2005－HPC－103（26）
2005／8／5

タスク並列スクリプト言語処理系における ユーザレベルの機能拡張を可能とする機構

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

グリッド計算など大規模な並列処理においては，実行環境の性能やアプリケーションの特性などその性質は多様であり，処理の進行に与える影響は大きい．そのため，プログラミング言語が提供する機能にこれらの点を考慮できるような仕組みを持たせることは処理の効率性，信頼性を確保する上で重要な要素となる．本稿では，我々が開発しているメガスケールコンピューティング向けの並列言語 MegaScript を対象とし，ユーザレベルでの言語機能の拡張を可能とするアダプタ機構について提案する．本機構は，ユーザが実行時特性を考慮して記述したスクリプトを分散環境の各ローカルホスト上にて，任意のイベント発生時に実行する．これにより，わずかなオーバヘッドの増加で，言語処理系に対し新たな機能の付加や動作の最適化を実現可能とする．

User-Level Extention Scheme for a Task Parallel Script Language System

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

In large-scale parallel processing like grid computing, various characteristics of the applications and the environments largely affect the program behaviour. Therefore in the programming language, providing a mechanism for easy handling of such characteristics is important to increase efficiency and reliability of parallel programs. In this paper, we propose a mechanism called '*adapter*' for our parallel script language MegaScript, which enables user-level extension of this language. This mechanism invokes user's function on the occurrence of specified event. The functions can be written in the script language, and the call-back is localized within the host of the event. This enables easy extension and optimization of our language with small overhead.

1. はじめに

近年，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において Pflops 級の計算能力が期待されている．現在，この性能を得るべく，Grid や 100 万プロセッサ規模によるメガスケールコンピューティングなど様々な研究が進められている．

これらの大規模並列計算では，実行環境として性能の異なるホストやネットワークが混在する広域分散型の環境が想定されている．また，適用するアプリケーションに関しても，計算処理や通信の頻度など処理の挙動は大きく変化する．これにより，実行環境やアプリケーションが変わる毎に並列計算の高性能化に求められる要素が異なる可能性が高い．しかし，このような実行時における特性の違いを自動的に補正し，常に最適な実行を保証することは非常に困難である．そのため，実行環境やアプリケーションの特性を考慮したプログラムを容易に記述できるようにすることは，大規模な環境で並列計算を行う際には重要な要素となる．

MegaScript[1)] とは，我々が開発しているメガスケールコンピューティング向けの並列プログラミング言語である．MegaScript は逐次や並列の外部プログラムを実行単位（タスク）としたタスク並列実行を行うため，処理の挙動は組み合わせるプログラムにより異なる．また，実行には Grid と同様に性能非均質な環境の利用が想定され，実行環境毎に計算機資源の構成が大きく異なる可能性がある．そのため，これらの特性の変化を言語機能側へ反映させ，最適な機能を実現でき

† 三重大学
Mie University
†† 豊橋技術科学大学
Toyohashi University of Technology

るようにすることは性能向上を考える上で必要である．

そこで，本研究では柔軟で拡張性の高い言語機能の実現を目的とし，MegaScript に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導入した．本機構は，ユーザが分散環境内の各ローカルホスト上にて，イベント発生に対しハンドラをフックさせることができ，指定したハンドラコードはローカルホスト上で実行される．

これにより，ユーザが処理系内部の詳細を知らなくても，新たな機能の追加や実行時の特性に応じた既存機能の動作の最適化など，言語機能のカスタマイズを容易に行うことができる．具体的な適用例としては，現在 MegaScript に未実装である例外処理機構や分散ファイルシステムなど実用上必要な機能の追加や，通信処理など実行時の状況に左右されやすい機能の最適化が挙げられ，これらの処理を言語処理系から分離した形で容易に記述することができる．

以下，次章で MegaScript の概要を述べ，3 章では今回提案するアダプタ機構について概説する．4 章および 5 章ではアダプタの詳細構造とその実装方法について述べる．6 章でアダプタの性能評価を示し，最後にまとめとする．

