九州大学学術情報リポジトリ
Kyushu University Institutional Repository

MPIプログラムのための並列ストリーム入出力関数と並列パイプ機構

天野, 浩文
九州大学情報基盤センター

津村, 忠蔵
（株）富士通情報通信システムズ

https://doi.org/10.15017/4777971

出版情報：九州大学情報基盤センター年報. 1, pp.1-8, 2001-10. 九州大学情報基盤センター
バージョン：
権利関係：

情報基盤センター年報第1号（2001年）
九州大学情報基盤センター

MPI プログラムのための並列ストリーム入出力関数と並列パイプ機構

Parallel Stream Input/Output Functions and Parallel Pipe Mechanism for MPI Programs

天野 浩文 †　　津村 忠蔵 ‡
Hirofumi Amano†　　Tadamasa Tsumura‡

† … 九州大学情報基盤センター
† … Computing and Communications Center, Kyushu University
‡ … (株) 富士通情報通信システムズ
‡ … Fujitsu Information Network Systems Limited

要旨　MPI (message passing interface) はハイエンドのスーパーコンピュータから安価な PC クラスタにいたるまでのさまざまな並列計算機上で手軽に並列処理を行う手段の一つとして広く用いられるようになってきている．しかし，大規模数値計算を行う上で不可欠な大量データ入出力機能については，まだ検討すべき点も多い．現在開発中の並列ストリーム入出力関数を用いると，各プロセスは，必要なデータが全体で共通のファイルの中にどのように配置されるかを計算することなく，それぞれ自分専用のファイルにアクセスするかのように記述できる．本稿では，並列ストリーム入出力関数と並列パイプ機構の概要と，その応用の可能性について述べる．

Abstract　MPI (message passing interface) is now widely used as an easy parallel programming tool on various parallel machines including high-end supercomputers and low-price PC clusters. However, it leaves much to be desired in input/output functions for large data, which is vital in large-scale numerical computation. With new parallel stream input/output functions, each process can access its own file so that it need not calculate where the necessary data is allocated in a single file shared among all processes. This paper presents an overview of the parallel stream input/output functions now under development and their parallel pipe mechanism. It also discusses their possible applications.

1　まえがき

ハードウェア技術の進歩により，比較的安価な PC や WS をネットワークによって結合し，これを仮想的な分散メモリ型並列計算機として利用することが容易になってきている．また，このような PC クラスタや WS クラスタのための並列プログラミングを行う方法として，プロセス間のメッセージ通信用ライブラリ **MPI** (message passing interface)[5] が急速に普及しつつある．

MPI では，Fortran や C といった通常の逐次型プログラミング言語の中からプロセス間通信のための関数を呼び出すことにより，並列プログラミングを行う．実行時には，これらの関数を埋め込まれたプログラムのコピーが各プロセッサに配布され，それらが MPI プロセスとして並列に実行される．各プロセスは自分のプロセス ID を取得することができるので，すべてのプロセスが同一の処理を分担するだけでなく，プロセスごとに内容の異なる処理を実行することも可能である．

これらの関数のインタフェースが国際的な標準として定義されており，プログラマは使用する並列計算機・環境の差異を意識することなく，移植性の高い並列プ

ログラムを開発することができる．

一方，大規模な科学技術計算を行うためのプログラムでは，大量の数値データを読み込んで計算し，大量の演算結果を書き出すことが多い．この入出力の処理を高速化するための方法として，複数のディスク装置を内蔵した RAID (redundant array of inexpensive disks)[7] を従来のディスクの代わりに用いる方法や，分散メモリ型並列計算機が有する複数のディスクでソフトウェア的に RAID を実現する手法がある．しかし，これらは単一のプロセスからの大容量入出力を高速化するのには適しているが，複数のプロセスが発する多数の入出力要求の全体を同時に処理するのにはあまり適していない．

また，並列に実行中の各プロセスからの集団的 (collective) な入出力要求をディスクの台数に見合う数のディスクアクセス要求にまとめることができるような並列ファイルシステムについても研究・開発が進められている [2]．並列ファイルシステムに対する集団型入出力は，複数のディスクの上にストライピングされた仮想的な単一ファイルに対して各プロセスが入出力要求を発する．ただし，この仮想単一ファイル内でのデータのオフセット値は各プロセスが個別に計算する必要があり，プログラムが煩雑になる．

