトを送信する形で処理が開始される．そこで，アダプタは登録先のイベントに関連するタスクやストリームが配置されるホスト上に移動し，そのホスト内のランタイムの管理下でコールバックの制御を行う方式をとる．また，動的スケジューリングなどオブジェクトの再配置が行われる際にも，関連するタスクやストリームと共通のホスト上へ情報を保ったまま再移動する．このように，アダプタの管理・制御をローカルのランタイムに委譲することで，効率の良いアダプタのコールバックが可能となり，ユーザはランタイムが提供する機能をローカル環境の処理状況に応じて強化できる．

3.3 Adapter クラス

アダプタの実体は，`Adapter` クラスをもとに生成されるオブジェクトとして実現する．このアダプタオブジェクトを，タスクやストリームなど API クラスが

保持するメンバへ登録することで，ランタイムからのコールバック機能がサポートされる．

Adapter はアダプタを表現・操作するためのベースクラスであり，1 オブジェクトが 1 個のアダプタに対応する．主なメソッドを以下にあげる．

- new(*arg1*, ...)
- callback(*arg1*, ...)

new は，引数として渡された値を使用してアダプタのオブジェクトを新たに生成する．callback はランタイムから任意のタイミングでコールバックされるメソッドであり，このとき登録先のイベントに応じた値が引数 *arg1*, ... として渡される．また，ユーザが callback メソッドから返した返り値は，ランタイムにより登録したイベントに応じて解釈され，その挙動に反映される．たとえば，タスクによるメッセージ送信をイベントとするコールバックの場合，タスクが出力したメッセージを引数としてアダプタの callback が呼び出される．ユーザがメッセージに対して何らかの処理を加えた後，その結果を返り値として返すことで，アダプタが加工したメッセージがストリーム上に流れるようになる．

3.4 アダプタのイベントへの登録

アダプタのコールバックでは，ランタイムから捕捉可能な通信やタスクの終了などをイベントの発生とし，これを起動のタイミングとして利用する．発生するイベントはタスクやストリームなど API クラスごとに変化する．また，イベントの発生は各 API クラスのオブジェクト単位で検出でき，それぞれ個別にアダプタを登録することができる．以下に，アダプタを登録可能なイベントの例をあげる．

- Task
  - メッセージの入出力
  - プロセスの正常/異常終了
  - ファイルアクセス
  - シグナルの送信
- Stream
  - メッセージの中継
  - 通信エラー
  - 動的なタスク接続
- Host
  - 計算機資源の使用率変化
  - システムメッセージの受信
  - 指定時間の経過
- Scheduler
  - 動的スケジューリングの発生

タスクなどの API クラスでは，上記に示したイベントごとにアダプタを登録できるようなメンバを保持し，1 オブジェクトに対してイベントの個数分だけアダプタを登録することができる．

```
class MyAdapter < Adapter
  def callback(msg)
    if msg =~ /^log/
      return nil
    else
      return msg
    end
  end
end
tw = Task.new("worker")
tw.event_output = MyAdapter.new
```

図 3 アダプタの例
Fig. 3 Example of Adapter.

これらのイベントに対するアダプタの利用方法としては以下のものがあげられる．

通信の最適化 メッセージのフィルタリングやまとめ送り，宛先制御など

タスクの操作 タスクの再実行や生成タイミングの制御など

利用資源の共有 ローカルファイルの他ホスト上への転送など

3.5 アダプタのプログラミング

アダプタの利用は，従来の MegaScript プログラミング方法に対して，アダプタの定義と生成，該当するメンバへの登録処理を加えることで可能となる．タスクネットワークの構築方法に変更はなく，今までと同様の形式で記述できる．

まず，利用するアダプタを定義するため，Adapter を継承し，callback メソッドをオーバライドする形で任意のコードを記述する．記述には Ruby 組み込みクラスのメソッドやアダプタ用の API を利用することができ，パターンマッチングによる文字列処理やタスクの操作などを簡潔に表現できる．こうして定義したアダプタクラスのオブジェクトを生成し，任意のイベントに対応するメンバへ代入することでアダプタの登録が行われる．

アダプタの登録はクラス定義時かオブジェクト生成後のいずれかで行える．クラス定義時にアダプタを登録した場合，そのクラスから生成されるオブジェクトはすべてアダプタが登録された状態となる．オブジェクト生成後に個別にアダプタを登録する場合は，特定のオブジェクトのみ挙動を変更するといった記述も可能となる．

図 3 にメッセージのフィルタリングを行うアダプタの記述例を示す．タスク tw のメッセージ出力イベント

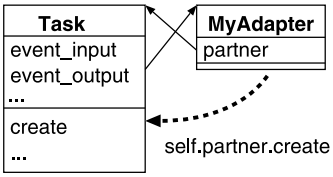


図 4 登録先オブジェクトの操作
Fig. 4 Access from Adapter.