2. タスク並列スクリプト言語 MegaScript

2.1 言語の概要

MegaScript は逐次／並列の外部プログラムを計算タスクとして扱い，複数のタスクを制御して並列実行させるタスク並列言語である．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独立した外部プログラムとして用意されるため，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を優先し Ruby[2] をベースとするスクリプト言語としている．

2.2 タ ス ク

タスクは MegaScript の並列実行単位である．タスクの実体は任意の言語で記述された独立性の高い逐次または並列の外部プログラムであり，内部の処理に MegaScript は関与しない．MegaScript では，このタスクを複数組み合わせてタスクネットワークモデルを構築する．

2.3 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，

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

図 1 タスクネットワークの定義例

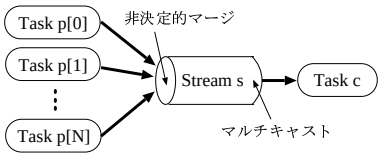

図 2 タスクネットワーク

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，一対多，多対多などの通信を簡潔に実現することができる．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，メッセージはそれらのタスクにマルチキャストされる．

2.4 プログラミングモデル

MegaScript は，並列動作させるタスクや通信路の役割を果たすストリームを表すオブジェクトを定義・生成し，それらを組み合わせたタスクネットワークモデルとしてその形状を記述する（図 1，2）．

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用いることで容易に記述することができる．また，MegaScript ではエンドユーザ向けにタスクネットワーク構築をサポートする階層型のライブラリモジュールを提供する．

2.5 ランタイムシステム

ランタイムは，タスクなどの API クラス定義を組み込んだ 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[3),4)]．

実行環境の各ホスト上では，マスタとスレーブから構成される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のオブジェクトを生成する．スケジューラは各オブジェクトの配置ホストを決定し，そのホスト上のスレーブに対してオブジェクトを送信，実体の生成を要求する．この要求にもとづき，各ローカルホスト上で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．

以下にタスク並列実行機能のベースとなる API クラスについて示す.

- `Task`
  タスクを表現・操作するためのクラスで，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，接続情報を保持し，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を表すクラスで，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ベースクラスで，継承によるオーバライドで任意の戦略を実現可能とする.

以下，これらの API クラスをもとに生成されるオブジェクトを MegaScript オブジェクトと呼ぶ.

3. アダプタの概要

3.1 基本概念

アダプタとは，我々が通信機能の強化を目的とし MegaScript に導入したインタフェース機構[5] の概念を拡張したもので，MegaScript の機能を容易に拡張・最適化する仕組みを提供する.

アダプタは，ユーザが規定した任意の処理を分散環境の各ローカルホスト上でコールバックする機能を提供する．アダプタの起動に用いるイベントは，エラー発生時やメッセージの受信時などユーザにより指定可能である．これにより，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ユーザ自身により強化することができる．アダプタの記述には，実行環境やアプリケーションの変化に容易に対応できるよう，記述性を優先しスクリプトを用いる．アダプタはクラスとして実現され，任意のデータやメソッドを保持できる．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が容易となり，アダプタの記述性や拡張性を高める．また，アダプタからランタイムの機能や情報を利用できるよう専用の adapterAPI を提供する．この API を利用することで処理系内部をブラックボックスとしたまま MegaScript オブジェクトを容易に操作できる.

3.2 動作形態

アダプタの実体は，新たに追加した API クラス `Adapter` からなるオブジェクトとして実現される．このアダプタオブジェクトを `Task` や `Stream` など他の API クラスオブジェクトの属性として登録することで，登録先の MegaScript オブジェクトにおけるイベントの発生を監視し，ランタイムシステムからのコールバックに備える．そのため，アダプタオブジェクトは登録する MegaScript オブジェクトと同じホスト上に配置され，そこで実行される.

前述の通り，アダプタの動作はイベント駆動型であり，ハンドラをフックさせておくことで MegaScript オブジェクトに発生する様々なイベントに対して反応し，コールバックを行う．また，一つの MegaScript オブジェクト上において，エラーの発生やメッセージの受信など複数のイベントの発生が想定されるため，各オブジェクト毎に捕捉可能なイベントの数だけアダプタを登録できる属性を持つ.

