Vol.2011-HPC-130 No.57
2011/7/29

MegaScriptにおける 大規模ワークフローの縮約機構の設計

三田明宏[†1] 仲貴幸[†1] 松本真樹[†1]
大野和彦[†1] 佐々木敬泰[†1] 近藤利夫[†1]

我々は，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としてMegaScript を開発している．MegaScript はオブジェクト指向言語であり，個々のタスクや通信路であるストリームなどをオブジェクトで表す．このため柔軟な記述が可能である一方，タスク数に応じたオブジェクトが生成されるため，実行可能なワークフローの規模がマスターホストのメモリ量に制約される問題がある．そこで，配列の縮約表現を用いて等価なワークフローを表すことで，大規模ワークフローの情報を少ないメモリ量で保持できる手法を提案している．

しかし，MegaScript 処理系に縮約機構を導入するにあたり，タスク間の通信が問題となる．現在の MegaScript 処理系は，一度にすべてのタスクプロセスを生成し，タスク間の通信を行っている．そのため，タスク間の通信を行う前に，縮約されているオブジェクトをすべて展開しなければならない．従って，この問題を解決するためにタスク間の通信方法を改良する必要がある．そこで本論文では，縮約状態のタスク群から必要なタスクのみを部分展開してプロセスを生成しタスク間通信を行えるように設計し，MegaScript 処理系に縮約機構を実装できることを明らかにする．

Design of MegaScript Runtime Contracting Large-scale Workflows

Akihiro Mita,[†1] Takayuki Naka,[†1]
Masaki Matsumoto,[†1] Kazuhiko Ohno,[†1]
Takahiro Sasaki[†1] and Toshio Kondo[†1]

We are developing a parallel script programming language *MegaScript* for large-scale workflows. MegaScript is an OOPL and each task and communication channel called stream is represented as an object. Although this feature enables flexible description of various workflows, the same number of objects are created for large amount of tasks. Thus, the executions of large-scale workflows are limited by the memory size of the master node. Therefore, we have proposed a scheme largely reducing the number of objects using array contraction.

However, the current implementation of MegaScript runtime creates all task processes at the beginning of a workflow execution. Thus all contracted task arrays must be expanded. In this paper, we show a new design of MegaScript runtime which enables workflow execution with progressive creation of task processes. This design can minimize the expansion of contracted arrays and efficient workflow execution is possible.

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．

大規模な並列アプリケーションを作成する手法の一つとして，複数の独立したプログラムをタスクとして組み合わせたワークフローが使われるようになっている[1)–3)]．ワークフローは，各タスクの独立性が高く高性能な既存ソフトウェアを再利用しやすいこと，タスク間のデータフローを非循環有効グラフ (ＤＡＧ) の形で直感的に記述できること，などの利点を持つ．また，ワークフローでは一般にタスクやタスク間通信の粒度が大きく，大規模な計算資源を安価に供給できる広域分散環境との相性がよい．このため，プロセッサ・メモリ・ディスクなど大量の計算資源を必要とする天文学・生物学・物理学などの科学技術の分野において，大規模な問題を解く実用的な手段として盛んに利用されている[4),5)]．

そこで我々は，大規模ワークフローの記述を目的としたタスク並列スクリプト言語MegaScript を提案し[6)]，開発を進めている[7),8)]．MegaScript では，個々のタスク (実行単位) やタスク間通信路であるストリームをオブジェクトで表し，これらのオブジェクトを生成・操作することでタスクのパラメータ設定やワークフロー構造の定義を行う．このため，任意の形状のワークフローを直感的に記述できる反面，ワークフローのタスク数と同規模の個数のオブジェクトが生成される．その結果，実行可能なワークフローの規模がスクリプトを実行するマスターホストのメモリ量に制約され，タスクスケジューリングや各実行ホスト (スレーブホスト) にタスク情報を送信する際のコストも，ワークフローの規模の増加に伴い無視できなくなる．

†1 三重大学
Mie University

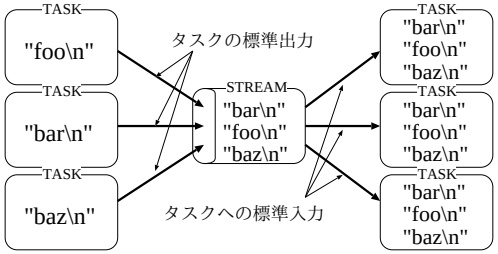

図 1 ストリームの振る舞い

そこで，ワークフロー構造を縮約して表現することにより，生成されるオブジェクト数を削減し，必要メモリ量やスケジューリング・通信コストを削減する手法を提案している[9]．この手法により，ユーザに余分な負担を強いることなく大規模なワークフローを効率よく実行できる．