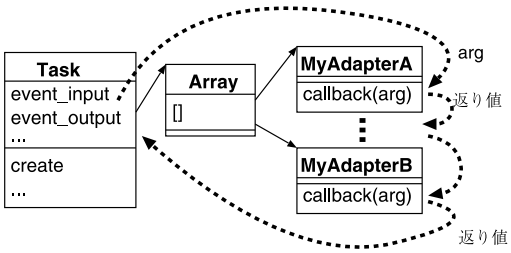


図 5 アダプタチェーン
Fig. 5 Adapter chain.

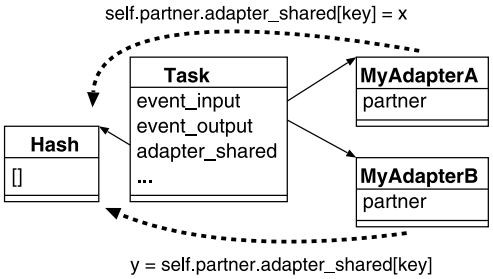


図 6 アダプタ間の共有変数
Fig. 6 Shared variables between Adapters.

`event_output` に `MyAdapter` クラスのオブジェクトを登録することで，`worker` の出力に対し，`callback` メソッドが呼び出される．この結果，文字列 “log” を先頭に持つメッセージはログ情報であるとして破棄し，それ以外のメッセージはストリームに送られる．

3.6 アダプタの記述をサポートする拡張機能

ユーザが記述する `callback` メソッドでは，引数や返り値を利用してランタイムと情報を受け渡し，処理系の挙動を操作する．しかし，より複雑な操作を記述する場合には，ランタイムの内部機能を直接呼び出して利用する必要がある．また，複数のアダプタを組み合わせたり連携させたりする機能があれば，より高度なアダプタを実現できる．

このような目的のために，専用の API をアダプタクラスのメソッドとして提供している．以下で主要な機能について述べる．

3.6.1 登録先オブジェクトの操作

タスクやストリームなどランタイム内部で管理されているオブジェクトをアダプタ側から操作できるよう，組み込みのメソッドとして `partner` を提供している．

`partner` はアダプタの登録先である API クラスのオブジェクトを指し示すポインタであり，アダプタ側からはアクセサとして利用することができる．このメソッドを用いることで，登録先のオブジェクトに対する操作や各種情報の取得などを，ランタイム内部を隠蔽した状態でアダプタ側から行えるようになる．具体的には，タスクの再実行やプロセス生成タイミングの制御などの機能をアダプタとして実現できる（**図 4**）．

3.6.2 アダプタチェーン

ある 1 つのイベントに対して複数のアダプタを連続して呼び出せるよう，アダプタの配列オブジェクトを登録できるようにしている．

イベントの発生時には，配列の先頭のアダプタからチェーン式にコールバックされ，引数には 1 つ前のアダプタの返り値が順次伝搬する（**図 5**）．この機能を利用することで，既存のアダプタを組み合わせて利用したり，複雑な機能を複数の単純なアダプタの集合として記述したりできる．

3.6.3 アダプタ間の情報共有と連携

同一のオブジェクトに登録されるアダプタ間で情報を共有できるよう，登録先のオブジェクト内部で専用のハッシュを用意している．このハッシュはアダプタから `partner` と変数へのアクセサ `adapter_shared` を介することで参照でき，異なる型からなる複数のオブジェクトを区別した状態で共有できる（**図 6**）．この機能を用いることにより，単独での処理だけでなく，複数のイベントのアダプタを連動させるような複雑な処理も記述できるようになる．

