2003－HPC－95　(21)
2003／8／5

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

西里一史[†]　大野和彦[†,☆]　中島浩[†]

我々はメガスケールコンピューティング向けの言語として，タスク並列スクリプト言語 MegaScript を提案している．MegaScript では，従来のコンパイラ型言語で作成されたプログラムをタスクとして扱い，複数のタスクを制御してタスク並列実行を行う．MegaScript は階層化されたライブラリモジュールを提供し，エンドユーザからコアユーザまで幅広く対応する．また，MegaScript はオブジェクト指向スクリプト言語 Ruby をベースとしている．

本研究では，MegaScript を構成する機能の一部としてランタイムシステムの設計と実装を行った．MegaScript ランタイムは基本的なタスク並列実行機能を提供し，その機能にはタスク・ストリーム定義，ストリーム接続，タスク・ストリーム生成機能がある．

Design and Implementation of The Runtime System for The MegaScript Parallel Language

Hitoshi Nishizato,[†] Kazuhiko Ohno[†,☆] and Hiroshi Nakashima[†]

We are designing and implementing a task parallel script language named MegaScript. MegaScript is developed for megascale computation. MegaScript provides a hierarchical library module for lower level parallel task management, which is easy enough for end users to write and detailed enough for core users. This language is based on the object-oriented script language Ruby.

In this research, we have designed and implemented a runtime system for MegaScript. The runtime provides functions such as task/stream definition, stream connection, and task/stream generation for task parallel execution.

1. はじめに

ゲノム情報/生命工学などのライフサイエンス，環境・気象シミュレーションなどの複雑な物理系が絡み合う系，大規模な都市における地震や山火事などの災害シミュレーションなどでは 1Pflops 以上の計算能力が期待されている．ここで，Pflops 以上の性能を得るためには，100 万台規模での汎用メガスケールコンピューティングが必要となる．

しかし，メガスケール規模でのプログラミングを，10^3 スケールでの既存の並列プログラミングモデルにより直接的に行うのは非常に困難である．そこで，メガスケール規模でのプログラミングを実現可能にするために，並列プログラムを単位とした「多重並列プログラミングモデル」を導入する必要がある．

† 豊橋技術科学大学
Toyohashi University of Technology
☆ 現在，三重大学
Presently with Mie University

現在我々は，既存の枠組で並列化された 10^3 スケール程度までの並列プログラムを単位とし，それらをタスク並列実行するための「タスク並列スクリプト言語 MegaScript」を提案している．

本稿では，MegaScript のランタイムシステムの設計と実装について述べる．ここで MegaScript ランタイムとは，分散環境上でタスク生成やタスク間通信を実現し，かつシステム情報やタスクの実行時間等の動的な情報収集も行うシステムである．また MegaScript ランタイムは，利用するノードへの自動的なタスク配置を行うために静的スケジューリングと動的スケジューリング機能を併せ持つ高機能なスケジューラを内包する．なお，MegaScript の全体的な構想については文献[1] を参照されたい．

2. MegaScript 概要

2.1 基本概念

タスク並列プログラミング言語は記述容易性とともに，複雑な処理を記述できる能力が必要である．さら

に，一般プログラマへの親和性および言語やライブラリの拡張性などを考慮し，オブジェクト指向スクリプト言語 Ruby[3] を拡張する形で MegaScript を設計・実装する．

2.1.1 タ ス ク

MegaScript の実行単位を「タスク」と呼ぶ．タスクは独立性の高い処理モジュールであり，逐次プログラムでも並列プログラムでも構わない．

現在は，タスクは他言語で作成された逐次の実行プログラムを想定している．

2.1.2 実 行 環 境

メガスケールコンピューティングでは 100 万台規模のプロセッサを扱うことを目標にしており，最終的な実行環境は広域に分散し，複数の並列計算機を含むヘテロなものになると予想される．このため，TCP/IP や MPI などの複数の通信手段を併用したり，異なるアーキテクチャの混在に対応するため複数のバイナリを使い分けたりといった実装が必要になる．

