ハイパフォーマンス
コンピューティング 82−26
（2000. 8. 4）

グローバルコンピューティングのための ストリーム計算実行時システム

関口 真良 首藤 一幸 杉野 博文 村岡 洋一
早稲田大学

広域ネットワーク上に分散した計算機群で高性能計算を行う場合、計算機間の通信にともなう大きな遅延が問題となる。光速はたかだか300km/ミリ秒なので、遅延の削減には限界がある。そこで、広域並列計算に適したプログラミングおよび実行モデルとして、ストリーム計算を提案する。これは、パイプライン処理を計算機群で行うものである。アプリケーションが複数のステージに分割可能ならば、通信遅延が大きい状況でも高いスループットが期待できる。我々は、モデルの有効性を確認するために、実行時システムおよびユーザプログラムのテンプレートを実装した。その上でサンプルアプリケーションを作成して性能を評価したところ、良好な速度向上と、遅延への高い耐性が確認できた。

Stream Calculation: a runtime system for global computing

Masayoshi SEKIGUCHI Kazuyuki SHUDO
Hirobumi SUGINO Yoichi MURAOKA

Waseda University

High-performance computing with the distributed computational resources suffers large communication latency. Most of the latency is unavoidable because the light speed is only 300km/msec. We therefore propose "Stream Calculation" as a programming and execution model, which employs the pipelining technique and suits distributed parallel processing. Applications achieve high performance if they can be divided into stages. We implemented a reference runtime system and templates for user programs to confirm the usability of the model. Evaluations of a sample application showed good speed-up and that its throughput stands various communication latency.

1 はじめに

近年、グローバルコンピューティングシステムが、Globus[8],NetSolve[7],Legion[9], Ninf[6][4], Ninflet[2][3] など複数提案されている。ただし、分散環境が広域になればなるほど、避け難い通信遅延が発生する。通信遅延をゼロにするのは不可能であり、広域分散計算を行う以上はこの通信遅延を覚悟し、通信遅延に影響を受けにくいアルゴリズムや実行モデルが必要である。

そこで、通信遅延を隠蔽できる実行モデルとしてストリーム計算 (**Stream Calculation**)[1] を提案する。ストリーム計算は、広域ネットワーク上の複数の計算機でパイプライン処理を行うことにより、通信遅延が大きくても高いスループットを達成できる。本稿では、まず第2章で本システムの仕組みについて述べる。第3章では、実際の実装について述べる。第4章では、実装結果とそれについての考察について述べる。また、第5章で、本システムの今後の課題について述べる。

2 ストリーム計算

広域分散計算において、処理を複数のステージに分割し、分割されたそれぞれの処理を各計算機に配置し、計算機間でデータを流すことによってパイプライン処理を行う。データストリームに沿って処理を進めるため、この実行モデルをストリーム計算 (**Stream Calulation**) と呼ぶこととする (図1)。ストリーム計算では通信頻度によらず通信遅延を隠蔽することが可能である。

現実装では、分岐の無いパイプライン処理のみだが、将来的にはデータ並列性を生かすために、途中で経路を分割することも考えている。

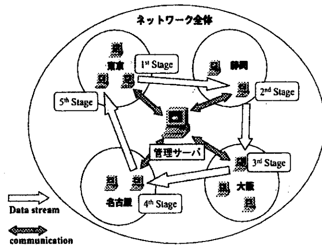

図 1: ストリーム計算

3 実装

Java 言語 [5] で実装した。また、システムを構成する各ソフトウェア間の通信にはJavaRMIを用いた。また、各ステージ間 (計算機間) の通信は、TCP/IP プロトコルの Socket 通信を用いた。これは、Java の持つセキュリティや機種非依存性など、ネットワークの先から入手したコードの実行に向いた性質が本システムでは必要となるためである。また、最近では、性能も C などに比肩し得るようになったことも理由のひとつである