また，異なるオブジェクトに登録されたアダプタどうしでも情報のやりとりや処理の連動を行えるよう，関連付けによるアダプタのリモートコール機能を提供している．あるアダプタ a_1 に対し，`Adapter` クラスの `relate` メソッドを呼び出すことで，連動させたい別のアダプタ a_2 を関連付けすることができる．その後，a_1 の `callback` メソッド内でリモートコール用のメソッド `related_call` を呼び出すと，a_2 の `callback` メソッドが呼び出される．この機能では異なるホスト上のアダプタに対しても，その配置先を意識することなく呼び出すことができる．このため，アダプタを介した複数のタスクの協調処理なども容易に記述できる．

4. 実　　装

4.1 アダプタ機構の全体像

アダプタ機構による処理の流れは以下のようになる．

(1) アダプタの登録
MegaScript プログラムの実行により，生成されたアダプタがタスクやストリームなどのメンバに登録される．

(2) 配置先ホストへの送信
スケジューラが呼び出されると，アダプタは登録先のオブジェクトとともに，配置先であるホストへ転送される．

(3) イベントの監視
配置先ホストのランタイムはアダプタを受信後，対応するイベントの発生を監視する．

(4) コールバック
ランタイムは，該当するイベントの発生を検出すると，規定の引数を用いてアダプタをコールバックする．
アダプタは `callback` メソッド内で明示的に削除されるか，登録先オブジェクトが消滅するまで存在する．このアダプタの生存期間の間，(3)，(4) を繰り返す．

これらの処理を実現するため MegaScript の処理系を拡張した．

4.2 API クラスの拡張

アダプタを登録できるよう MegaScript の API クラスを拡張した．各 API クラスにおいて，想定されるイベントごとにアダプタを登録できるようインスタンス変数を追加した．この変数には，`Adapter` またはそれを継承したクラスのオブジェクトを，単体もしくは配列の形で代入できる．

また，アダプタのベースクラスとなる `Adapter` を `Task` や `Stream` と同様に API クラスの 1 要素として MegaScript モジュール内に組み込んだ．アダプタのオブジェクトは各ローカルホストに転送されるため，保持している情報をシリアライズする機能を持たせた．以上により，プログラム作成時にアダプタの表現・操作を可能とした．

4.3 アダプタ起動部

アダプタ機構により行われる主要な処理は，イベントの監視とイベント発生時の `callback` メソッドの呼び出しである．タスクやストリームが持つ機能はランタイムにより制御されているため，ランタイムを拡張する形でアダプタ機構に必要な処理を組み込んだ．

ランタイムは，タスクプロセスの管理やメッセージの送受信などを並行して行えるよう，複数のスレッドにより構成されている．これらのスレッド内で該当するイベントの制御を行っている部分に，イベント検出用のコードを追加した．イベントを検出すると，これを引き起こしたオブジェクトを特定し，対応するアダプタが存在すればコールバックの処理を行う．

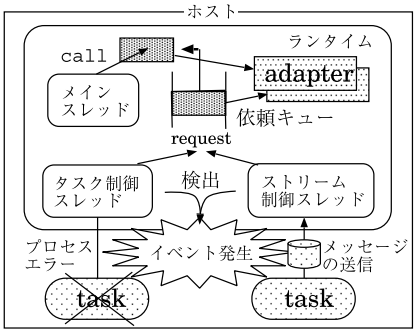


図 7 アダプタのコールバックモデル
Fig. 7 Model of Adapter callback.

しかし，現在の Ruby の実装では複数のスレッドからインタプリタを呼び出すことができない．そこで，各スレッドからのコールバック要求を受け付ける「依頼キュー」を導入し，このキューを使って各スレッドにおけるアダプタの呼び出しを実現した．各スレッドはイベントを検出するとアダプタの呼び出しリクエストを依頼キューに送る．メインスレッド上のアダプタ処理部はこのキューを監視し，要求があれば対応するアダプタの `callback` メソッドを呼び出す（**図 7**）．

5. 性 能 評 価

アダプタ機構の有用性を示すため，アダプタの実行時オーバヘッドと記述性について評価を行った．

評価環境は Pentium4 2.8 GHz，メモリ 512 MB のホストをギガビット・イーサネットで接続した PC クラスタであり，OS は Vine Linux 3.2（kernel-2.4.31）を用いた．使用したソフトウェアのバージョンは Phoenix 1.0，Ruby 1.8.4 である．

また，評価用の実アプリケーションとして，2 本の DNA 配列を比較して共通する部分領域を列挙するプログラム[6] を，MegaScript により並列化して使用した．この MegaScript プログラム（以下 `genom`）は N 個の探索タスクと 1 個の集計タスクからなり，探索空間を N 分割して並列処理を行う．各探索タスクは発見した共通部分領域を随時出力し，これらが 1 本のストリームにより非同期に集計タスクへと送られる．

5.1 実行時オーバヘッド

アダプタ機構により発生するオーバヘッドは 2 種類

表 1 ランタイムの初期化・終了処理時間
Table 1 Time for initialization and finalization.