MegaScript 処理系に縮約表現を導入するにあたり，タスク間の通信処理が問題となる．現在，スレーブホストではマスターホストから転送されたタスク情報を元に一度にタスクプロセスを生成しており，それを前提として通信機構が設計されている．縮約手法の効果を高めるには縮約されたタスク群から少数のタスクを展開しプロセス生成を行うことが望ましいが，その場合現行の通信機構が正しく動作しない．そこで，本論文ではタスクプロセスを一度に生成しなくてもタスク間通信ができる方法を提案し，縮約状態のオブジェクトに対しても正しく動作するストリーム通信機構を設計した．

以下，2 章で MegaScript の概要を説明し，3 章で問題点を述べ，4 章でタスク間通信の方法を例を用いて説明する．5 章で提案手法の考察を述べ，最後にまとめる．

2. 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粗粒度並列言語であり，独立したプログラムをタスクと見なして並列実行する．また，タスク間通信のために，ストリームと呼ばれる仮想通信路を提供している．一つのストリームの入出力端にはそれぞれに複数のタスクを接続することができ，入力端に接続したタスク群の出力が非決定的にマージされ，出力端に接続したタスク群にマルチキャストされる (**図 1**)．現実装では標準入出力のテキストストリームのみ扱い，行単位でアトミックなメッセージとしている．

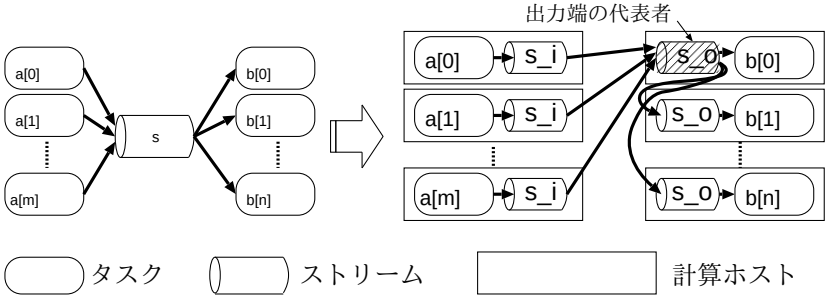

図 2 ストリームの実装手法

2.1 動作モデル

現在実装されている MegaScript 処理系 は，最初に処理系を起動したホスト上でマスタープロセスが起動し，利用可能な他のホストはスレーブプロセスを起動することでタスク実行ホストとして扱えるようにする．各プロセスには Ruby インタプリタ[10] が組み込まれており，マスタープロセスはこの上で MegaScript プログラムを実行し，メモリ上にオブジェクトで表現されたワークフローの全体像を構築する．

2.2 ストリーム

ストリームの実装概要を**図 2** に示す．MegaScript 処理系では，プログラム中に定義された 1 つのストリームに対し，入力端と出力端をそれぞれ生成する．入力端は入力側に接続されているタスクを実行するホスト毎に 1 個ずつ配置され，出力端は出力側に接続されているタスクを実行するホスト毎に 1 個ずつ配置される．

出力端のうち 1 つは代表出力端として処理系内で扱われる．ストリームを流れるメッセージは一旦代表出力端に集められてメッセージのマージが行われる．代表出力端には，それ以外の出力端の配置先ホストが処理系より与えられており，その情報を元にストリームメッセージの転送を行う．以上の実装により，処理系内でマージ・マルチキャストが実現されている．

2.3 スケジューラ

MegaScript のタスクスケジューラはワークフローの構造を解析し，各タスクの実行順や実行ホストを決定する．MegaScript では，ユーザの記述したメタプログラムや実行時プロファイリングを元に，プログラム毎に入出力量や計算量を表すコスト関数を生成しておくことができる．スケジューラは，スクリプト実行時に各タスクに与えられる実行時引数をこの

コスト関数に当てはめることで，各コスト値を計算し，通信・実行時間を推定する．

2.4 縮約表現

MegaScript はオブジェクト指向言語であり，個々のタスクやストリームなどをオブジェクトで表す．このため，柔軟な記述が可能である一方，タスク数に応じてオブジェクトが生成されるため，実行可能なワークフローの規模がマスターホストのメモリ量に制約されてしまう．そこで，メモリ上のワークフロー構造を縮約して表現することにより，生成されるオブジェクト数を削減しメモリ使用量を削減する．

MegaScript では同種のタスクを多数生成する場合を想定したタスク配列クラスを用意している．通常，大規模なタスク配列を用いるのはパラメータスイープのように同じプログラムを異なる条件で多数実行するケースであり，実行ファイル名は同一になる．実行時引数についても，オプションなどは同一のものが多く，タスク毎に異なる引数は一部に限られる．さらに，異なる引数を範囲を表す Range や引数付きブロックを与える Proc を用いるならば，これらのオブジェクトの規模はタスク配列の大きさによらず非常に小さい．ストリームについても，タスク配列は全体でまとめて接続を行うことが多く，この場合，配列内のタスクの入出力ストリームは同一になる．つまり，ユーザが記述した引数を展開せずにそのまま保持すれば，多くの場合は自然と小規模なデータ構造にとどまる．例えば，従来の手法では以下のコードを記述すると，タスク配列が一つ生成される．

```
TaskArray.new(n, 'sim', 1..n)
```