3.1 システムの構成

以下、システムの構成について述べる。

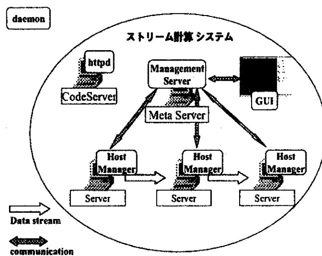

図 2: システムの構成

3.1.1 CodeServer

HTTP、FTP サーバである。実行プログラム (クラスファイル) がおいてあり、ここからユーザが指定するクラスファイルをダウンロードし実行する。

3.1.2 HostManager

各計算機上のデーモンである。HostManager が起動してる計算機 (以下 SC Server と呼ぶ) は、パイプラインの各ステージとしてユーザプログラムを処理する。UserInterface からの指示に従って、クラスファイルを CodeServer からダウンロードし実行するとともに、各計算機の状態を Management Server に報告する。

3.1.3 UserInterface(GUI)

ユーザは、これを利用して、ダウンロードするプログラムの決定、ルーティングの指定、および、実行開始および停止の指示をすることができる。ここでルーティングとは、各計算機にどのような順序でデータストリームを流すかということである。

ユーザが直感的に理解しやすいような、GUI を作成した。また、ルーティングの決定、クラスファイルの選択ともドラッグ&ドロップで操作可能であり、複雑な知識を必要としない (図 3)。

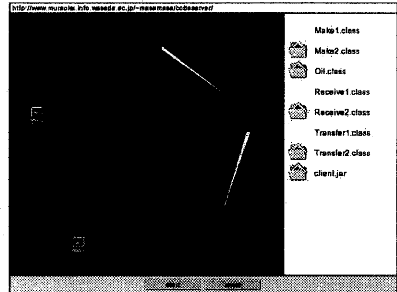

図 3: GUI

3.1.4 Management Server

すべての HostManager の管理を行うための、デーモンである。システム中にただ一つ存在する。

HostManager から送信される情報をもとに、各計算機の性能情報を把握しておき、UserInterface を通してユーザに情報を提供する。また、UserInterface からの指示を受け、該当する HostManager のリモートメソッドを起動し、クラスのダウンロード、実行の開始、停止などを行う。その他に HostManager から送信されるログの収集等も行なう。

3.2 デーモンの起動から実行まで

本節では、デーモンの起動から実行までのプロセスについて述べる。

以下は計算機管理者が行う設定である。

3.2.1 Management Server の起動

ある 1 台の計算機上で Management Server を起動する。(図 4(a))。

3.2.2 HostManager の起動

Management Server が起動している状態で、各計算機上で HostManager を起動する。HostManager は、起動時に Management Server の場所 (ホスト名) を指定する必要がある。HostManager は、起動した後に自身の計算機の情報 (CPU クロック, メモリ, ネットワーク帯域,etc) を Management Server に登録し、Management Server からの指示待ちの状態になる (図 4(b))。以上が、計算機管理者が行うべきことである。

ここからは、実際にシステムを使用したいユーザが行う作業である。

3.2.3 GUI の起動

ユーザプログラマは使いたい計算機の数だけ分割されたプログラムを用意し、コンパイルされたコードを CodeServer 上に置く。その後、GUI を起動する。GUI はシステムから切り離しが可能なので、必要なときだけ立ち上げればよい。GUI の起動と共に、Management Server は保持している各計算機の情報を GUI に送信し、GUI はその情報をもとに現在のシステムの状況を画面に表示する。

3.2.4 ユーザープログラムの実行

ユーザは GUI 上に表示される内容を見て、ルーティングと、各計算機に配置するクラスファイルを決定し、ドラッグ&ドロップで指定する。GUI はその内容を Management Server に伝え、Management Server では GUI からの指示にもとづいて各計算機上のデーモン (HostManager) のリモートオブジェクトのメソッドを実行する (図 4(c))。各計算機ごとに、コードのダウンロード (図 4(d)) とデータストリームの接続が完了したら、Management Server が実行開始の命令を出し、実際の処理が開始される (図 4(e))。