ホスト数	2	4	8	16	32
Org (秒)	6.316	6.668	10.695	13.504	16.737
Ada (秒)	6.322	6.689	11.089	14.001	17.153
対 Org 比	1.001	1.003	1.037	1.037	1.025

表 2 genom の実行時間
Table 2 Execution time of genom.

ホスト数	2	4	8	16	32
Org (秒)	760.22	389.31	200.53	111.85	67.01
Ada (秒)	760.29	389.57	200.61	112.31	67.21
対 Org 比	1.000	1.000	1.000	1.004	1.003

あげられる．まず，イベントの検出や呼び出しの判定処理などアダプタ機構を組み込んだことによるオーバヘッドがあり，これはアダプタを利用しない場合にも発生する．次に，アダプタの `callback` メソッドの呼び出しにかかるオーバヘッドがあり，これはアダプタを利用する場合にのみ発生する．

そこでこれらのオーバヘッドを評価するため，まずオリジナル版およびアダプタ機構実装版の 2 種類のランタイムについて，実行時間の比較を行った．続いて，通信ベンチマークおよび実アプリケーションによって，アダプタの利用でどの程度のオーバヘッドが生じるかを評価した．

5.1.1 アダプタ機構組み込みによるオーバヘッド

オリジナル版ランタイム（Org）とアダプタ機構実装版ランタイム（Ada）の双方について，ホスト数を変えながら初期化・終了処理に要する時間を測定した．

結果を**表 1** に示す．アダプタ機構を組み込んだランタイムの初期化・終了処理で発生するオーバヘッドは 3%前後であり，実用上問題ない程度であることが分かった．

次に，アダプタ機構組み込みによるイベント検出・呼び出し判定のオーバヘッドを測定するため，ホスト数を変えながら `genom` をオリジナル版およびアダプタ機構実装版ランタイムの上で実行し，実行時間を測定した．ここで，探索タスク数はホスト数と同数としており，問題サイズはホスト数にかかわらず一定である．

結果を**表 2** に示す．この問題では，共通部分領域送信のため全体で 11,000 回のタスク間通信が発生する．この通信のたびに Ada ではイベントが検出され，アダプタを呼び出すか否かの判定が行われるが，アダプタを使用していないため呼び出しは生じない．これらの処理によるオーバヘッドは最大でも 0.4%であり，実用上無視できるといえる．

表 3 アダプタの呼び出しオーバヘッド
Table 3 Overhead of calling Adapter method.

Type	none	output	relay	input	rewrite
対 none 比	1	1.02	1.03	1.02	1.02

5.1.2 アダプタ呼び出しによるオーバヘッド

アダプタを呼び出すことにより発生するオーバヘッドを，発生頻度の高い通信処理イベントにより測定した．ベンチマークプログラムには，生成者/消費者型のモデルで 2 タスク間の単方向通信を繰り返すプログラムを使用し，以下のそれぞれの場合について実行時間を測定した．ランタイムはいずれもアダプタ機構実装版である．

none アダプタを使用しない．

output，relay，input それぞれタスク出力時，ストリームによる中継時，タスクへの入力時に，空の（なにもしない）アダプタを呼び出す．

rewrite タスク出力時にアダプタ内で Ruby の組み込みメソッドを呼び，メッセージを書き換える．

アダプタを使用しない場合（none）に対する時間比を**表 3** に示す．アダプタ呼び出しのオーバヘッドは，多くて 3%程度である．このベンチマークは通信のみを繰り返すものであり，実アプリケーションにおける通信頻度はこれより大幅に小さい．したがって，オーバヘッドもさらに小さくなることが期待できる．