4. 詳細構造

4.1 Adapter クラス

アダプタは，`Task` や `Stream` と同様に API クラスの一要素として実現される．`Adapter` クラスには，イベント発生時にコールバックされる `callback` メソッドが用意されている．また，インスタンス変数を追加することで任意のデータを保持できる.

アダプタはこの Adapter をベースとし，フックしたいイベントに対応できるアダプタクラスを実現する．`callback` メソッドには対応するイベントに応じた実行時引数が渡され，これを利用してイベントの制御などを行う．また，クラスベースとしているため，継承による一部機能の書き換えや階層構造の構築などが容易であり，ユーザの記述性や拡張性を高める.

4.2 API クラスの拡張

既存の各 API クラスにおいて，発生するイベント毎に対応する属性を付加し，アダプタオブジェクトの登録を許可するよう拡張した.

以下に，アダプタを登録可能なイベントの一覧を挙げる.

- Task
  - メッセージの送受信
  - プロセスの正常／異常終了
  - ファイルの読み書き
- Stream
  - メッセージの中継
  - 通信エラー
  - 接続タスクの動的な追加
- Host
  - 計算資源の使用率変化
  - 許容タスク数の超過
  - 一定時間の経過
- Scheduler

– 動的スケジューリングの発生

4.3 アダプタの基本機能

アダプタでは，ランタイムの機能拡張を行うために以下のような基本機能を提供する．

4.3.1 ローカルコールバック

登録する MegaScript オブジェクトと共通のホスト上に移動し，各ローカル環境上でのアダプタメソッドのコールバックをサポートする．大域的な例外処理を行う方式と異なり，ホスト間通信を必要とせず効率的な処理が可能である．起動に用いるイベントはユーザにより指定でき，ランタイムによるイベント検知時に自動的に呼び出される．

また，動的スケジューリングによるオブジェクトの再配置時にも，これを捕捉し情報を保ったまま共通のホスト上へ再移動を行う．

4.3.2 adapterAPI

アダプタ内からランタイムの機能や情報を利用できるよう，専用の adapterAPI を提供する．この API は，タスクの実行やシステムメッセージの送信など様々な機能を提供し，本来ランタイムの内部でしか不可能だった処理をスクリプトとして容易に記述することができる．

また，この API を MegaScript オブジェクトに適用できるようにするため，アダプタ側から登録したオブジェクトを参照できるよう組み込みのインスタンス変数として`@partner` を提供する．`@partner` 変数は対応するオブジェクトの実体を指し示し，API のメソッドはこの変数を対象とし利用する．

4.4 アダプタの拡張機能

アダプタでは，より複雑な機能を容易に実現できるよう，以下のような拡張機能を提供する．

4.4.1 アダプタチェーン

MegaScript オブジェクトで発生する任意のイベントに対し，複数のアダプタを連続して起動できるようアダプタの配列オブジェクトを登録可能とする．

イベント発生時には，先頭のアダプタからチェーン式に実行され，その際に渡される引数は順次伝搬される．これにより，複雑な機能を複数の単純なアダプタの集合として実現可能とする．

4.4.2 アダプタ間の情報共有

共通の MegaScript オブジェクトに登録されるアダプタ同士で情報を共有化できる変数`@adapter_shared` を提供する．この変数は，登録する MegaScript オブジェクト上に生成され，アダプタからは`@partner` 変数を介することで参照できる．

この`@adapter_shared` には Ruby のオブジェクトを代入でき，配列やハッシュオブジェクトを用いることで異なる型からなる複数のオブジェクトを共有できる．これにより，単独での処理だけでなく複数のアダプタを連動させた複雑な機能を実現可能とする．

```
class MyAdapter < Adapter
  def callback(msg)
    return msg.gsub(/,/,' ')
  end
end
t = Task.new("worker")
t.out = MyAdapter.new
```