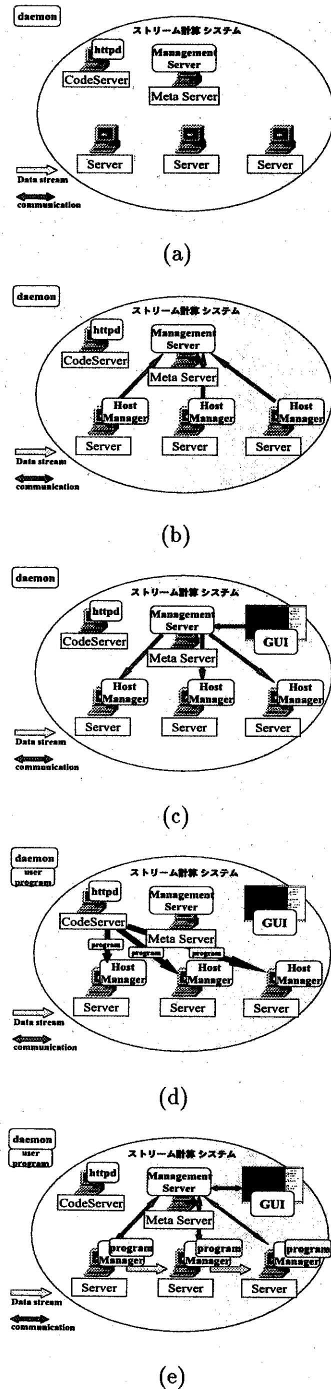

図 4: システム運用の流れ

3.3 資源の獲得法

広域分散計算を行うには、多数の人に計算機を提供してもらう必要がある。そのためには、高いセキュリティを保証する必要がある。現状では、Javaのセキュリティ機能に頼っているが、独自のセキュリティマネージャを用意し、さらにセキュリティを強化する必要がある。

また、各ユーザが使用するデータに関してはWebサーバから動的にダウンロードして使用する。

3.4 プログラムの並列化手法

現状ではプログラムの分割は、ユーザが手動で行わなければならない。将来的には、分割を自動で行うトランスレータを開発する予定である。

ストリーム計算で利用可能なアプリケーションは、以下のような特徴をもつ必要がある。

- ステージに分割できる
- データが連続して流れてくる
- 計算量のオーダが高い

3.5 プログラミングモデル

本システムはユーザに、送受信用のストリームを与え、実際の通信のプロトコルはユーザが決めるという形態をとる[1]。Javaの`ObjectOutputStream`, `ObjectInputStream`を使用すれば、任意のシリアライズ可能なオブジェクトの送受信が可能なため、ユーザの負担は比較的小さくて済む。

また、実行開始時の処理や、UserInterfaceからの入力による強制終了時の処理などは、提供する抽象クラス`SCClient`内で定義しており、ユーザがこれらの処理に頭を悩ます必要はない。ユーザは、`SCClient`クラスを継承し、抽象メソッドrun()の内部に実際の処理の部分だけを記述すれば良い。

その他にも、連続する静止画像に対する処理など、よく使われると予想される計算モデルにはクラスライブラリをあらかじめ用意しておき、ユーザに提供することを考えている。

3.6 計算と通信のオーバラップ

`ObjectOutputStream`, `ObjectInputStream`はブロッキング通信を行うので、そのままでは計算と通信のオーバラップは行われない。これを解決するには、送信、受信、そして計算の3つを全て別スレッドで動かせば良い。計算スレッドと、送信スレッドや受信スレッドとの通信には、内部バッファを用いることで対応する。受信スレッドは、受信可能なデータがある限り受信し、それを内部バッファに書き込む。計算スレッドは、内部バッファからデータを取り出し処理を行い、送信するデータを内部バッファに書き込む。送信スレッドは、内部バッファにデータがある限りそのデータを送信する。このようにすることで、送信または受信用のスレッドが入出力待ちをしている間も計算スレッドが動作し、効率の良い実行が可能である。