従来手法では，**図 3**(a) のように個々のタスクオブジェクトを生成し，それらを要素とするタスク配列が作られる．しかし，与えられた配列サイズと引数を保持する縮約状態では図 3(c) のように少ないオブジェクト数で表現できる．一部不規則な要素を持つ場合は，そのオブジェクトのみ生成しハッシュで保持する (図 3(b))．

従来のタスク配列は非縮約状態で実装されているが，これらの 3 種類の表現を使い分けることで生成するオブジェクト数を最小限に抑えられる．また，説明は省略するが，タスク配列が保持するストリームの接続情報やストリーム配列についても同様の手法で縮約できる．

3. ストリーム通信機構の問題点

本章では，現在の MegaScript 処理系が抱えている問題点について，例を用いて説明する．**図 4** のように定義されたタスクとストリームに対して，スケジューラによって**図 5** のように配置された場合を考える．この例では，n 個のタスク b のうち，ホスト 1 に $b[0]$ から $b[i-1]$ までの i 個のタスクを，ホスト 2 に $b[i]$ から $b[j-1]$ までの $j-i$ 個のタスクを，

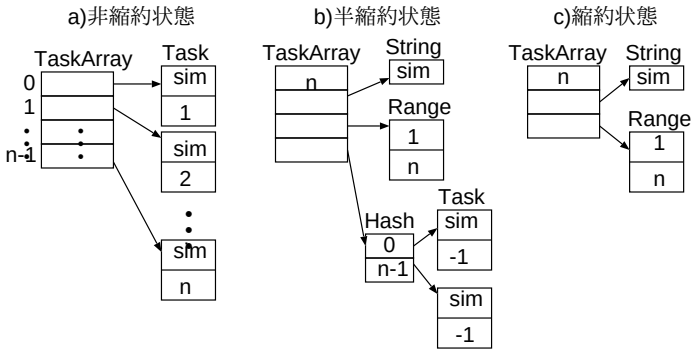

図 3 タスク配列の縮約表現

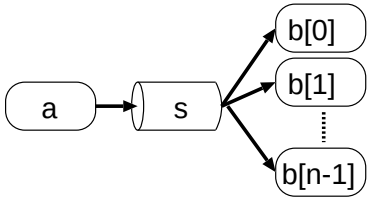

図 4 一対多のワークフロー

ホスト 3 に $b[j]$ から $b[n-1]$ までの $n-j$ をそれぞれ割り当てている．また，代表出力端とタスク a はホスト 2 に配置されている．図 5 中でタスク a から出力されたメッセージは代表出力端に収集され，ホスト 1，3 に転送される．

しかし，このタスク間通信では，すべてのタスクプロセスが生成されていることが前提となっている．例えば，$b[0]$ のタスクプロセスが生成していない場合，転送されたデータは転送先がないためにデータを破棄するしかなく，後から $b[0]$ が生成してもデータを再度要求することができない．しかし，縮約表現を用いた大規模ワークフローを実行する際，一度にタスクプロセスを生成すると，各タスクプロセスに十分なメモリ量を確保することは難しくなる．そこで，必要となるタスクプロセスのみ生成し，一部のタスクプロセスが遅れて生成されても，適切にタスク間通信ができる設計を行う．

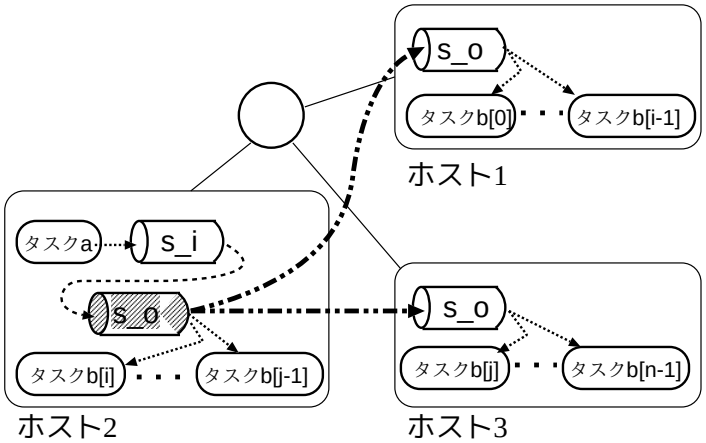

図 5 一対多の従来実装手法

4. ストリーム通信機構の改良

縮約機構を導入するために，ストリーム通信を行うタスクプロセスが一度に生成されるとは限らないという問題を解決する必要がある．この問題を解決するために，送信されたメッセージをバッファに格納し，再度の転送要求にも応えることができる機構を導入する．この改良を行うことで，必要になるまでタスクプロセスやストリームの入出力端の生成を遅らせることが可能となり，メモリ資源を有効活用できるようになる．