そこで，本研究では，MPI プログラムからの集団型入出力を簡単に記述でき，複数のディスクを有する分散メモリ型並列計算機上で効率的に処理するための**並列ストリーム入出力機構**の開発を行っている．各プロセスが仮想的に自分専用のストリームをオープンして使用するため，利用者プログラム側での面倒なオフセット計算は不要である．また，集団型入出力要求をディスクの台数に見合うだけのアクセス要求にまとめるため，多数の入出力要求が発生してもボトルネックが発生しない．

本論文では，このような MPI プログラムのための**並列ストリーム入出力関数**，および，これを用いて複数の並列プログラム間でディスクを経由せずに並列データ転送を可能にする**並列パイプ機構**について述べる．

2 MPI と集団型入出力機能

2.1 MPI プログラムの基本的な構成

MPI プログラムは，おおむね，図 1 に示すような形式をとる．

MPI プロセスの初期化のための `MPI_Init()` と終了処理のための `MPI_Finalize()` の呼び出しの間が並列実行部分となる．この部分のコピーが各プロセッサに配布され実行される．

```
#include <stdio.h>
#include <mpi.h>
...  /* 大域変数宣言等 */
void main(int argc, char *argv[])
{
  ...  /* 変数宣言等 */
  MPI_Init(&argc, &argv);  /* 初期化 */
  /* ------------------------- */
  /*                           */
  /*        並列実行部分       */
  /*                           */
  /* ------------------------- */
  MPI_Finalize();  /* 終了処理 */
}
```

図 1: MPI プログラムの基本的な構成

MPI では，各プロセスが自分の MPI プロセスとしての ID（ランク）を取得するための関数や MPI プロセスの総数を取得する関数といった各プロセスが自分の情報や同時に動作している MPI プログラム全体の情報を取得するための関数や，宛先の MPI プロセスを指定してメッセージを送信するための関数，送信元を指定してメッセージを受信する関数など，多様なプロセス間通信の関数が用意されている．これらの関数は `MPI_Init()` と `MPI_Finalize()` の間でのみ使用できる．

2.2 並列プログラムにおける集団型入出力

並列プログラムにおいてデータ入出力を行う方法とタイミングには，以下のようなものがある．

1. **逐次型**（従来型）

   従来の逐次型プログラムのために用意されている入出力インタフェースをそのまま流用する．

   (a) 並列実行を行う部分の前後の逐次動作部分で動作するプロセス，あるいは，並列実行部分で単一のプロセスが，他のプロセスを代表して入出力を代行する（図 2 の (a)）．

   (b) 並列実行部分の中で各プロセスが自分の必要とする入出力を自由なタイミングで個別に行う（同 (b)）．

2. 集団型

並列実行部分で各プロセスが同一の関数を同期して呼び出し，それぞれが個別のデータを操作する（同 (c)）.

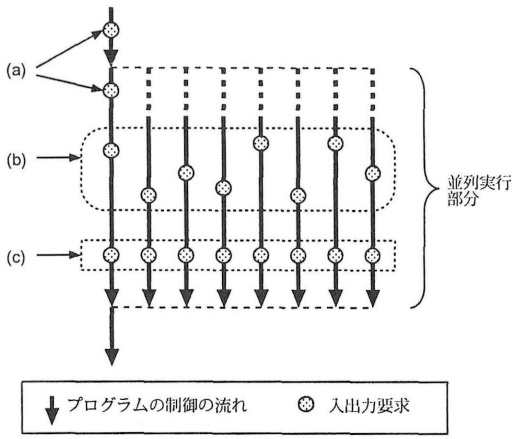

図 2: 並列プログラムにおける入出力

従来型入出力インタフェースは，以下のような欠点を持っている.

- 単一のプロセスに入出力を代表させると，転送性能が上がらず，また，利用者プログラム側でデータ分配・収集を制御する必要がある.
- 逐次型プログラム用入出力インタフェースはそもそも並列プログラムによる使用を想定していない.
  - 同一ファイルの異なる部分に同時にアクセスする機能がない（不連続なデータに対する一括アクセスの機能がない）.
  - 複数のプロセスから発生する多数のアクセス要求をまとめて効率的なファイルアクセスを行う機能を持っていない.

集団型入出力は，これらの欠点を克服しようとするもので，各プロセスがほぼ同量のデータに対し同様の計算を行うデータ並列型プログラム [4] においては非常に有効な機能である．MPI の最新版である MPI-2[6] においては，このような集団型入出力機能が取り入れられている.