現在は，同アーキテクチャの PC クラスタを実装対象としている．

2.2 基 本 機 能

2.2.1 タスク生成

タスクを生成するには，基本的にタスクに対応する実行プログラムを子プロセスとして起動すればよい．MegaScript では，扱うノードが膨大であるため，タスクを生成するノードはユーザに直接指定させず，MegaScript スケジューラが自動的に決定する．

2.2.2 タスク間通信

複数のタスクが協調動作をして一つの問題を解くためには，それらのタスク間でデータのやりとりを行う手段を用意する必要がある．

本研究では，タスクは独立した実行プログラムであることを仮定しているので，そのプログラムを実行したプロセスに対し，外部から観測できるデータ送受信路はファイル入出力しかない．さらに，ライブラリの差し換えなども行わないとすれば，プロセスを起動する MegaScript ランタイムから見えるデータ送受信路は，プロセスの標準入出力だけである．そこで MegaScript では，各タスクの標準入出力を接続する方法でタスク間通信を実現する．

ここで，我々が既に開発した (Perl)+[2] にならって，論理的な通信路を表す「ストリーム」という概念を導入する．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できるものとする．MegaScript のタスク間通信は，このストリームを利用したストリーム通信によって行う．なお，アーキテクチャの違いなどを意識せずにタスク間で簡単に流せるのはテキストデータであるので，基本的に MegaScript のストリーム通信はテキストベースとする．そして，そのストリーム通信は図 1 のように行単位でバッファリングを行うものとする．但し，将来的にバイナリデータを流せるようにすることも検討中である．

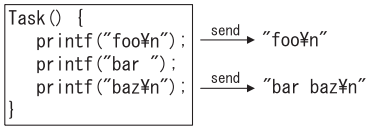

図 1 バッファリング

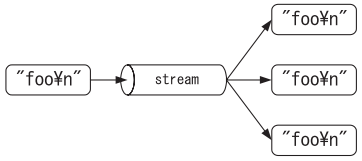

図 2 マルチキャスト

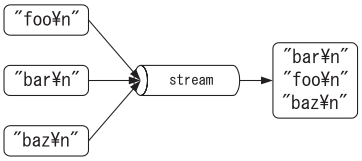

図 3 マージ

なお，ストリームの出力端に複数のタスクが接続された場合，そのストリームを流れるメッセージは，図 2 のように出力端に接続されているタスク全てにマルチキャストされる．

また，ストリームの入力端に複数のタスクが接続された場合，それらのタスクからそのストリームに入力されるメッセージは，図 3 のように一連のメッセージ列としてマージされる．ここで，図 3 では`"bar\n"`，`"foo\n"`，`"baz\n"`の順にマージされているが，実際は非同期にマージされるのでマージ結果は非決定的なものとなる．

3. ランタイム API

MegaScript はタスク並列実行機能を Ruby の API モジュールとして提供しており，現段階の API 仕様は以下のようになっている．

3.1 タスク定義

`define_task(`*exefile*`, [`*num*`=1])`

上記の形式でタスクの定義を行う．この API は，新しいタスクオブジェクトを生成して返す．その返り値

を変数に代入することで，タスクを表すタスク変数を作ることができる．これによって，MegaScript 内でタスクの実体に対して名前をつけることができ，ストリーム接続などの記述を簡略できる．

exefile には，このタスクが実行するファイル名を指定する．*num* には，生成するオブジェクトの個数を指定する．2 以上の値を指定したときは，指定個数のオブジェクトのリストを返す．したがって，返り値を代入した変数はタスクオブジェクトの配列となる．*num* を省略，または 1 を指定した場合は，タスクオブジェクトを 1 つ返す．

3.2 ストリーム定義

`define_stream([`*num*`=1])`

上記の形式でストリームの定義を行う．この API は，新しいストリームオブジェクトを生成して返す．その返り値を変数に代入する事で，ストリームを表すストリーム変数を作ることができる．

num には，生成するオブジェクトの個数を指定する．2 以上の値を指定したときは，指定個数のオブジェクトのリストを返す．したがって，返り値を代入した変数はストリームオブジェクトの配列となる．*num* を省略，または 1 を指定した場合は，ストリームオブジェクトを 1 つ返す．

