社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report

2004－HPC－99　(4)
2004／7／30

タスク並列スクリプト言語における ストリーム通信の改良

阪口 裕輔[†]　大野 和彦[†]　佐々木 泰敬[†]
近藤 利夫[†]　中島 浩[††]

大規模並列分散処理においては，タスク間における通信処理の全体に占める割合が大きくなりやすい．メガスケールコンピューティング向けの言語として提案されている MegaScript では，既存の資源の有効利用や様々な言語のタスクに対応するため，標準入出力を利用したストリーム通信を採用している．しかし，それに伴い，通信処理記述時の制約が発生し，通信オーバヘッドの増大を招くことがある．

本稿では，MegaScript におけるタスク間通信の性能向上を目的とし，プログラマブルな通信インタフェースの導入を提案する．このインタフェースは，タスク・ストリーム間において通信時の仲立ちを行い，それぞれに対して都合の良い接続口を提供することで通信時の制約を緩和させる．また，ユーザ側でタスクに応じたインタフェースを実装することで，通信処理の最適化をはかることができる．

Improvement of Stream Data Communication for a Task Parallel Script Language

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

In large-scale parallel distributed processing, the rate of the communication time to the whole processing time tends to become large. MegaScript is a programming language designed for mega-scale computing, and adopts a stream communication scheme using standard input/output. In this scheme, various kind of programs can be used or reused as script tasks, regardless of their programming languages. However, it causes some restrictions in description and may increase communication overhead.

In this paper, we propose a programmable communication bridge called *interface* to improve task communication in MegaScript. A interface provides connection ports to both tasks and streams, and mediates their communication. Using this interface, some restrictions in description is dissolved. The user can customize each interface to optimize communication.

1. はじめに

近年，複雑な物理系が絡み合う環境·気象シミュレーションやゲノム情報解析など Pflops 以上の計算能力が求められる分野が増えてきており，100 万台規模のプロセッサを用いたメガスケールコンピューティングの必要性が高まってきている．

しかし，メガスケール規模でのプログラミングを，既存の並列プログラミングの延長として実現することは困難である．その解決策として，既存の逐次プログラムや部分問題を並列化した小規模な並列プログラムを組み合わせることにより大規模な並列性を引き出す「多重並列性」が考えられる．また，メガスケール規模の環境においては性能の異なる計算機やネットワークの集合としてシステムを構築する必要があるため，それらの資源を効率よく利用するプログラミング環境が必要である．そこで，これらを実現する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*[1)~3)] が提案されている．

並列処理では，それぞれのタスク間においてデータのやりとりが必要となるが，MegaScript ではタスク間通信の方法として，各タスクの標準入出力を接続したストリーム通信機構を用いている．しかし，その特性上，タスク・ストリームの接続に関する制約が大きく，冗長なタスクネットワーク構造，非効率な通信処

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

理が生じる．

そこで，このような制約を緩和し，より柔軟なタスク間通信と既存のプログラム資源の有効利用をサポートするため，ストリーム通信機能の拡張を行った．現在の MegaScript では，各タスクはストリームとの間で直接データのやりとりを行っているが，この間に通信時の仲立ちを行う「インタフェース」機構を導入する．インタフェースは，従来のタスク生成における容易性を残したまま，スクリプト側でより適切・効率的な記述を可能にする．また，インタフェース内では，フィルタリングやタグによるメッセージの操作といった処理を可能にし，より高機能なタスク間通信機構を実現する．

本稿では，MegaScript におけるストリーム通信の改良としてインタフェース機構について述べる．以下，次章で MegaScript の概要を述べ，3 章では今回提案するインタフェースについて概説する．4 章および 5 章ではインタフェースの形態とその実装方法について述べる．6 章ではインタフェース実装にともなう予備評価について述べ，最後にまとめとする．

2. MegaScript の概要

MegaScript は逐次／並列プログラムを一つのタスクとして扱い、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用いることでデータの送受信を行うことができる．

処理の主要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 プログラム内にはタスク・ストリームネットワークの構築に関する情報を記述する．そのため，プログラムの記述性・拡張性を考慮し，オブジェクト指向スクリプト言語 Ruby[4] を拡張する形で設計・実装されている．

2.1 タ ス ク

MegaScript の実行単位はタスクと呼び，逐次／並列の独立した処理モジュールとして実装される．タスクとしては既存のプログラムを再利用することを想定しており，それらを組み合わせることで効率よく大規模な処理モデルを実現することが可能である．

2.2 ストリーム

MegaScript におけるタスク間通信では，実行単位であるタスクが独立した外部プログラムであり，既存の資産を有効利用するといった点を考慮し，各タスクの標準入出力を接続する形でタスク間通信を実現している．