MPI-2 における集団型入出力インタフェースは，MPI の集団型通信機能を入出力用に拡張したもので，図 3 に示すように各プロセスが単一のファイル内でのオフセットを計算して入出力を行う.

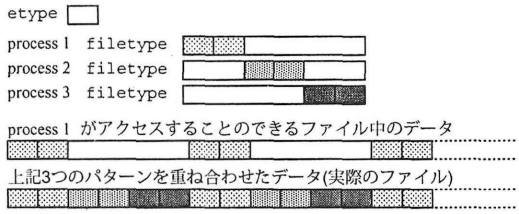

図 3: MPI プログラムにおける集団型入出力

図 3 中の `etype` は，書き込もうとするデータの型とサイズを表している．各プロセスは，プロセス間通信の際に使用する派生データ型と同様に，各プロセスからのファイルデータの見え方（「見える」部分と「見えない」部分）を `filetype` として定義する．図の例は，各プロセスが `etype` 型のデータを 2 個ずつ入出力する場合を示している．これらはそれぞれ「ずれた」位置にあり，これらを重ね合わせると，入出力の対象となる連続データファイルとなる.

このような形でファイルと MPI プロセスの間のデータの受け渡しが行われる場合，このための入出力関数は同期して呼び出されるので，関数側でデータの分配・収集を行うことにより，ファイルアクセスを効率化できる可能性がある．ただし，MPI-2 の規格では，この機能をどのように実装するかは規定されていない.

3 並列ストリーム入出力関数と並列パイプ機構

3.1 並列ストリーム入出力関数

現在開発中の並列ストリーム入出力インタフェースは，集団型入出力機能を有しており，以下のような特徴をもっている.

1. 各プロセスは，それぞれが仮想的な自分専用のストリーム（メンバストリーム）にアクセスする（メンバストリームは MPI プロセスの数だけ存在する）.
2. 各プロセスが同期して入出力関数を呼び出す際は，同じ型のデータを同じ個数だけ，自分のメ

ンバストリームに対してアクセスする（各メンバストリーム内のオフセットは，すべて同じサイズだけ変動する）．

3. メンバストリームは，ディスク上の実ストリームに自動的にマッピングされ，利用者プログラムはこのマッピングを意識する必要はない．

4. 入出力関数は全プロセスが同時に呼び出し，それらの関数の中で自動的にバッファリングや，必要ならばデータの並べ替えを行い，ファイルアクセスを効率化する．

並列ストリーム入出力インタフェースの概念図を図 4 に示す．

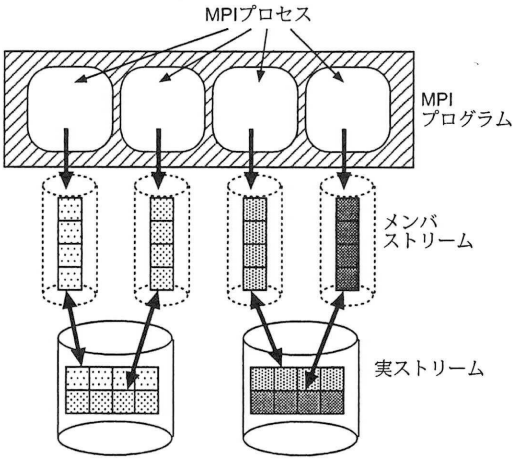

図 4: 並列ストリーム入出力の概念図

3.2 並列パイプ

ある MPI プログラムが出力する並列ストリームを，同数のプロセスが動作する次の MPI プログラムの入力に供給することも可能である．これが並列パイプ機構である．並列パイプにより，ファイルを経由することなく並列プログラム間の並列データ転送が可能になる．

並列パイプの概念図を図 5 に示す．

3.3 並列ストリーム入出力ライブラリの実装

前節までに述べた機構は，並列ストリーム入出力関数のライブラリとして実装される．各関数は従来のス

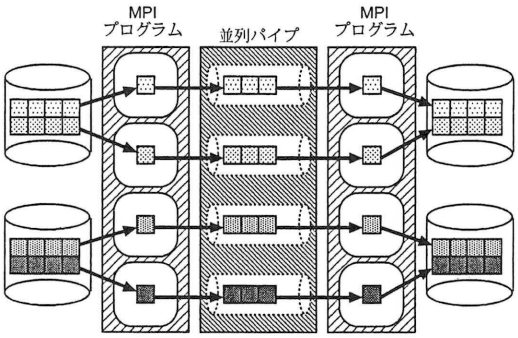

図 5: 並列パイプの概念図

トリーム型入出力関数 `fopen()`, `fclose()`, `fread()`, `fwrite()` などを並列用に拡張したもので，逐次版とほぼ同様のインタフェースを有している．関数本体も MPI ライブラリ関数を用いて記述されている．