[1] 各ステージ間の通信はTCP/IPプロトコルであるが、それより上位のプロトコルについては、ユーザに委ねられる。

```
import java.io.*;
import jp.ac.waseda.info.muraoka.masamasa.stream.core.*;

public class SCExample extends SCClient {

    public SCExample(InputStream input,
                     OutputStream output){
        super(input,output);
    }
    ...
    public void run(){
        ...
        while(条件を満たすまで){
            ...
            実際の処理
            ...
        }
    }
}
```

図5: プログラミング例

ただし、このような複雑なプログラミングをユーザに行わせるのは負担が大きいため、クラスライブラリ(`StreamingOutputStream`, `StreamingInputStream`)を用意した。ユーザは、これらのクラスを`ObjectOutputStream`, `ObjectInputStream`と全く同様に使用することで、何も意識することなく上記のマルチスレッドプログラムを書いたことになる。(図6参照)

4 評価

予備評価としてLAN環境での実験を行った。

4.1 アプリケーション

本システムに適用できるアプリケーションとして、動画のフィルタリングを取り上げる。油絵タッ

```
import jp.ac.waseda.info.muraoka.masamasa.stream.io.*;

public class Stream {
    private OutputStream output;
    private StreamingOutputStream out_stream;
    ...
    public void run(){
        out_stream =
            new StreamingOutputStream(output);
        ...
        while(条件を満たすまで){
            ...
            out_stream.writeObject(sendData);
            ...
        }
        out_stream.flush();
        out_stream.close();
    }
}
```

図 6: StreamingOutputStream の使用例

チにするフィルタリングを行なった。動画のデコードにはJMF(Java Media Framework) [10] を利用した。動画をデコードし、各フレームに対しフィルタリングをかけ、そしてフィルタリングされた絵を表示するという3段階の処理である。よって、ステージ分割は3ステージとした。ただし、各ステージの処理時間が均等になるようにフィルタリングの部分を前後のステージに分割した。

4.2 LAN 環境での実行結果

実験は3台の同性能のPC(DELL-OptiplexGX1 Pentium350MHz メモリ:384MB) で行った。各マシンは 100Base/TX の Ethernet でつながれている。OS は RedHat-Linux6.2J (kernel2.2.14, glibc2.1.3)、Java の実行系にはIBM社の JDK1.1.8[11] を使用した。

100,200,300,500 フレームを実行するのにかかる処理時間を測定した。表1に、3台を使用したストリーム計算システムでの処理時間と、1台での実行時間との比較を載せる。単位は秒である。

1台での測定、3台での測定とも同じハードウェア、ソフトウェアを用いているので、マシンによる性能の差異は無いものとし、速度向上率 (= $\frac{1\text{台での処理時間}}{3\text{台での処理時間}}$)、および、速度向上率を PC の台数3で割った相対実行速度を求めると表 2のようになる。

フレーム数	ストリーム計算での処理時間 (単位:秒, 括弧内は遅延)	1台での処理時間 (単位:秒)
100	167.70(0.208)	457.18
200	317.46(0.249)	907.05
300	478.26(0.226)	1355.84
500	792.32(0.292)	2262.33

表 1: ストリーム計算での処理時間

フレーム数	速度向上比	相対実行速度
100	2.72	0.91
200	2.85	0.95
300	2.83	0.94
500	2.85	0.95

表 2: 速度向上比と相対実行速度

4.3 遅延の大きい環境での性能低下予測

本システムは遅延の大きい環境での実行を想定している。本節では、遅延の大きい環境での性能低下について考察する。上記の実行結果では、1フレームあたりの処理時間はおよそ 1600 ミリ秒である。広域分散計算環境で想定される遅延の大きさは大きくても 500 ミリ秒程度なので、通信時間に比べて計算時間がはるかに大きく、通信遅延は隠蔽される。表3に、擬似的に通信遅延 (単位はミリ秒) を増加させた際の、100 フレームをストリーム計算で処理する実行時間 (単位は秒) と、相対実行速度を載せる。