ここで，MegaScript ではタスクを論理的に接続する通信路をストリームと呼び，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われる．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，多対多通信が可能である．

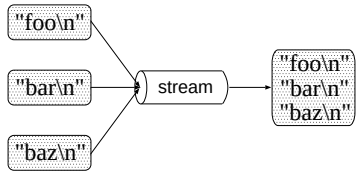

図 1 メッセージのマージ

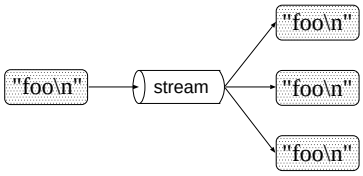

図 2 メッセージのマルチキャスト

現バージョンでは，ストリームに流せるメッセージはアーキテクチャの差異を意識せずに送れるテキストデータである．ストリームの入力端に複数のタスクが接続された場合，図 1 に示すようにメッセージは行単位でマージされる．また，出力端に複数のタスクが接続された場合は，図 2 に示すようにメッセージは接続されたタスク全てにマルチキャストされる．

2.3 プログラミング

MegaScript のプログラムでは，並列動作させるタスクと，タスク間のデータ受け渡しに必要なストリームを定義し，両者を用いてタスク・ストリームネットワークの形状を設定して，タスクを起動させる．このとき接続されたタスク間において通信路が確立され，通信データの送受信が可能となる．

タスクプログラム内における通信処理としては，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階層化されたライブラリモジュールが提供され，タスク・ストリームネットワーク構築のサポートを行う．

3. インタフェース

3.1 基 本 概 念

前述の通り，MegaScript ではタスク間における通信手段として標準入出力を利用している．これにより，各タスクに接続できる入出力ストリームはそれぞれ 1 本ずつに限られる．複数の相手と個別に通信するには宛先タグを付加したメッセージを 1 本のストリーム

に流し，マルチキャストを利用する形で行わなければならない．この場合，無駄な通信が大量に発生し，受信側タスクでメッセージの選別が必要となる．また，MegaScript では，広域に分散した環境での動作を想定しており，通信を行うタスク間が WAN を経由したものとなる可能性もあり，通信回数の増加は全体の処理の低下を招きやすい．以上より，タスクとストリームを直接接続する現在の MegaScript では，タスク・ストリームネットワークの構築に制約が大きく，通信処理が全体に対するボトルネックとなりやすい．

そこで，新たな要素として「インタフェース」を導入する．インタフェースはストリームとタスクの間を橋渡しするものであり，ストリーム・タスクそれぞれに対して都合の良い接続口を提供する．両接続口の差異は，インタフェース内でデータ変換などの処理を行うことで隠蔽する．また，メッセージがインタフェースを通過する際にデータの加工を可能とし，データの管理・制御に関する処理をインタフェース内に記述することでタスクプログラムのブラックボックス化をはかることができる．

これにより，従来のタスク生成における容易性を残したまま，スクリプト側でより適切・効率的な記述を可能とし，既存資産の有効利用や通信処理の最適化を実現する．

3.2 インタフェースの挿入

インタフェースは，タスク・ストリーム間における通信時の仲介を行う機構である．そのため，インタフェースの挿入場所としては，送信側タスクとストリーム入力端の間，ストリーム出力端と受信側タスクの間が考えられる．

送信側タスクとストリーム入力端の間にインタフェースを接続した場合の例を図 3 に示す．送信側タスクは，標準出力を利用してインタフェースにメッセージをわたす．インタフェースは，受け取ったメッセージに対して，問題に応じた処理を適用し接続されたストリームにメッセージをわたす．また，図のように，インタフェースでは複数ストリームの接続を許容する．そのため，インタフェース内の操作により，送信先ストリームの切り替えを行うことができ，適切なストリームに対してメッセージを送ることができる．

次に，ストリーム出力端と受信側タスクの間にインタフェースを接続した場合の例を図 4 に示す．インタフェースは，ストリームからメッセージを受け取り，メッセージに対し問題に応じた処理を適用し，受信側タスクへわたす．受信側タスクは標準入力を利用してそのメッセージを受け取る．また，インタフェースに