3.3 ストリーム接続

`connect(`*task*`,` *stream*`,` *direction*`)`

上記の形式でタスクとストリームの接続処理を行う．この API により，実行時のネットワークモデルをタスクオブジェクトとストリームオブジェクトで仮想的に表すことができ，メモリ上に並列構造を構築できる．

task, *stream* にはそれぞれ `define_task()`, `define_stream()` で作成したオブジェクトを渡す．また，*direction* は入出力の方向を表し，定数 `IN` または `OUT` のいずれかを与える．ここで，`IN` はストリームをタスクの標準入力に接続するモードであり，`OUT` はストリームをタスクの標準出力に接続するモードである．

3.4 スケジューリング

`schedule()`

上記の形式でタスクのスケジューリングを行う．この API により，その時点で定義済みで未スケジューリングのタスク全てについて，最適なスケジューリングを行うように MegaScript スケジューラに依頼する．

現在は，MegaScript ランタイムがタスクを各ノードにほぼ均等に配置している．

3.5 タスク生成

`create_task(`*task*`, [`*args ...*`])`

上記の形式でタスクの生成を行う．この API により，`schedule()` で割り当てられたノードで，タスクをプロセスとして生成し実行を開始する．

task には，`define_task()` で定義したタスクオブジェクトを与える．また，第 2 引数以降は実行時引数としてタスクプロセスに渡される．

3.6 ストリーム生成

`create_stream(`*stream*`, [`*usebuff*`=false])`

上記の形式でストリームの生成を行う．この API により，`connect()` によってストリームの入出力端に接続されたタスク同士でタスク間通信を開始する．なおこの API は，ストリームの入出力端に接続されている全タスクの生成完了後に呼ばなければならない．

stream には，`define_stream()` で定義したストリームオブジェクトを与える．また，*usebuff* には，ストリームの出力端とそれに接続されたタスク間にバッファを設けるかどうかを bool 値で指定する．*usebuff* に true を指定すると，ストリームを流れてきたデータは，バッファリングされてストリームの出力端に接続されたタスクにすぐには渡されない．

4. プログラミング

MegaScript ではランタイム API を利用した，低水準ではあるが非常に柔軟で強力な並列処理の記述が可能である．しかし，科学技術計算などを目的とするエンドユーザは並列処理の知識に乏しく，ランタイム API を使いこなすことは困難であると考えられる．そこで，MegaScript プログラミングをユーザフレンドリーなものにするために，図 4 のような階層構造を持ったライブラリモジュールを提供する予定である．この階層構造を用意することによって，並列処理に詳しいコアユーザから問題を解くことのみに興味があるエンドユーザまで幅広く利用可能となる．

今回，Module 層までのライブラリモジュールのサブセットの設計を行った．ここで Module 層とは，ランタイム API をクラス・メソッドでカプセル化し，オブジェクト指向プログラミングをサポートする層である．図 5 に，Module 層でのプログラム例を示す．このプログラムは，タスクの 1 対 1 の接続を行うものである．図 5 を見ると分かるように，Module 層でのプログラミングは，タスクやストリームを意識して行わなければならない．

一方，Application 層でのプログラム例は図 6 に示すようなものになる．このプログラムは，Parameter-Survey ライブラリを利用したモンテカルロシミュレーションによる π の計算を行うものである．図 6 を見ると分かるように，Application 層でのプログラミン