上記の機構を実現するためには，次のような問題を考える必要がある．

(1) ストリームの入出力端をいつ生成するか
(2) どのホストがバッファ機能のあるストリーム入出力端を持つか
(3) 縮約されているタスク/ストリームをどう扱うか

本章では一対多，多対一，多対多それぞれのタスク間通信において，これらの問題の解決方法を述べる．(3) の問題については，4.4 節で述べる．

4.1 一対多のタスク間通信

一対多のタスク間通信の構造である図 4 を例に説明する．(1) の問題について，メッセージ送信元であるタスク a のプロセスを生成するときに，その通信路であるストリームを生成する．(2) の問題について，代表出力端では，データの再度の転送要求に応えるために出力データを保持するバッファが必要となる．一方，他の出力端にもバッファを持たせることは，利点と欠点がある．利点は，通信回数を減らせる可能性があることである．図 5 のタスク $b[k]$(但し，$0 \leq k \leq i-1$) のプロセス生成が遅れたときを例に考える．全出力端にバッファを設け，代表出力端のミラーを出力端のバッファに持たせる．この事によって，タスク $b[k]$ はホスト 1 にデータ転送を依頼しなくても，自ホスト内のバッファから取ってくることで通信処理を行わずに解決できる．欠点は，複数ホストにバッファを持たせると，メモリ資源を余分にとってしまうことである．そこで，遅れてタスクプロセスを生成し出力データを要求した場合，代表出力端のバッファにあるデータを転送するときにそのホストの出力端にバッファを生成し，代表出力端にあるバッファ内のデータをコピーする．このようにすることで，二度目以降は代表出力端のあるホストにバッファデータの転送要求を出さずとも，自ホスト内のバッファからデータを転送することで要求を満たすことができる．この処理によって，実行速度の向上と処理系で占有するメモリ量の削減の両立をはかる．

以上のことをまとめると，一対多のタスク間通信の処理は以下の通りである．

(i) タスク a のタスクプロセスを生成するとき，ストリームを生成する
- ストリームの代表出力端にバッファを持たせる

(ii) 出力端バッファのデータ転送要求が発生したとき，そのホストの出力端にバッファを生成する
- このバッファは代表出力端のミラーリングを行う

(iii) データ出力側のタスクプロセスがメッセージ出力をする度に，代表出力端に転送する
- 代表出力端に転送されたメッセージをすべてバッファに格納する

(iv) 代表出力端を経由して受信タスクがあるホストに転送する
- 受信タスクプロセスが存在しなければ，データを転送しない

図 6 の例を用いて，処理の流れを説明する．各ホストにタスクを配置する前に，スケジューリング結果から受信タスクがもっとも多く配置されるホスト 2 に生成される出力端を代表出力端に決定する．そして，実際にタスクを配置した場合，図 6 のようになる．最初に，タスク a のプロセスが生成されるとする．(i) よりストリームの入出力端を生成する．このときに代表出力端は，バッファの生成を併せて行う．(iii) よりタスク a の出力は，代表出力端に集められ，バッファに格納される．次に，ホスト 2 のタスク $b[2]$ とホスト 3 のタスク $b[5]$ のプロセスが生成されるとする．(iv) よりタスク $b[2]$ には代表出力端に流れた出力データを転送する．また，出力データは代表出力端を経由してホスト 3 の出力端に転送される．(ii) よりホスト 3 の出力端にバッファを生成し，バッファ内に出力データを格納する．そし

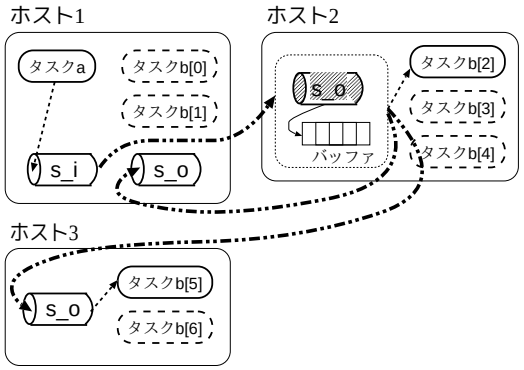

図 6 一対多のタスク間通信の例

て，バッファ内のデータをタスク $b[5]$ に転送する．タスク a，$b[2]$，$b[5]$ のプロセスの終了後，それぞれのホストにあるタスク $b[0]$，タスク $b[3]$，タスク $b[6]$ のプロセスが生成されるとする．タスク $b[0]$，$b[3]$，$b[6]$ は自ホスト内のバッファにタスク a の出力データが保持されているため，そのバッファからタスク a の出力データが転送される．自ホストにあるバッファ内のデータを転送することで，ホスト間の通信処理を出来る限り行わずに出力データの転送ができる．次に，タスク $b[0]$，タスク $b[3]$，タスク $b[6]$ のプロセスの終了後，タスク $b[1]$，タスク $b[4]$ のプロセスが生成されるとする．タスク $b[1]$ とタスク $b[4]$ も同様の処理で自ホスト内のバッファから出力データが転送される．以上の操作により，タスク a の出力データがすべてのタスク b に転送されることになる．