遅延 (msec)	ストリーム計算での処理時間 (単位:秒, 括弧内は遅延)	相対実行速度
+10	169.14(0.29)	0.90
+50	168.73(0.25)	0.90
+100	170.44(0.27)	0.89
+200	170.76(0.34)	0.89
+500	170.77(0.66)	0.89

表 3: 遅延の大きさによる性能低下

以上の結果より、通信遅延の増加によるパフォーマンスの低下はほぼ無いと考えられる。

5 今後の課題

以下の問題が明らかになった。これらは今後の課題である。

5.1 オブジェクトシリアライザの性能

画像のデータを送受信する際に、Java の **java.awt.Image** オブジェクトはシリアライズ可能でないため、独自のシリアライザを作成して対応したが、シリアライザの処理時間がシステムのオーバーヘッドのほとんどをしめた。本システムは、データの送受信量が多いため、シリアライザの性能が非常に重要になる。

5.2 動的ルーティング

システムの運用中に、どこか一箇所の計算機の負荷が急激に増大し処理速度が落ちた場合、システム全体の処理速度が落ちてしまう。よって、実行中にいずれかの計算機の負荷が増大した場合には、負荷の少ない計算機にルーティングを変更する動的ルーティングが必要である。またその際、ステージの他の計算機への移送を行う機能が必要となる。

5.3 プログラミングモデル

現在のプログラミングモデルでは、プログラマがステージ間のデータのやりとりまで記述する必要がある。プログラマの負担は小さいとは言えない。アプリケーションをより易しく書ける方式や、そのためのプログラム変換ツールの開発が課題である。

6 まとめ

本稿では、広域分散計算環境における通信遅延を隠蔽するための実行モデルとして、パイプライン処理を利用したストリーム計算を提案した。

Java を用いてストリーム計算システムを実装した。GUI 上のドラッグ&ドロップでの操作、通信と計算のオーバーラップ, 実行開始停止時の処理のためのクラスライブラリの提供など、ユーザーの負担を軽減するための仕組みを用意した。

また、LAN 上での予備評価を行い、本システムで実際に並列化を行った結果について述べた。今後の課題として、適用するアプリケーションの開発、動的ルーティング、などがあげられる。

参考文献

[1] Hirobumi Sugino , Kazuyuki Shudo , Masayoshi Sekiguchi , Yoichi Muraoka , *"A Model for Stream Calculation",The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications(PDPTA2000),Jun 2000.*

[2] 高木浩光, 松岡聡, 中田秀基, 関口智嗣, 佐藤三久, 長嶋雲兵, *"Java による大域的並列計算環境 Ninflet",JSPP'98* 論文集,*pp135-142,Jun 1998.*

[3] 大久光崇, 高木浩光, 松岡聡, 小川宏高, *"Java を用いた広域並列計算システム Ninflet 上の通信クラスライブラリの実現"*, 情報処理学会研究報告 *98-HPC-72,1998.*

[4] 中田秀基, 高木浩光, 松岡聡, 長嶋雲兵, 佐藤三久, 関口智嗣, *"Ninf による広域分散並列計算",JSPP'97* 論文集,*pp281-288,May 1997.*

[5] James Gosling , Bill Joy , Guy L. Steele Jr. , *"Java Language Specification",1996.*

[6] Ninf(Network based Information library for High Performance Computing), *http://ninf.etl.go.jp/.*

[7] NetSolve Homepage, *http://www.cs.utk.edu/netsolve/.*

[8] The Globus Project, *http://www.globus.org/.*

[9] Legion: A Worldwide Virtual Computer, *http://www.cs.virginia.edu/ legion/.*

[10] Java(TM) Media Framework API HomePage, *http://java.sun.com/products/javamedia/jmf/index.html.*

[11] Java developer kits, *http://www.ibm.com/java/jdk/118/linux/.*