各関数は集団型インタフェースで呼び出されることを前提として実装されており，各プロセスから呼び出されると自動的に以下のような処理を行う．

1. ファイルのオープン・クローズ
   ローカルにデータディスクのあるプロセッサで動作しているプロセスが代表してファイルのオープン・クローズを行う．ディスクを持たないプロセスは，自分の入出力を代行するプロセスの ID を取得する．

2. データアクセス
   ローカルにデータディスクのあるプロセッサで動作しているプロセスは，ディスクを持たないプロセスの分もまとめてデータのアクセスを代行する．ディスクを持たないプロセスは，NFS 経由でデータにアクセスするのではなく，ディスクを持つプロセスから読み出したデータを受け取ったり，書き込みデータをそちらに送付したりする．

MPI の集団型入出力機構と異なり，並列ストリーム入出力関数は，特殊な並列ファイルシステムを持たない通常の UNIX WS からなるクラスタにおいても複数のディスクを並列駆動する機能がある．並列ストリーム入出力機構が使用するファイル群は各 WS から見るとそれぞれが通常の UNIX ファイルであり，UNIX ファイルシステムと齟齬を来たすような特殊なデータ構造は使用していない．このため，昼間は個人用の

WS として使用している計算機を夜間にクラスタとして使用する場合に特別な配慮を必要としない他，バックアップ等の操作に UNIX 上で用意されている既存の手段を流用できるという利点がある．

並列パイプ機構は，同じ数のプロセスで動作する MPI プログラム間で用いることを前提として実装されている．ただし，単純に同一 ID のプロセス間でデータを転送するだけでなく，サイクリックに ID のずれたプロセス間を接続する `shift`，プロセスが二次元に配置されていると考えた場合の `rotate`，`transpose` の機能も備えている．これらの機能を実現するため，並列パイプを具現する並列パイプデーモンの形で実装されている．

4 実験結果

並列ストリーム入出力関数群の性能を測定するため，19 台の Sun Microsystems Ultra 5 を 100Mbps Ethernet で接続した WS クラスタによって実験を行った．OS は Solaris 2.7, 使用した MPI ライブラリ (MPI 規格を実装したもの) は MPICH である．

4.1 並列ストリーム入出力と逐次型入出力の比較

まず，並列ストリーム入出力関数によって連続的なデータにアクセスする場合の性能をみるため，MPI プロセス数を変えながら，並列ストリーム入出力関数 (`p_fwrite()`, `p_fread()`) が 1〜4 台のディスクを利用する場合と，通常の逐次型ストリーム入出力関数 (`fwrite()`, `fread()`) が 1 台のディスク (各 WS から NFS マウントされている) を使用する場合の性能を測定した．各 MPI プロセスは，連続した 8,000,000 バイトのデータを 1 バイトずつ読み書きするような関数呼び出しを行っている．

書き込みに要した時間のグラフを図 6 に，読み出しに要した時間のグラフを図 7 に，それぞれ示す．

この実験では各プロセスがアクセスするデータが連続であるため，1 台のディスクを使用する場合だけを見ると，並列ストリーム入出力関数 `p_fwrite()`, `p_fread()` のバッファリングの性能と，NFS 経由で `fwrite()`, `fread()` を利用する場合のバッファリングの性能とを比較していることになる．この場合，並列ストリーム入出力関数の性能は逐次型入出力関数の性能に遠く及ばない．これは次のような理由による．

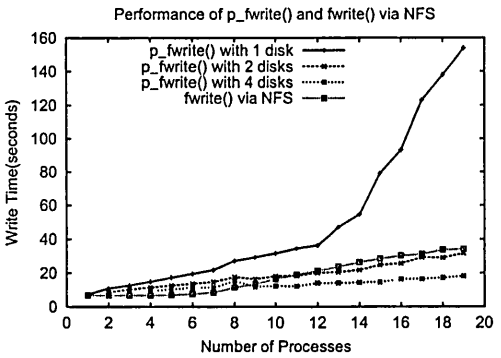

図 6: 並列ストリーム入出力と逐次型入出力の比較 (write)