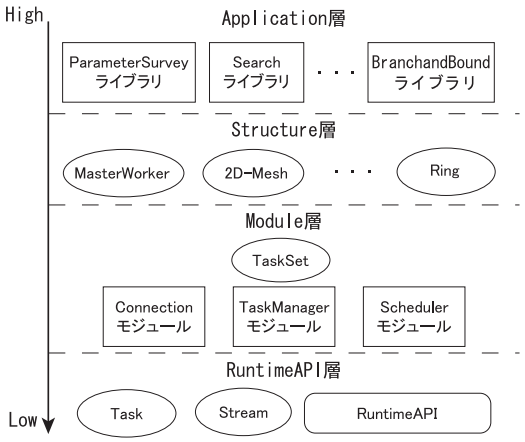

図 4 MegaScript ライブラリモジュール

```
class Producer < Task
  def initialize(*arg)
    @exefile = "./producer"
    @parameter = arg
  end
end
class Consumer < Task
  def initialize
    @exefile = "./consumer"
  end
end
tp = Producer.new("100")
tc = Consumer.new
s = Stream.new
Connection.build(tp, s, tc)
TaskManager.create(tp, tc)
Scheduler.run
```

図 5 Module 層でのプログラミング

グは，解きたい問題の種類が分かれば簡単に行うことができる．

5. ランタイムの実装

MegaScript ランタイムの実装にあたっては，C++ と MPI，および Ruby 拡張用 API を利用した．

5.1 MegaScript ランタイムの動作概要

ランタイムが Megascript プログラムの実行を開始するノードをマスターノード，ランタイムが利用するそれ以外のノードをスレーブノードと呼ぶ．図 7 に，MegaScript の実行モデルを示す．

MegaScript プログラムの実行が開始されると，

```
class Pi < Task
  def initialize(*arg)
    @exefile = "./pi"
    @parameter = arg
  end
end
ParameterSurvey(Pi, 10000..50000)
```

図 6 Application 層でのプログラミング

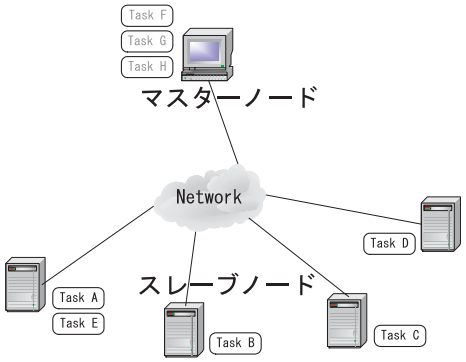

図 7 MegaScript の実行モデル

MegaScript が利用する全てのノードでランタイムプロセスが生成される．その後，マスターノード上では Ruby インタプリタが呼び出され，MegaScript プログラムのパースが行われる．一方，スレーブノード上では「受信スレッド」が起動される．この受信スレッドとは，他のノードから送信されてくるシステムメッセージを受信し，メッセージの種類に応じた処理を行うという動作を繰り返すスレッドである．

マスターノード上では，MegaScript プログラムの実行が進むにつれ，ランタイム API が次々と呼び出される．`create_task()`/`create_stream()` API が呼び出されると，スレーブノードに対し適切なシステムメッセージが送信され，スレーブノード上にタスクプロセスが配置されタスク並列実行が行われる．

そして，全てのタスクプロセスが実行を終了すると MegaScript ランタイムは終了する．

5.2 システムメッセージ

MegaScript が利用するノード間で送受信される主なシステムメッセージには以下に示すものがある．なお，M はマスターノード，S はスレーブノードを表す．

- `CREATE_TASK` (M→S)
  タスク生成要求
- `CREATE_TASK_FIN` (S→M)
  タスク生成完了通知
- `CREATE_LOWER_STREAM` (M→S)