4.2 多対一のタスク間通信

多対一のタスク間通信の構造を**図 7** を例に説明する．3 章で挙げた (1) の問題について，メッセージ送信タスクであるタスク a のプロセスのいずれかを生成するときに，ストリーム入出力端を生成する．図 7 ではタスク a が複数ある．そこで，ストリームオブジェクトに入出力端の生成フラグを設定し，同じストリーム入出力端を複数生成しないように制御する．例えば $a[0]$ が最初にプロセス生成される場合，ストリーム入出力端の生成を行い，ストリーム入出力端のフラグを生成済に変更する．$a[1]$ から $a[n-1]$ のタスクプロセスを生成しても，ストリーム入出力端のフラグが生成済となっているため，ストリーム入出力端を生成しない．(2) の問題について，ストリーム出力端が一つしかないため，必然的にタスク b が配置されるホストの出力端が代表出力端となる．以上より，多対一のタスク間通信の処理は，一対多のタスク間通信の処理の (i) に複数のストリームを生成しない操作を追加するだけで良い．

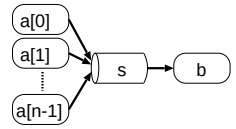

図 7 多対一のワークフロー

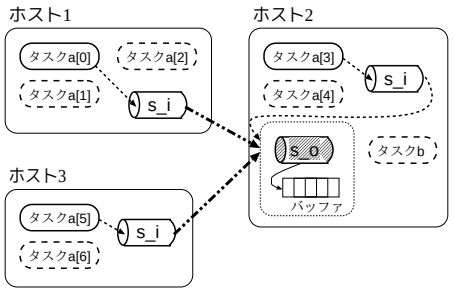

図 8 多対一のタスク間通信の例

図 8 を例に，処理の流れを説明する．各ホストにタスクを配置する前に，スケジューリング結果から受信タスクが配置されるホスト 2 のストリーム出力端を代表出力端に決定する．そして，実際にタスクを配置する．最初に，各ホストに配置されているタスク $a[0]$，$a[3]$，$a[5]$ のプロセスが生成されるとする．(i) より，ストリームを生成する．併せて代表出力端はバッファの生成を行う．(iii) より，タスク $a[0]$，$a[3]$，$a[5]$ の出力データは，代表出力端に転送し，バッファに格納される．この例では受信タスクであるタスク b のプロセスが生成されていない．そこで (iv) より，代表出力端からタスク b へのデータ転送は行われない．タスク $a[0]$，$a[3]$，$a[5]$ のプロセスの終了後，タスク $a[1]$，$a[4]$，$a[6]$ のプロセスが生成されるとする．これらのタスクの出力結果も同様に代表出力端のバッファに格納される．タスク $a[1]$，$a[4]$，$a[6]$ のプロセスの終了後，タスク $a[2]$，b のプロセスが生成されるとする．タスク b がタスク a 群の出力結果の転送を要求する．そこで，代表出力端のバッファのデータをタスク b に転送する．そして，タスク $a[2]$ の出力データも同様に代表出力端に転送し，バッファに格納後，タスク b に転送する．以上の操作により，タスク a 群の出力データがすべてのタスク b に転送されることになる．

4.3 多対多のタスク間通信

多対多のタスク間通信の構造を**図 9** を例に説明する．この構造は一対多と多対一を組み合わせたものであり，それぞれのタスク間通信と同じ処理で多対多のメッセージ転送ができる．

図 10 を例に説明をする．まず，代表出力端となるホストを決定する．今回の例ではどのホストも受信タスク b の配置される数が同じであるため，ホスト 1 の出力端を代表出力端と