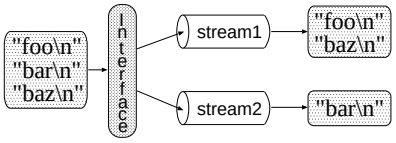

図 3 インタフェース（送信側タスク）

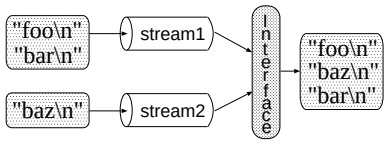

図 4 インタフェース（受信側タスク）

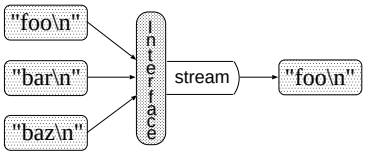

図 5 インタフェース（ストリーム内）

複数のストリームが接続されている場合には，それぞれのストリームからのメッセージをマージして受け取り，受信側タスクへわたす．

ここで，ストリームは入出力端に複数タスクの接続をサポートしており，入力端に複数のタスクが接続されている場合，それぞれのタスクからのメッセージが一つのストリーム上に集まる．そのため，ストリーム内においては別々のタスクからのメッセージをそれぞれ参照することも可能である．そこで，ストリーム上においてもインタフェースを動作できるように拡張し，別々のタスクから受信したメッセージの集合に対し，様々な処理を適用できるようにする．

図 5 に，ストリーム上でインタフェースを実現した場合の例を示す．インタフェースはストリーム内においてメッセージの集合を受け取り，問題に応じた処理を適用し受信側タスクにわたす．例えば，ある条件に合致したメッセージのみ通過させる，といった処理が可能となり，より効率的な通信を実現できる．

3.3 インタフェースの効用

インタフェースの導入により，具体的には以下のような処理が実現可能となる．

3.3.1 複数ストリームの接続

タスク側では従来通り標準入出力のみによる通信を行う一方で，ストリーム側に対しては複数のストリームを 1 タスクに接続できるような拡張が可能となる．インタフェース内ではメッセージの入力・出力先を操作し，接続口の差異を吸収する．

送信側タスクのインタフェースに複数のストリーム

が接続された場合，図3のようにそれぞれのストリームに対して個別にメッセージを送ることができる．

また，受信側タスクのインタフェースに複数のストリームが接続された場合，図4のようにそれぞれのストリームからメッセージをマージして受け取ることができる．

3.3.2 メッセージのフィルタリング

タスクから送られるメッセージに対し，何らかのフィルタリング処理を行ってからストリームに送る，あるいはその逆の操作が可能となる．フィルタリング処理としては，メッセージの加工，しきい値にもとづくデータの選別，宛先タグの付加・削除によるメッセージの制御といったものが考えられる．

3.3.3 送信データ量・回数の削減

タスクから送られてくる複数のメッセージをインタフェース内で一つにまとめることで，通信回数を減らすことができる．このとき，ユーザ側でタスクに応じた細かな指定ができるため，効率のよいまとめ送りが可能となる．

また，送信データに対し，圧縮やバイナリ化などの処理を適用することにより，データ量の削減をはかることもできる．

3.3.4 タスクの再実行

ストリームから見て，タスクプログラムの存在はインタフェース内に隠蔽されている．したがって，ストリーム側には一つのタスクプロセスが長期間存在していると認識させておいて，実際にはタスクプロセスを複数回起動することができる．これを利用すると，一連の処理を行って終了するような既存のタスクプログラムを，インタフェース内で繰り返し実行するように指定することができ，反復改善法のような処理が実現できる．

また，将来的にはエラー検出や例外処理機構と組み合わせることで，異常発生時にタスクを再起動して続行するような記述も可能になる．

4. インタフェースの形態

4.1 インタフェースの導入形態

インタフェースは，実質的にはタスクのラッパプログラムとしての働きをし，ストリーム側に対してはタスクであるかのように振る舞い，タスクに対してはストリームであるかのように入出力を行う．

インタフェースの導入形態としては，タスク・ストリームの接続口の差異を吸収するだけでなく，フィルタリング処理やエラー時の回復などより高度な処理を実現できるように，インタフェース内に任意の処理をユーザが記述できるようにする．したがって，ユーザが記述したインタフェース関数を実行する機構とする．

インタフェースではメッセージの加工・抽出処理が多用されるため，記述言語としてはパターンマッチングなどが容易で MegaScript のベース言語でもある Ruby を用いる．インタフェースでの処理は，多くの場合，1メッセージにつき数十ステップ程度でかつコンスタントオーダーであると期待できるため，多少実行効率の低いスクリプト言語でも問題ないと考えられる．また，Ruby では C などで記述されたコードを呼び出すこともでき，速度が必要な場合にも対応できる．

4.2 インタフェース・プログラミング

インタフェースは，ユーザにより Ruby のメソッドとして実装される．ユーザはインタフェース・クラスを継承し，入出力処理メソッドをオーバライドする形でタスクに応じたインタフェースを実装する．