5.1.3 実アプリケーションでのオーバヘッド

実アプリケーションでアダプタを使用する際のオーバヘッドを評価するため，`genom` に対して性能改善のためのチューニングを施した．

`genom` の探索タスクに用いているプログラムは，もともと人間が読める形で結果を出力するようになっている．このため，各探索タスクの出力行のうち解データを含むのは半数であり，残りは集計タスクにとって不要な行である．したがって，表 2 に示した実行時間は余分な通信時間を含んでおり，不要な行の送信を抑制することで通信量が半減し，性能が改善できる．そこで，以下の 2 通りの方法で不要行送信を抑制した．

`task` 探索タスクプログラムのソースを修正し，不要行を出力する部分を削除した．

`filter` 3.5 節で示したような，各メッセージにパターンマッチングを行って不要行を破棄するフィルタリングアダプタを作成し，タスクのメッセージ出力イベント（`event_output`）に付加した．

これらのチューニングされた `genom` を用いて，5.1.1 項と同じ条件で実行時間を測定した．

結果を**表 4** に示す．今回の条件では計算時間に対し

表 4 チューニングされた genom の実行時間
Table 4 Execution time of tuned genom.

ホスト数	2	4	8	16	32
task (秒)	758.25	385.96	198.61	106.20	65.00
対 Ada 比	0.9973	0.9907	0.9900	0.9456	0.9671
filter (秒)	759.80	386.64	199.04	106.32	65.33
対 Ada 比	0.9994	0.9925	0.9922	0.9467	0.9720
対 task 比	1.0020	1.0018	1.0022	1.0011	1.0051

表 5 1 ホスト上に複数のタスクがあるときのオーバヘッド
Table 5 Execution time of genom on single host.

タスク数	1	2	8	32
Ada (秒)	279.49	279.68	280.09	283.93
task (秒)	278.85	278.99	279.49	281.32
対 Ada 比	0.9977	0.9975	0.9979	0.9908
filter (秒)	279.42	279.58	280.05	281.82
対 Ada 比	0.9997	0.9996	0.9999	0.9926
対 task 比	1.0020	1.0021	1.0020	1.0018

通信量が少ないためチューニングの効果は小さいが，task，filter ともにホスト数増加につれて改善が見られ，32 並列で数パーセントの速度向上を得ている．また，前者に対する後者の実行時間比では，0.1〜0.5%の増加にとどまっている．この増加分はチューニングをアダプタで行うことによるオーバヘッドと見なせるが，実用上，無視できる大きさといえる．

次に，同一ホスト上で複数のタスクにより非同期にアダプタ呼び出しが発生する場合のオーバヘッドを評価するため，1 ホスト上で genom を実行した．問題サイズは固定し☆，探索タスク数を 1〜32 で変化させた．

結果を **表 5** に示す．チューニングを行わない場合（Ada），計算量が一定であるためタスク数 1 のときが最も効率が良く，タスク数の増加につれて並行実行によるオーバヘッドが増加している．これに対し，タスクプログラム修正によるチューニング（task）では，ホスト内のストリーム通信（プロセス間通信）の回数が半減するため，32 タスク時に 0.9%程度の速度改善が見られる．一方，フィルタリングアダプタによるチューニング（filter）でも同様な改善が見られ，task に対するオーバヘッドはタスク数と無関係に 0.2%程度であった．したがって，アダプタ利用によるオーバヘッドは数十タスク程度までであればタスク数に依存しないと考えられる．タスク数がより多いときにはオーバヘッドが増加する可能性もあるが，その場合はプロセスの並行実行やストリーム通信のオーバヘッドも増加するため，アダプタの利用のみが問題になる状況は考えにくい．また，現在の MegaScript 処理系では，タスク間の依存を考慮したスケジューリングとタスクプロセスの生成タイミングの制御により，1 ホスト上で同時に実行されるタスク数を抑制するようにしている．

☆ 表 2，表 4 の評価で用いた問題の約 1/5 の大きさである．

5.2 アダプタの記述性

5.2.1 チューニング用アダプタの記述性

アプリケーションのチューニングを行う際の記述性を評価するため，5.1.3 項で評価に使用した genom のチューニング作業について考える．

タスクプログラムを修正する方法では，約 1300 行あるソースファイルから該当する個所を見つけだし，余分な出力コードを削除する必要がある．今回のケースではそれほど困難な作業ではないが，タスク内部がブラックボックスでも上位層の MegaScript プログラムを作成できるという 2 階層モデルの観点からは，性能改善のためにタスク内部のコードを書き換える手法は望ましくない．また，出力を受け取るタスクによって入力形式が異なるような場合，そのつど同様の作業を行うのはタスクプログラムの再利用性を低下させる．