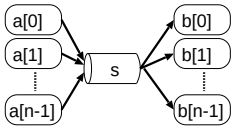

図 9 多対多のワークフロー

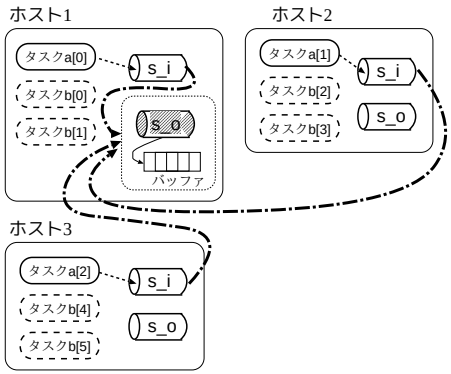

図 10 多対多のタスク間通信の例

する．最初にタスク $a[0]$，$a[1]$，$a[2]$ のプロセスが生成されるとする．その時点でストリーム入出力端を生成する．タスク $a[0]$，$a[1]$，$a[2]$ のそれぞれの出力データは，代表出力端に集められバッファに格納される．タスク $a[0]$，$a[1]$，$a[2]$ のプロセス終了後，タスク $b[0]$，$b[2]$，$b[4]$ のプロセスが生成されるとする．タスク $b[0]$ の実行ホストは代表出力端のあるホストと同じであることから，代表出力端のバッファに格納されている出力データが転送される．一方，タスク $b[2]$，$b[4]$ は代表出力端と異なるホストで実行されるため，ホスト 2 とホスト 3 のストリーム出力端に出力データを転送する．このときに，ホスト 2 とホスト 3 はそれぞれバッファを生成し，出力データをバッファに格納する．各ホストにあるバッファのデータをタスク $b[2]$，$b[4]$ に転送する．タスク $b[0]$，$b[2]$，$b[4]$ のプロセス終了後，タスク $b[1]$，$b[3]$，$b[5]$ のプロセスも同様の手順で生成し，データを転送する．以上の操作でタスク a 群の出力データをすべてのタスク b 群に転送することができる．

4.4 タスク/ストリームの縮約導入後のタスク間通信方法

4.4.1 タスクの縮約

縮約されたタスク群はオブジェクトの転送時間および必要メモリ量を削減するために，各ホストに縮約状態のままタスクを転送される．しかし，縮約されたタスク群はスケジューリングの結果によって複数のホストに分割して配置される可能性がある．例えば**図 11** の縮約状態のタスク群が，タスク $[0]$ からタスク $[i-1]$ をホスト 1 に，タスク $[i]$ からタスク $[j-1]$ がホスト 2 に，タスク $[j]$ からタスク $[n-1]$ がホスト 3 にスケジューリングされるとする．このスケジューリング結果を満たしつつ縮約状態を展開せずにタスク転送を行うために，縮約されたタスク群を指定位置で分割する機能を追加する．例では，タスク $[0]$ からタスク $[i-1]$ の縮約表現のタスク群に，タスク $[i]$ からタスク $[j-1]$ の縮約状態のタスク群に，タスク $[j]$ からタスク $[n-1]$ の縮約状態のタスク群に分割している．それぞれのタスクオブジェクトは縮約状態を維持して，各ホストに転送される．縮約されたタスク群から必要とされるタスクを展開し，そのタスクプロセスを生成するようにすれば，タスクオブジェクトの転送時間および必要メモリ量を減らすことが出来る．

タスク群が縮約されていた場合の一対多のタスク間通信を，**図 12** を例に説明する．スケジューリング結果から，受信タスクであるタスク b が最も多く配置されるホスト 1 を代表出力端とし，各ホストに縮約されたタスク b 群を配置する．最初に，タスク a，$b[i]$，$b[j]$ のプロセスが生成されるとする．タスク $b[i]$ と $b[j]$ は縮約表現の一部となっているため，部分展開が必要となる．また，部分展開を行った後，親である縮約状態のタスク b 群からタスク間の接続情報を持っているストリームオブジェクトを調べ，出力端の接続先をタスク $b[i]$ に変更する．タスク $b[j]$ に関しても出力端の接続先を変更する．その後，タスク $b[i]$ とタスク $b[j]$ のプロセスを生成する．タスク $b[i]$ とタスク $b[j]$ がタスク a の出力データを要求するため，ホスト 2 とホスト 3 のストリーム出力端にバッファを生成し，代表出力端のバッファに格納されている出力データをホスト 2 とホスト 3 のバッファに転送する．そして，自ホスト内にあるバッファのデータをタスク $b[i]$ とタスク $b[j]$ に転送する．タスク a とタスク $b[i]$，タスク $b[j]$ のプロセスが終了後，タスク $b[0]$，タスク $b[i+1]$，タスク $b[j+1]$ がプロセス生成されるとして，これらのタスクを部分展開し，ストリームの接続先を変更し，プロセス生成を行う．タスク a の出力データを自ホスト内にあるバッファから，タスク $b[0]$ とタスク $b[i+1]$，タスク $b[j+1]$ に転送する．タスク $b[0]$ とタスク $b[i+1]$，タスク $b[j+1]$ のプロセスが終了後，残りのタスクも順次展開して，自ホスト内のバッファにあるタスク a の出力データを渡すことで，すべてのタスク b にタスク a の出力データを転送することができる．