図 3 通信機能を拡張するアダプタ例

4.5 アダプタ・プログラミング

アダプタを作成するには，API クラス `Adapter` を継承し，`callback` メソッドに独自の処理を記述する．アダプタの記述には，Ruby 組み込みクラスのメソッドや専用の adapterAPI を利用でき，パターンマッチングやオブジェクトの操作など簡潔に表現できる．こうして定義したクラスのオブジェクトを `Task` や `Stream` の属性に代入することでアダプタが登録される．ランタイムはアダプタに対応するイベントの発生毎に，適切な引数を渡し `callback` メソッドをコールバックする．また，アダプタ自体は他の API クラスの属性として登録されるため，タスクネットワークモデルの構築方法に変更はなく，今までと同様の方式で記述できる．

図 3 に簡単なアダプタの記述例を示す．タスク t のイベント `out` に `MyAdapter` 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，メッセージ中のカンマが空白に置換されてからストリームに送られる．

また，エラー回復や通信制御など典型的なケースについては，あらかじめ該当する機能を持つアダプタを用意し，階層型のアダプタライブラリとして提供する予定である．これにより，ユーザは既存のアダプタから選んで使うか，より高度なモノを自作するか選択できる環境を目指す．

5. 実　　装

5.1 アダプタ起動部の実現

アダプタ起動部分の実装として，実行時に生じるイベントを検知し，規定のアダプタを呼び出せるよう Ruby 拡張用 API を用いてランタイムの拡張を行った．

まず，ランタイム内で MegaScript オブジェクト毎に想定するイベントの発生ポイントを定め，イベント処理をフックする機能を追加した．イベント発生時にはそれを引き起こしたオブジェクトを特定し，対応す

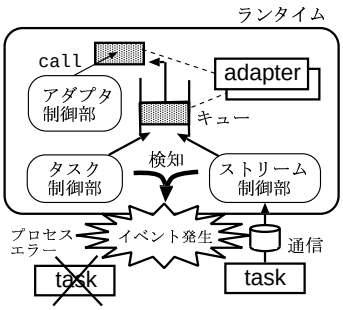

図 4 ランタイムにおけるアダプタの動作概要

るアダプタが存在すれば，callback メソッドに適切な引数を渡し呼び出す（図 4）．

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並行して動作させる必要があるため，複数のスレッドにより構成されている．また，Ruby インタプリタの制約から，Ruby コードはメインスレッド上で実行される必要がある．そこで，アダプタの実行はメインスレッド上のアダプタ処理部に依頼する形で行う．そのため，複数の処理要求に対処できるようキューを用意し，順次処理を行った後，依頼元に結果を返す．

5.2 アダプタ機能の実現

アダプタは各ローカル環境で実行できるよう，対応する MegaScript オブジェクトと同様にシリアライズ化した後，システムメッセージとしてスレーブ側へ送信する．ローカルホストに配置後は，適宜情報を更新しながらランタイムからの呼び出しを待つ．

adapterAPI は Ruby 拡張用 API を利用し，Adapter の組み込みメソッドとしてランタイム内部に実装した．また，アダプタからの登録先 MegaScript オブジェクト参照を実現するため，相手先へのポインタを持つインスタンス変数として@partner を Adapter 内に組み込んだ．

5.3 API クラスの拡張

アダプタを任意のイベント発生時に起動できるよう，API クラスを拡張する．各 API クラスにおいて，想定するイベント毎にアダプタを登録できるようインスタンス変数の追加を行った．この変数には，規定のアダプタクラスからなるスカラもしくは配列のオブジェクトを代入できる．

また，共通の MegaScript オブジェクトに登録されたアダプタ同士での情報共有を実現するため，各 API クラス内に@adapter_shared をインスタンス変数として組み込んだ．この変数には任意の Ruby オブジェクトを代入でき，登録されたアダプタから共通に参照できるようアクセサを実装した．

6. 評　　価

アダプタを利用する際に生じるオーバヘッドと一般的な機能を実現する際の記述性に関して評価を行った．

実行環境は Pentium4 2.8GHz，メモリ 512MB のホストを GbE で接続した PC クラスタであり，kernel-smp-2.6.1.11，MPICH1.2.6，Ruby1.8.2 を使用した．