インタフェースを実現する上で必要となる機能は，Ruby の API モジュールとして提供する．API には，インタフェースの定義・接続など基本操作を行うものと，タスク・ストリームとの入出力部などインタフェースの記述をサポートするものがある．

基本 API としては，インタフェースの定義やタスク・ストリームとの接続指定など，タスク・ストリームネットワークへのインタフェース組み込みをサポートするものを用意する．また，インタフェース記述用の API としては，タスク・ストリームとの入出力処理部分などを用意し，インタフェース実装者の負担を減らすと同時に，タスク・ストリームの実装方式を隠蔽する．

また，複数ストリームの接続やまとめ送りといった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インタフェースを用意し，インタフェース・ライブラリとして提供する予定である．これにより，ユーザは既存のインタフェースから選んで使うか，より高度なものを自作するか，選択できる環境を目指す．

5. インタフェースの実装

5.1 MegaScript ランタイムにおけるストリーム通信

ストリーム通信実現のための処理として，MegaScript ランタイムは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を生成する．これらのスレッドは，タスクプロセスへのデータ書き込み用と読み出し用の2本のパイプを経由して，タスクプロセスとデータのやりとりを行う．

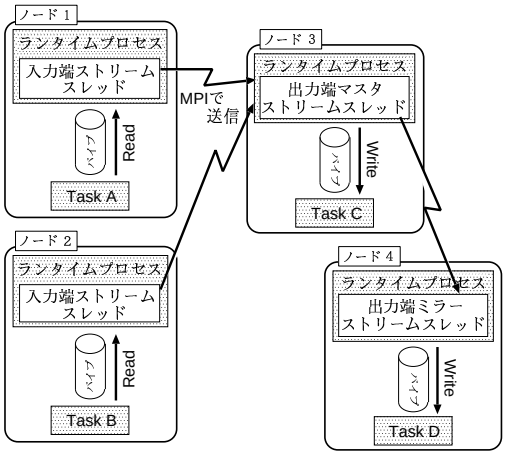

図 6 ストリーム通信例

図 6 に送信側タスクが 2 つ，受信側タスクが 2 つ，それぞれ異なるノード上に生成され，1 本のストリームに接続されている場合の例を示す．

入力端ストリームスレッドは，接続されているタスクプロセスに繋がっているパイプからデータを読み出し，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．このとき，各出力端ストリームスレッドにはマスタとミラーという関係があり，各タスクに送られるメッセージの並びの整合性をとるため，メッセージの送信は出力端ストリームスレッドのマスタに当たるスレッドにのみ行われる．

一方，出力端ストリームスレッドにおいて，マスタに当たるスレッドは入力端ストリームスレッドから送られてきたデータが格納されているキューからデータを受け取り，それをミラーに当たる出力端ストリームスレッドと自ノード内のタスクプロセスに繋がっているパイプとに対して送信または書き込みする．ミラーに当たるスレッドはマスタスレッドから送られてきたデータが格納されているキューからデータを受け取り，自ノード内のタスクプロセスに対し，パイプを経由してデータをわたすことで通信が完了する．

5.2 インタフェース起動部分の実現

インタフェースの実装方法の一つとしては，各インタフェースを別プロセスとし，タスクプロセスと同時に生成する方法が考えられる．しかし，この方法ではプロセス数が倍増してしまい，処理効率の低下を招く．ストリーム側の処理は MegaScript ランタイム上で動作しているため，これを利用し，ストリームがタスクプロセスと通信する際にインタフェースのコードを実行できるようにする．

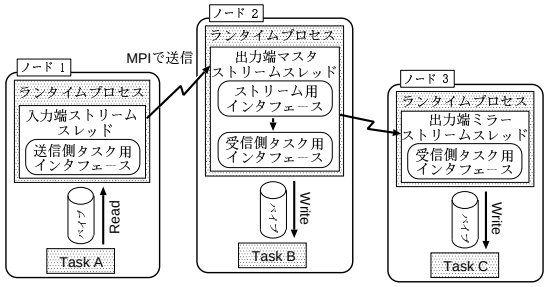

図 7 インタフェース実装例

そこで，インタフェース起動部分の実装として，MegaScript ランタイムの入力端ストリームスレッド内に送信側タスクのインタフェースを，出力端ミラーストリームスレッド内に受信側タスクのインタフェースを，また出力端マスタストリームスレッド内ではストリーム用インタフェースを実行できるように拡張する．インタフェースの実装にあたっては，Ruby 拡張用の API を利用した．