多対一，多対多のタスク間通信も同様に必要時に展開・生成することで非縮約の転送と同じ処理でタスク間の通信ができる．

4.4.2 ストリームの縮約

MegaScript で記述したワークフローには，**図 13** のような接続関係がよく出現する．このワークフローを縮約すると，**図 14** のように表現でき，タスク群とストリーム群は縮約状態のまま接続関係を保持することができる．また，タスク a 群とタスク b 群をスケジューリングすると，縮約状態のまま纏めて転送することによるオブジェクトの転送時間とスケジューリングコストの削減の観点からタスク $a[k:l]$ とタスク $b[k:l]$(但し，$0 \leq k \leq n, k \leq l \leq n$)

Vol.2011-HPC-130 No.57
2011/7/29

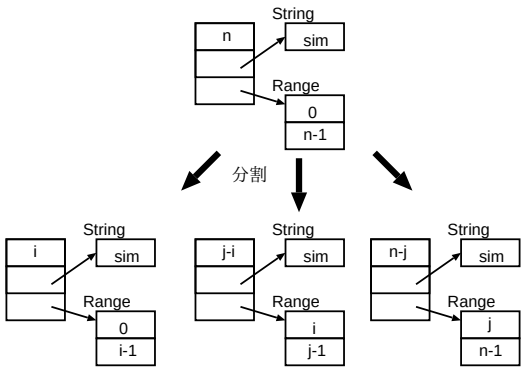

図 11 縮約タスクの分割

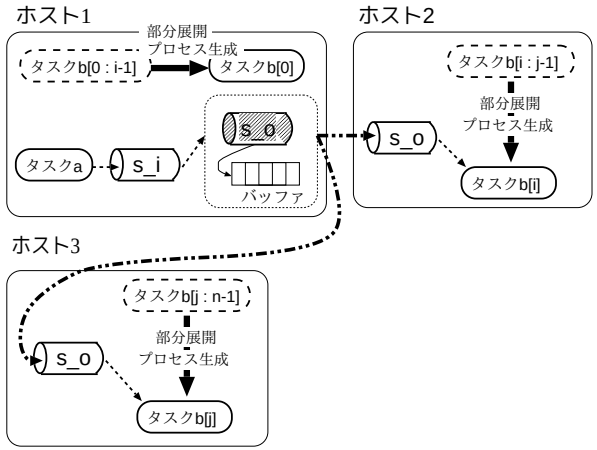

図 12 縮約状態の一対多のタスク間通信の例

は同じホストに配置される．そこで，縮約状態のストリーム $s[0:n]$ を分割してストリーム $s[k:l]$ を生成し，同じホストに配置し，そして，タスク間通信の処理で，ストリームオブジェクトを部分展開する．そのストリームオブジェクトの入出力端に接続するタスクオブジェクトを設定し，ストリーム入出力端を生成する．タスク群も適時，部分展開を行いプロセスを生成する．ストリームの入出力端から自ホスト内にあるタスク b へデータを転送する．

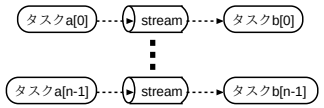

図 13 一対一のワークフロー例

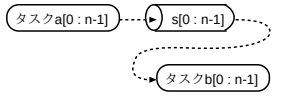

図 14 縮約した一対一のワークフロー例

図 15 を例に説明する．スケジューリング結果に従いタスク a 群とタスク b 群を分割する．縮約されたタスク群を各ホストに転送する．タスク $a[0:i-1]$ とタスク $b[0:i-1]$，及びストリームオブジェクト $s[0:i-1]$ をホスト 1 に，タスク $a[i:j-1]$ とタスク $b[i:j-1]$，及びストリームオブジェクト $s[i:j-1]$ をホスト 2 に，タスク $a[j:n-1]$ とタスク $b[j:n-1]$，及びストリームオブジェクト $s[j:n-1]$ をホスト 3 に配置したとする．最初にホスト 1 内のタスク a[0] のプロセスが生成されるとする．タスク群 $a[0:i-1]$ から部分展開を行いタスク $a[0]$ のプロセスを生成する．そして，そのプロセスで必要とされる通信路 $s[0]$ のオブジェクトを部分展開し，ストリーム入出力端を生成する．タスク $a[0]$ の出力データを出力端 $s\_o[0]$ のバッファに格納する．次に，ホスト 1 内のタスク $b[0]$ のプロセスが生成されるとする．タスク $b[0:i-1]$ から部分展開を行いタスク $b[0]$ のプロセスを生成する．出力端のバッファから出力データをタスク $b[0]$ のプロセスに転送する．順次，タスク $a[0:i-1]$，タスク $b[0:i-1]$，$s[0:i-1]$ を展開していき，データ転送を行う．ホスト 2，3 もホスト 1 の処理と同様にタスク a 群とタスク b 群，及びストリームオブジェクト s 群を順次部分展開して，ストリーム入出力端を生成し，各タスク a の出力データをタスク b に転送する．