一方，フィルタリングアダプタを用いる場合は，図 3 のような 10 行程度のコードを MegaScript プログラムに追加すればよい．このアダプタを作成するにはタスクプログラムの出力内容から不要行のパターンを作成するだけでよく，そのコードまで解析する必要はない．また，出力を受け取るタスクを置き換える場合もアダプタのみ修正すればよく，タスクプログラムは変更しなくてよい．したがって，階層の分離や保守性・再利用性の面から有利であるといえる．

5.2.2 機能を拡張するアダプタの記述性

MegaScript の機能を拡張するアダプタの記述性評価として，プロセスエラー発生時にタスクを再実行する機能を拡張する例を考える．この拡張をランタイム内のコード変更とアダプタの作成の 2 通りの方法で実施し，必要な作業量を比較した．

ランタイムのコード変更を行う場合，記述するコードはエラー発生時の捕捉やエラー確認後のプロセスの再実行など数十行に及んだ．この作業には広範囲にわたるコードの追加・書き換えを要するためランタイムのコード理解が必要であり，ユーザレベルでの機能拡張は非常に困難であると考える．

一方，アダプタを作成した場合は **図 8** のコードで実現できた．異常終了を示す引数を受け取った場合，タスクの実体を指す partner に対し，プロセス起動を命令する API である create メソッドを適用することで，タスクプロセスの起動部分を簡潔に記述できる．この ReStartAdapter オブジェクトを生成し，タスクのプロセス終了イベントである event_terminate に代入することで，エラー発生時に自動的にメソッドが

```
class ReStartAdapter < Adapter
  def callback(arg)
    if arg == false
      self.partner.create
    end
  end
end
task.event_terminate = ReStartAdapter.new
```

図 8 タスク再実行を行うアダプタ
Fig. 8 Task restarting Adapter.

```
class ReStartAdapter2 < ReStartAdapter
 def callback(arg)
  if arg == false
   super(arg)
   self.partner.adapter_shared["msg"].each do
     |msg| self.partner.stdin.print(msg)
   end
  end
 end
end
class CommAdapter < Adapter
 def initialize
  self.partner.adapter_shared["msg"]=Array.new
 end
 def callback(msg)
  self.partner.adapter_shared["msg"].push(msg)
 end
end
task.event_terminate = ReStartAdapter2.new
task.event_input = CommAdapter.new
```

図 9 拡張したタスク再実行アダプタ
Fig. 9 Extension of task restarting Adapter.

呼び出される．以上の結果より，アダプタを利用することで，処理系内部を隠蔽したまま，より少ないコード量で機能を実現することができ，一般ユーザでも利用が容易と考える．

また，アダプタによる記述はランタイムやタスクプログラム内に直接記述する場合と比べ，機能の拡張が容易であり，異なるアプリケーションに対しても柔軟に対応できる．タスク再実行アダプタを継承し，メッセージの整合性を保つよう，前プロセスで受信していたメッセージをプロセス起動後に再度渡すように拡張したアダプタの例を，**図 9** に示す．ここでは，`CommAdapter` と `ReStartAdapter2` の 2 つのアダプタから構成した．`CommAdapter` はメッセージ入力イベント `event_input` へ登録し，受信したメッセージを共有変数 `adapter_shared` に配列として格納することで `ReStartAdapter2` からも参照可能とした．`ReStartAdapter2` では，親クラスの `callback` メソッドを呼び出してタスクを再起動した後，`adapter_shared` を介して取り出したメッセージをタスクの標準入力に順次流し込み，エラー発生前の状態を復元していく．このように，クラス継承や複数アダプタの連携を利用することで，複雑な機能も容易に記述できる．

6. 関連研究

実行システム自身の挙動をアプリケーションプログラム内から操作・変更できる仕組みとしてリフレクションがある[7),8)]．リフレクションはプログラム自体をデータとして扱い，計算の対象とすることで自己変更を行える．MegaScript では，計算タスクはブラックボックスであり，タスク内部の挙動を変更することはできない．アダプタ機構では，並列実行を制御するランタイムレベルで機能拡張や最適化のためのコードを割り込ませる仕組みをとっている．