6.1 オーバヘッドの評価

アダプタ機構の追加によるオーバヘッドの測定を 2 種類のアダプタにより行った．

まず，異なるホスト上の 2 プロセス間で単方向通信を繰り返した場合の，アダプタの有無によるオーバヘッドの変化を測定した．アダプタは，受信側タスクのメッセージ入力部に登録し，オーバヘッド測定のためアダプタのメソッド内は空とする．測定の結果，アダプタの実行は 5 パーセント未満のオーバヘッドで行えることが分かった．

次に，タスクの終了時にアダプタを起動した場合のオーバヘッドを測定した．タスクには，内部での処理を行わず，瞬時に終了するプログラムを用いた．また，アダプタのメソッド内では処理を行わないものとする．測定の結果，アダプタの実行は最悪の場合でも 30 パーセント程度のオーバヘッドで行えることが分かった．

以上の結果より，アダプタ追加によるオーバヘッドは全体に対して十分小さく，実用に耐えうると考える．

6.2 記述性の評価

アダプタの記述性の評価として，プロセスエラー発生時におけるタスクの再実行に要するコードを，ランタイム内に記述した場合とアダプタにより記述した場合とで比較した．

最初に，ランタイムプログラムを書き換えてタスク再実行機能を実装した場合について示す．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．この作業には広範な部分でのコードの追加や書き換えを要し，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．

次に，アダプタを用いてタスクの再実行機能を実現した場合のコード例を図 5 に示す．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 変数に対し，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に対応する変数に代入することで，エラー発生時に自動的にメソッドが呼び出される．

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

図 5 タスク再実行を行うアダプタのコード例

```
class ReStartAdapter2 < ReStartAdapter
  def callback(ret)
    if ret == false
      super(ret)
      self.partner.adapter_shared.each{ |i|
       self.partner.STDIN.print("#{i}")
      end
    end
  end
end
class CommAdapter < Adapter
  def callback(msg)
    self.partner.adapter_shared.push(msg)
  end
end
task.terminate = ReStartAdapter2.new
task.in = CommAdapter.new
```

図 6 拡張したタスク再実行アダプタのコード例

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，一般ユーザでも十分に対応できると考える．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，環境の変化へ柔軟に対応でき，機能の拡張も容易である．

図 6 に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示す．ここでは，`CommAdapter` と `ReStartAdapter2` の二つのアダプタから構成した．`CommAdapter` は，受信したメッセージを共有変数`@adapter_shared` に格納し，`ReStartAdapter2` からも参照可能とした．`ReStartAdapter2` では，親クラスの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することができる．

7. おわりに

本稿では，MegaScript 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の導入について述べた．アダプタ機構は，各ローカル環境でのコールバック機能を提供し，MegaScript オブジェクト上で発生する任意のイベントに対し，ユーザ規定の処理を適用可能とする．これにより，ユーザがエラー回復や通信の制御などを実行環境やアプリケーションの特性を考慮した高性能な機能として実現でき，より効率の良い並列計算をサポートする．

基本的なアダプタ起動部を実装し評価を行った結果，アダプタ利用に伴うオーバヘッドは実用上問題ない程度に抑えることができた．また，アダプタの記述に関しても，同等の機能をランタイムやタスクプログラム内に記述するのに比べ，記述の容易性や機能の拡張性が高く，複雑な機能であってもより少ないコードで簡潔に記述できることが分かった．

今後は，現状で未実装となっているイベントのアダプタへの対応や幅広いユーザを対象とする階層型ライブラリの整備を行う予定であり，それに伴い，実用性の高いアプリケーションによる評価を行い，その有用性を示していく．また，今回は記述性を優先したアダプタを導入したが，実行効率を重視したアダプタに関しても検討を行っていく．

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, HPC-95, pp. 119–124 (2003).
4) 森英一郎, 大野和彦, 佐々木敬泰, 近藤利夫, 中島浩: タスクネットワークの形状に基づく並列スクリプト言語のスケジューラ情報処理学会研究報告, HPC-100, pp. 19–24 (2004)
5) 阪口裕輔, 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におけるストリーム通信機構の改良, 情報処理学会研究報告, HPC-99, pp. 19–24 (2004).