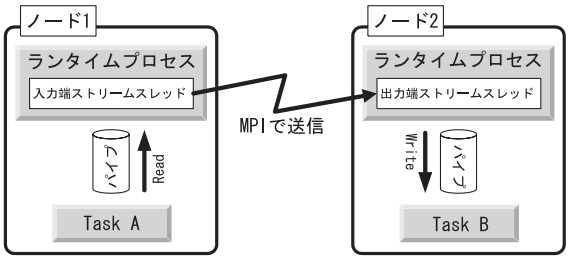

図 8 ストリームの実現

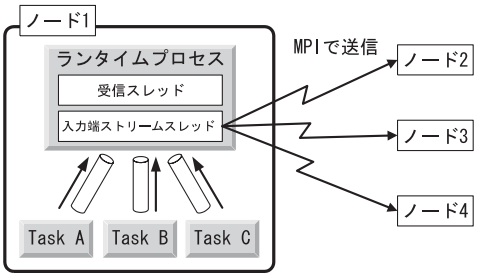

図 9 入力端ストリームスレッド

ストリームの出力端生成要求

- CREATE_LOWER_STREAM_FIN (S→M)
  ストリームの出力端生成完了通知
- CREATE_UPPER_STREAM (M→S)
  ストリームの入力端生成要求
- CREATE_UPPER_STREAM_FIN (S→M)
  ストリームの入力端生成完了通知
- STREAM_DATA (S→S)
  ストリームを流れてきたデータ

5.3 タスクの生成

タスクの生成は，スレーブノード上の受信スレッドが CREATE_TASK メッセージを受信した時に行われる．タスクの生成が完了すると，マスターノードには CREATE_TASK_FIN メッセージが返信される．

まず，タスクを生成するにあたって，pipe() システムコールを利用して，タスクプロセスへのデータ書き込み用とデータ読み出し用の 2 本のパイプを作成しておく．次に，タスクを生成するためには，基本的に fork() システムコールを呼び出した後，子プロセス側で exec 系の関数を利用してタスクプロセスの実行を開始すればよい．ただし，MegaScript ランタイムがタスクプロセスの標準入出力にアクセスできる必要があるため，fork() 呼び出し直後に，dup2() システムコールを利用して標準入出力と作成しておいたパイプとを繋ぎ換えておく必要がある．

そして，MegaScript ランタイムは，作成した 2 組のパイプを保持することにより，タスクプロセスの標準入出力を読み書き可能とする．

5.4 ストリームの実現

MegaScript ランタイムは，図 8 のようにストリームの両端に接続するタスクが生成されるノードに，必要に応じて入力端ストリームスレッドと出力端ストリームスレッドの 2 タイプのスレッドを生成する．これらのストリームスレッドは，MegaScript ランタイムがタスク生成時に作成しておいたパイプを経由して，タスクプロセスとデータのやりとりを行う．さらに，入力端ストリームスレッドは出力端ストリームスレッドに対して MPI による物理通信を行い，タスクプロセスから読み出したデータを送信する．

入力端ストリームスレッドの起動は，スレーブノード上の受信スレッドが CREATE_UPPER_STREAM メッセージを受信した時に行われる．入力端ストリームスレッドの起動が完了すると，マスターノードには CREATE_UPPER_STREAM_FIN メッセージが返信される．

図 9 に入力端ストリームスレッドの動作概要を示す．入力端ストリームスレッドは，まずストリームの入力端に接続されているタスクプロセスに繋がっているパイプ群に対して select() システムコールを発行する．その後，データの読み出しが可能なタスクプロセスを特定してデータを読み出す．読み出したデータは，上述のように対応する出力端ストリームスレッドが存在するノードに対して STREAM_DATA メッセージとして送信される．

一方，出力端ストリームスレッドの起動は，スレーブノード上の受信スレッドが CREATE_LOWER_STREAM メッセージを受信した時に行われる．出力端ストリームスレッドの起動が完了すると，マスターノードには CREATE_LOWER_STREAM_FIN メッセージが返信される．

図 10 に出力端ストリームスレッドの動作概要を示す．5.1 節で述べたように，全てのスレーブノードには受信スレッドが常駐しており，他ノードから送られてくるメッセージは全て受信スレッドが受信する．そして，送られてきたメッセージが STREAM_DATA メッセージであった時，受信スレッドはそのデータを処理すべき出力端ストリームスレッドに，ストリーム毎に用意されたキュー経由でデータを受け渡す．そして，そのキュー経由でデータを受け取った出力端ストリームスレッドは，ストリームの出力端に接続されているタスクプロセスに繋がっているパイプ群に対してデータを書き込む．