大規模並列計算に対する既存アプローチとして，UNICORE[9)]，オーガニックジョブコントローラ[10)]などのワークフロー型システムがあげられる．ワークフロー型システムはユーザのジョブを，タスク間の依存関係を表すワークフローとして与える．データの受け渡し時には，システム側でデータ変換処理を行うことでタスクの変更を最小限に抑えることができる．MegaScript ではアダプタを用いることで，タスク間通信時の不整合性や非効率性の解消だけでなく，その他の機能に関しても拡張・最適化できる．

7. おわりに

本稿では，MegaScript 処理系に対するユーザレベルでの機能拡張を可能とするアダプタ機構を提案した．アダプタ機構はユーザが記述したアダプタをアプリケーション実行時にランタイムへと組み込むことにより，言語機能の拡張や最適化をサポートする．

アダプタによる機能の実現は同等の機能を処理系やタスク内部に記述するのに比べ，拡張性が高くタスク間の独立性を確保でき，様々なアプリケーションへ柔軟に対応することができる．また，将来的には計算環境の差違をアダプタにより吸収できるような仕組みも導入していく．基本的なアダプタ起動部を実装し評価を行った結果，アダプタ機構により発生するオーバヘッドは実用上許容できる範囲に抑えられていることが確認できた．

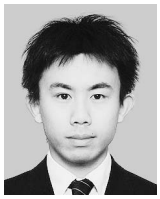
今後は，エラー回復や通信制御など典型的なケースについて，あらかじめ該当する機能を持つアダプタとして用意し，機能レベルに応じた階層型のアダプタライブラリとして構築していく．これにより，ユーザが既存のアダプタから選んで使うか，より高度な物を自作するか選択できる環境を目指す．

謝辞 本研究は，科学技術振興事業団・戦略的基礎研究「低電力化とモデリング技術によるメガスケールコンピューティング」による．

参 考 文 献

1) 大塚保紀，深野佑公，西里一史，大野和彦，中島浩：タスク並列スクリプト言語 MegaScript の構想，先進的計算基盤システムシンポジウム SACSIS2003, pp.73–76 (2003).
2) まつもとゆきひろ，石塚圭樹：オブジェクト指向スクリプト言語 Ruby，ASCII (1999).
3) 西川雄彦，阪口裕輔，田中一毅，大野和彦，中島浩：タスク並列スクリプト言語用アプリケーション層ライブラリの実現，情報処理学会研究報告 2004-HPC-99, pp.13–18 (2004).
4) 西里一史，大野和彦，中島 浩：タスク並列スクリプト言語 MegaScript のランタイムシステムの設計と実装，情報処理学会研究報告 2003-HPC-95, pp.119–124 (2003).
5) Taura, K., Kaneda, K., Endo, T. and Yonezawa, A.: Phoenix: A Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resouces, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP 2003*), pp.216–229 (2003).
6) 松尾 洋：バイオプログラミングバイオインフォマティクス演習，オーム社 (2005).
7) Smith, B.C.: Reflection and Semantics in Lisp, *Proc. 11th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pp.23–35 (1984).
8) Chiba, S.: A Metaobject Protocol for C++, *Proc.ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA'95*), pp.285–299 (1995).
9) Romberg, M.: The UNICORE Architecture — Seamless Access to Distributed Resources, *Proc. 8th IEEE International Symposium on High Performance Distributed Computing* (*HPDC8*), pp.287–293 (1999).
10) 上田晴康，吉田武俊，安里 彰：ジョブ投入と待ち合わせの出来るジョブ制御スクリプト：オーガニックジョブコントローラの試作，情報処理学会研究報告 2003-OS-94, pp.99–106 (2003).

(平成 18 年 1 月 27 日受付)
(平成 18 年 5 月 16 日採録)



阪口 裕輔

2006 年三重大学大学院工学研究科情報工学専攻博士前期課程修了．同年ソニーイーエムシーエス（株）入社．現在，組み込み向けソフトウェアに関する開発に従事．

大野 和彦（正会員）

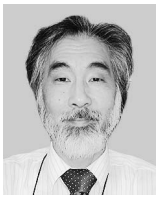
1998 年京都大学大学院工学研究科情報工学専攻博士後期課程修了．同年豊橋技術科学大学助手．2003 年三重大学講師．言語の設計・実装・最適化等並列プログラミング環境に関する研究に従事．博士（工学）．

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．