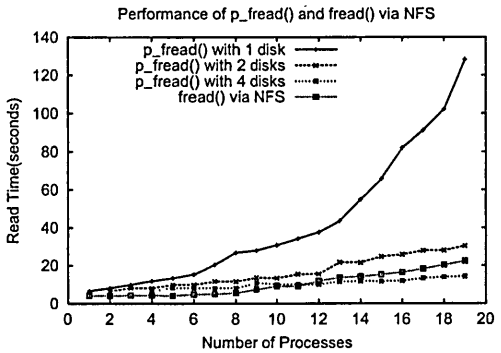

図 7: 並列ストリーム入出力と逐次型入出力の比較 (read)

並列ストリーム入出力関数の現在の実装は，ディスクアクセスの効率を考慮したディスクブロック単位ではなく，1 回あたり集団的に読み書きされるデータ 2〜19 バイト単位のバッファリングを行っている．一方，NFS や `fwrite()`, `fread()` ではブロックサイズを考慮に入れた実装を採用している．このため，複数のディスクを並列に使用することによる転送性能の総和の向上を活用するのでない限り，並列ストリーム入出力関数は従来型の NFS マウントによる単一ファイルにアクセスする場合の性能を越えることができない．

4.2 並列ストリーム入出力と MPI 入出力の比較

次に，並列ストリーム入出力関数によって不連続なデータにアクセスする場合の性能をみるため，MPI プロセス数を変えながら，並列ストリーム入出力関数 (`p_fwrite()`, `p_fread()`) により 1 台のディスクを利用する場合と，MPICH の非集団型入出力関数 (`MPI_File_write_at()`, `MPI_File_read_at()`) により 1 台のディスクを使用する場合の性能を測定した．

各 MPI プロセスは，各プロセスのデータがラウンドロビンで配置された単一ファイルに対して，4 バイトのデータを 100,000 回読み書きするような関数呼び出しを行っている．

書き込みに要した時間のグラフを図 8 に，読み出しに要した時間のグラフを図 9 に，それぞれ示す．

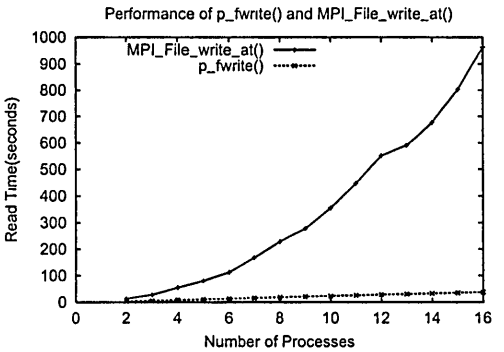

図 8: 並列ストリーム入出力と MPICH 入出力の比較 (write)

これは，ファイルに格納されるデータの配置が，各プロセスが必要とするデータの配置とうまくマッチしない場合の性能を比較している．MPICH の非集団型入出力関数の性能は，このようなミスマッチが起きている場合に，プロセス数が増えるにつれ急激に降下している．

このようなデータ配置のミスマッチがある場合，書き込む前や読み込んだ後にプロセス間通信を行うことによって各プロセスのアクセスするデータが連続になるような形式に再配分すると，再配分のオーバーヘッドを含めても全体のアクセス性能が向上することが知られている [1]．比較の対象とした入出力関数は非集団型であるため，このような性能向上は実現されていない．

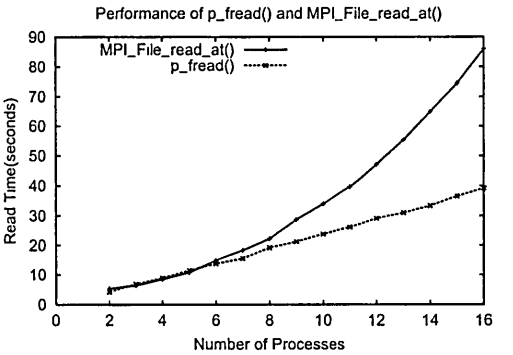

図 9: 並列ストリーム入出力と MPICH 入出力の比較 (read)