以上の処理を行うことで，タスクやストリームが縮約されていても，正しくタスク間の通信ができる．

5. 考　　察

今回のタスク間通信の例は，1 ホストで 1 プロセスしか実行しない環境で説明した．1 ホストで複数のプロセスが実行されることも考えられるが，その場合も本論文で述べた通信方法で問題なく実行できる．

また，一対多のタスク間通信には以下の問題が考えられる．一対多のタスク間通信において，代表出力端の受信処理に負荷が集中することが考えられる．この問題を解決するために，入力端にもバッファを設けて同ホスト内のタスクのメッセージデータを集めて，パッキングし纏めて送信することで受信の負荷を減らすことができる．しかし，これは送信元タスクが配置されているすべてのホストの入力端にバッファを生成するため，タスクプロセスが

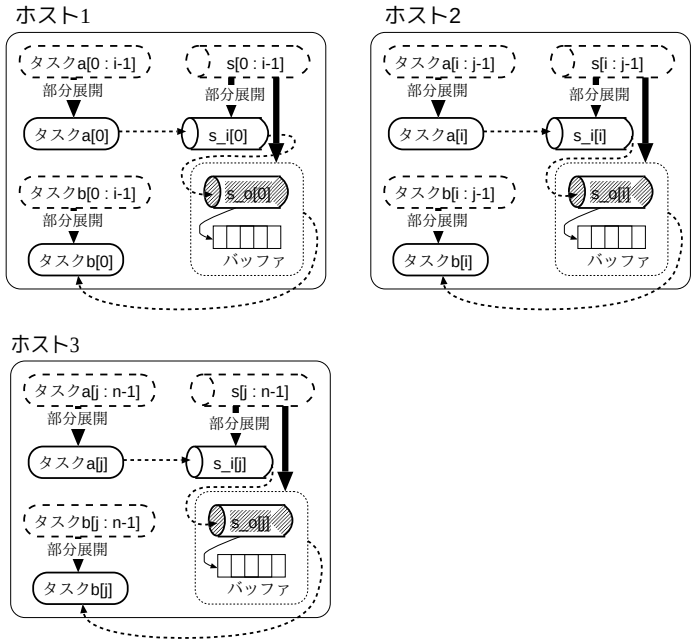

図 15 (縮約あり) 一対一のタスク間通信の例

使えるメモリ資源が減る．この問題をどのように解決するかは今後の課題である．

6. おわりに

本論文では，現在実装されている MegaScript 処理系に縮約機構を導入する際の問題点を述べ，その問題を解決するために従来の手法を改良した新たなストリーム通信機構の提案を行った．そして，タスク群やストリーム群が縮約されていても，適時部分展開することで縮約状態のままタスク間通信ができることを示した．

今後は，MegaScript 処理系に提案手法の実装を行う必要がある．また，大規模並列処理を行う実アプリケーションで性能評価を行うことも今後の課題である．

参考文献

1) Kenjiro Taura, Takuya Matsuzaki, Makoto Miwa, Yoshikazu Kamoshida, Daisaku Yokoyama, Nan Dun, Takeshi Shibata , Choi Sung Jun and Jun'ichi Tsujii. Design and Implementation of GXP Make – A Workflow System Based on Make. eScience, IEEE International Conference on, 214-221, 2010.
2) Ewa Deelman, Gurmeet Singh, Mei-Hui Sua, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Scientific Programming. 219-237, 2005.
3) Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz and Ian Foster. Swift: A language for distributed parallel scripting. Parallel Computing. 2011.
4) E.Deelman, D.Gannon, M.Shields, and I.Taylor. Workflows and e-science: An overview of workflow system features and capabilities. Future Gener. Comput. Syst., 25:528-540, 2009.
5) Y.Gil, E.Deelman, M.Ellisman, T.Fahringer, G.Fox, D.Gannon, C.Goble, M.Livny, L.Moreau, and J.Myers. Examining the challenges of scientific workflows. Computer, 40:24-32, 2007.
6) 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, 73-76, May 2003.
7) 西里 一史, 大野 和彦, 中島 浩. タスク並列スクリプト言語 MegaScript 向けランタイムシステム. In 情処研報 2004-HPC-99, pages 7-12, July, 2004.
8) 湯山 紘史, 大塚 保紀, 西里 一史，大野 和彦, 中島 浩. タスク並列スクリプト言語 MegaScript によるタスクモデルの記述手法. 先進的計算基盤システムシンポジウム SACSIS2004, 135-136, May 2004.
9) Kazuhiko Ohno, Akihiro Mita, Masaki Matsumoto, Takahiro Sasaki, Toshio Kondo, and Hiroshi Nakashima. Efficient Implementation of Large-scale Workflows based on Array Contraction. Proceedings of the 22th IASTED International Conference on Parallel and Distributed Computing and Systems –PDCS 2010–, 153-162, November 2010.
10) まつもとゆきひろ and 石塚圭樹. オブジェクト指向スクリプト言語 Ruby. ASCII, 1999.