従って，あるストリームに関して，各ノードで生成される上述のストリームスレッドが協調して動作することで，論理的なストリームを実現できる．

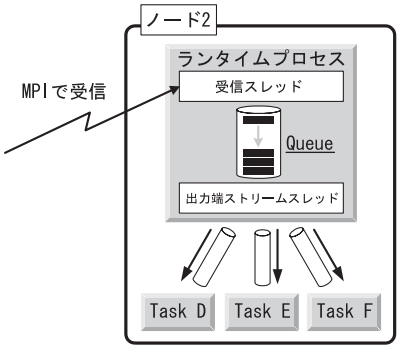

図 10 出力端ストリームスレッド

表 1 初期化・終了処理に要する時間

ノード数	実行時間 [sec]	
	MegaScript	C & MPI
2	0.32	0.27
4	0.54	0.61
8	1.04	1.23
16	2.05	2.31

6. 性能評価

評価用の実行環境として，Vine Linux 2.1.5 をインストールした Pentium III 1GHz の PC 17 台を Gigabit Ethernet で接続した PC クラスタを利用した．

まず，MegaScript ランタイムの初期化と終了処理にかかる時間を測定し，C と MPI による同様のプログラムの実行時間と比較した．測定結果を表 1 に示す．表 1 を見ると，初期化・終了処理にかかる時間は C と MPI によるものとほぼ変わらないことが分かる．

次に，15-Queen 問題を利用して性能評価を行った．15 個の `solver` に初期値を与えて実行させ，各 `solver` の出力を 1 つのストリームでマージし，結果を `collector` に集めるという 1 対 n 接続を行うプログラムを作成した．さらに，作成したプログラムの実行時間を，同様の動作を行う C と MPI によるプログラムの実行時間と比較することにより性能評価を行った．

実行時間の測定結果は，表 2 に示すものとなった．ここで，表 2 の OH とは，C と MPI で作成したプログラムの実行時間に対する MegaScript プログラムの実行時間のオーバーヘッドのことである．ちなみに，15-Queen 問題の解の個数は 2,279,184 個である．

表 2 より，解の個数のみ収集した場合，3%未満のオーバーヘッドで実行できることが確認できた．このオーバーヘッドは十分小さいものとみなせる．

しかし，約 230 万個の全ての解を収集した場合，約 45%もオーバーヘッドがあることが確認できた．この結果は実用上差し支えがあると考えられる．現在，`collector` が配置されたノードにおいて，高水準ファイル出力関数呼び出しが約 230 万回も発生していることが，オーバーヘッドの主な要因ではないかと考えている．そこで，今後はオーバーヘッドの詳細な解析とその改善を図る予定である．

表 2 15-Queen 問題の実行時間

	実行時間 [sec]		OH [%]
	MegaScript	C & MPI	
解の個数のみ収集	24.13	23.50	2.68
全ての解を収集	158.26	109.07	45.1

7. まとめと今後の課題

本稿では，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について記述した．実装したランタイムシステムの性能評価を行った結果，通信の頻度が大きい場合，実用上問題があることが確認できた．そこで，今後は通信ボトルネックの特定とその改善を行っていく必要がある．

一方，現在の MegaScript ランタイムは，システム情報やタスクの実行時間等の動的な情報の収集を行っていない．そこで，今後は MegaScript スケジューラとの連携に向けて動的情報収集機構を組み込む必要がある．さらに，より柔軟で強力な並列処理記述を可能にし，なおかつ MegaScript スケジューラとも親和性が高くなるようにランタイム API の構文およびセマンティクスを修正していく必要もある．

また，図 4 の階層構造を構築するにあたって，Application 層側からは使いやすさを重視して設計し，Runtime API 層側からは機能面を重視して設計し，全体的に整合性のとれたライブラリモジュールを作り込む必要がある．場合によっては，ランタイム API へのフィードバックも行っていく予定である．

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) 外崎由里子, 中田尚, 大野和彦, 中島浩: 並列スクリプト言語 (Perl)+の実装と設計, 並列処理シンポジウム JSPP'02, pp. 241-244 (2002).
3) 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).