4.1 節のように，NFS と逐次型入出力関数経由で単一ファイル上の不連続なデータにアクセスする場合との比較は行っていない．しかし，単一ファイル内の不連続なデータに一括してアクセスする機能は通常の逐次型入出力関数には用意されていないため，不連続なデータ断片のそれぞれに対するアクセスを排他的に実行するようなロック・アンロック操作をユーザプログラム側で陽に用いる必要がある．したがって，この場合の実行性能は，MPICH の入出力関数の場合と同様に，深刻な性能低下を引き起こすであろうと予想している．

4.3 並列パイプによるデータ転送とファイルによるデータ転送の比較

最後に，4.1 節で使用した並列ストリーム入出力関数がもつ並列パイプ機能を利用して複数の MPI プログラム間で縦列状にデータを転送する場合と，同じプログラム群が NFS マウントされたディスク上のファイルを使用してデータを転送する場合との性能を比較する．MPI プログラムは，それぞれ，配列データの読み出し・転置・書き込みを行っている．

この実験の結果を，図 10 に示す．

並列ストリーム入出力関数の並列パイプ機能はデータ転送の仲立ちをする並列パイプデーモンを用いて実装されているが，それによるオーバーヘッドを含めても，ディスクファイル経由でデータ転送を行う場合より経過時間を短縮できることがわかる．

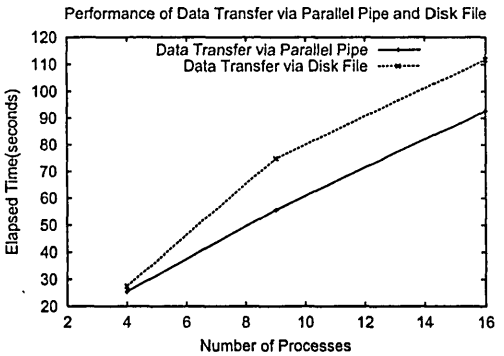

図 10: 並列パイプによるデータ転送とファイルによるデータ転送の比較

5 並列ストリーム入出力ライブラリと並列パイプの応用

並列ストリーム入出力関数と並列パイプには，以下に述べるような応用がある．

5.1 並列プログラムの部品化と再利用

並列ストリーム入出力関数は，図 5 に示したように，並列パイプを経由することによりディスク上のファイルによらずに 1 つの並列プログラムから次の並列プログラムにデータを転送することができる．これは 3 段以上の多段の接続でも同様である．

このため，ある一連の並列処理が複数のフェーズに分割でき，かつ，それぞれのフェーズで動作する各プロセスが同一の並列データストリームを処理するように書き直すことができれば，それらのプログラムを並列パイプで多段に接続するだけで，元のプログラムと同じ処理を実現できる．

このような考え方でさまざまな並列プログラムを細かな部品に分割し集積することができれば，共通の処理については，既存の並列プログラム部品から再構成することができるようになり，並列プログラム部品の再利用を促進することができる．

5.2 並列計算のパイプライン化

並列パイプ機構は，同一の MPI プログラム内でのプロセス間通信とは別の機構により，複数の MPI プログラム間での効率的なデータ転送を実現している．このため，それぞれが独立の MPI プログラムを動作させるように設定されている複数の並列計算機間を接続することも技術的に可能である．この様子を図 11 に示す．

このような処理は，これまで，一次側の並列プログラムの出力をいったんディスクに書き出した後，二次側の並列プログラムに読み込ませるより他に方法がなかった．このため，両者のプログラムの処理を直列に実行せざるを得ず，例えば，一次側のプログラムからの出力が全部終了しなくても二次側の処理を開始できるような場合でも，両者の処理を重ねることによって全体の処理に必要な時間を短縮するようなことはできなかった．

並列パイプ機構を異なる並列計算機間に拡張すると，前述の制約が緩和され，一次側で処理を終えて出力されたデータをただちに二次側のプログラムに供給できるようになる．また，それぞれの並列計算機が複数の外部通信インタフェースを有している場合には，それらを並列に駆動することによって，されに高い転送速度を達成できる可能性がある．

近年，超高速ネットワークを用いて地理的に離れた計算機資源を統合運用するグリッドコンピューティング [3] が注目を集めている．LAN 環境下でのクラスタと異なり，グリッド環境下では地理的な距離による通信遅延が非常に大きくなるため，複数の計算機資源にまたがる MPI プログラムを良好に動作させることは難しい．このため，サイト内と同じ方法で複数の計算機資源を同時に利用することによってより大規模な計算を行うのは非常に難しい．