図 7 に，送信側タスクが一つ，受信側タスクが 2 つ，それぞれ異なるノード上に生成され，インタフェースを介して 1 本のストリームに接続されている場合の例を示す．

まず，送信側タスクのインタフェースを実装するために，入力端ストリームスレッドを拡張する．このスレッドでは，タスクプロセスからパイプを経由してデータを受け取り，出力端マスタストリームスレッドへ送信している．そのため，タスクプロセスからデータを受け取った際にインタフェースを起動するようにし，データの処理を行う．

次に，ストリーム用インタフェースの実装として，出力端マスタストリームスレッドを拡張する．このスレッドには，各入力端ストリームスレッドからのデータが集まるため，受信側タスクへ送信される前にインタフェースを起動するようにし，データ群に対して処理を行う．

また，受信側タスクのインタフェースの実装として，出力端ミラーストリームスレッドを拡張する．このスレッドでは，出力端マスターストリームスレッドからデータを受け取り，タスクプロセスへパイプを経由してデータを書き込む．そのため，タスクプロセスへデータを送信する前にインタフェースを起動するようにし，データの処理を行う．また，マスタスレッドが動作しているノード上のタスクに関しては，マスタスレッド内でストリーム用インタフェースの処理が完了した後，受信側タスク用のインタフェースを起動するようにし，データの処理を行う．

以上により，ランタイム内でのインタフェースの起動を実現する．

6. 予備評価

インタフェースを Ruby メソッドとして実装する際の問題点として，MegaScript ランタイムからインタフェースを呼び出す際に生じるオーバヘッドがある．

そこで，今回インタフェース起動部分を実装した MegaScript ランタイムを用いて，インタフェース実行時の処理時間を測定し，インタフェースを実装していない MegaScript ランタイムとの比較を行った．評価用のタスクとして文字列を繰り返し送信するタスクと，その文字列を受信するタスクを用いる．これを同一ノード上に配置した場合と，異なるノード上に配置した場合とで測定を行った．オーバヘッド測定のため，インタフェース内では処理を行わないものとする．評価環境として，Pentium4 2.4GHz の PC を Gigabit Ethernet で接続した PC クラスタを利用した．

まず，同一ノード上で実行した場合，インタフェースを実装した MegaScript ランタイムでは 2 倍程度処理に時間がかかった．また，異なるノード上で実行した場合においては，インタフェースを実装した MegaScript ランタイムでも数パーセントしか処理時間が低下しなかった．

次に，ある程度実用的な処理を行った場合の処理時間の比較として，インタフェースにおいて送られて来た文字列の簡単な操作を行った．同一ノード上で実行した場合，インタフェースを実装した MegaScript ランタイムでは 2.3 倍程度処理に時間がかかった．ただし，インタフェース内の処理の規模により，処理時間は大きく増減する．また，異なるノード上で実行した場合においては，先程と同様，インタフェースを実装した MegaScript ランタイムでも，数パーセントしか処理時間が低下しなかった．

以上より，通信を行うタスク同士が同一ノード上にある場合，通信にかかる時間に比べインタフェース起動時のオーバヘッドが大きく，処理時間の比率には大きな差が生じる．

これに対し，通信を行うタスク同士が異なるノード上に配置された場合，処理時間に関しては数パーセント程度の差しか生じていない．これは，異なるノード間においては通信がドミナントとなるため，インタフェース起動にともなうオーバヘッドは通信処理によりほぼ隠蔽されてしまうことがわかる．

7. おわりに

本稿では，タスク並列スクリプト言語 MegaScript におけるストリーム通信機能の改良として，新たに導入したインタフェースについて述べた．

今回，インタフェース機構実現の第一段階として，MegaScript ランタイムの入力端・出力端ストリームスレッドにインタフェース起動部分を実装した．評価結果より，インタフェース組み込みに伴うオーバヘッドは，実際の処理回数を考慮するとごく小さなものであり，実用上は問題ないといえる．

評価を行ったインタフェースは多くの機能が未実装であり，今後，それらを機能性・利用容易性といった点を考慮して実装し，それらを含むオーバヘッドの測定，実用的なアプリケーションにおける通信効率の検証などを行う必要がある．

また，大規模な処理においては，複数のインタフェースの実装が必要であり，ユーザがタスクに応じたインタフェースを容易に実装できるようにすることが重要となる．そのため，インタフェース記述時に容易に利用できるような API 関数群，ライブラリ構造の提供が必須となる．

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) 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).
3) 大塚保紀, 大野和彦, 中島浩: タスク並列スクリプト言語 MegaScript によるタスク動作モデル記述, 情報処理学会研究報告, HPC-95, pp. 113–118 (2003).
4) 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)