一方，パイプライン型の並列処理では，パイプライン上の各計算ステップが十分に計算時間を要するものであれば，その間の通信遅延があってもあまり問題とならない．このため，並列パイプ機能を拡張することで，広域分散環境下でのパイプライン型の並列処理を実現することができる．現状では，グリッドコンピューティング環境下で複数の計算資源を統合的に用いることによってより大きな計算力を達成するのに最も適した方法はこのような広域分散パイプライン処理であろう．

6 むすび

本論文では，MPI プログラムのための並列ストリーム入出力関数，および，これを用いて複数の並列プログラム間でディスクを経由せずに並列データ転送を可能にする並列パイプ機構について述べた．

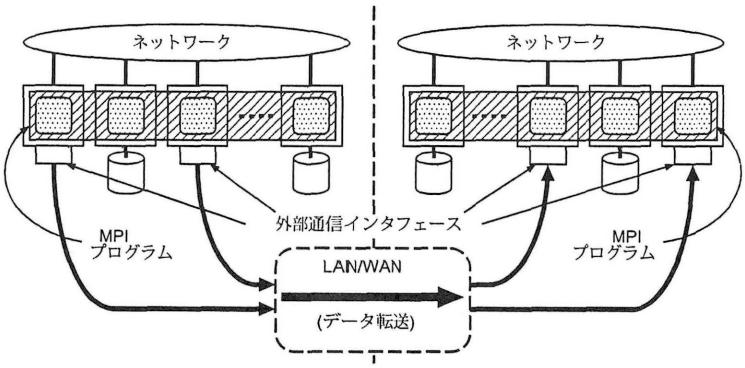

図 11: 並列計算のパイプライン化

4.1 節に述べたように，この並列ストリーム入出力ライブラリには，バッファリング性能の面でまだ改良の余地が残っている．一方，ファイルに格納されるデータ配置と集団型入出力に参加する各プロセスが要求するデータ配置の間にミスマッチがある場合に大幅な性能向上を達成できることがわかっている．

今後はディスクブロックサイズを考慮に入れたバッファリング機構を追加することにより，データ配置のマッチ・ミスマッチに関わらず高い性能を達成できる並列入出力ライブラリの実現を目指したい．また，MPICH の集団型入出力 (`MPI_File_write_at_all()` など) との性能比較も行い，新たな機能を提供しながらこれらの関数と互格の性能を有する入出力ライブラリにしたいと考えている．

並列ストリーム入出力関数とその並列パイプ機能を利用すると，並列プログラムの部品化と再利用を促進することができる．また，並列パイプを用いて並列計算のパイプライン化を行うと，ディスク上のファイルを経由して処理をつなぐ場合よりも処理時間を短縮できる．さらに，並列パイプ機構を広域分散環境に拡張すると，グリッドコンピューティングへの応用が期待できる．

謝辞

本研究にご協力くださった戸川忠嗣氏（現在 (株) 富士通九州システムエンジニアリング）に深謝いたします．

参考文献

[1] Bordawekar, R., del Rosario, J. M. and Choudhary, A.: "Design and Evaluation of Primitives for Parallel I/O", *Proc. Supercomputing '93 Conf.*, pp. 452–461, November 1993.

[2] Corbett, P. F., Feitelson, D. G., Prost, J.-P. and Baylor, S. J: "Parallel Access to Files in the Vesta File System", *Proc. Supercomputing '93 Conf.*, pp. 472–481, November 1993.

[3] *"The GRID: Blueprint for a New Computing Infrastructure"* (Foster, I. and Kesselman, C. eds.), Morgan kaufmann, 1999.

[4] Hatcher, P. J. and Quinn, M. J.: "Dataparallel Programming on MIMD Computers", The MIT Press, 1991.

[5] Message Passing Interface Forum: "MPI: A Message Passing Interface Standard", `http://www.mpi-forum.org/doc/mpi-11-html/mpi-report.html`

[6] Message Passing Interface Forum: "MPI-2: Extensions to Message Passing Interface", `http://www.mpi-forum.org/doc/mpi-20-html/mpi2-report.html`

[7] Patterson, D. A., Gibson, G. and Katz, R. H.: "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proc. 1988 ACM SIGMOD*, pp. 109–116, June 1988.