ハイパフォーマンス
コンピューティング 93−11
（２００３．３．１２）

オブジェクト通信のための Java クラスライブラリの設計と実装

藤村大輔[†] 渡邊誠也[††] 正木亮[††]

本稿ではオブジェクト通信のための Java クラスライブラリ JPI (Java-object Passing Interface) の設計と実装について述べる．JPI はオブジェクト通信を用いた並列 Java プログラムで利用するために (1) Java に適した簡潔で明瞭なインタフェース，(2) マルチスレッドを活かした通信機構，(3) アーキテクチャ非依存性に基づく可搬性，を目標に設計した．複数台の計算機上での並列計算をサポートする自然な API をユーザに提供し，特別な前準備を必要としない並列実行環境を提供する．利用者は JPI を使用することで Java 実行環境がインストールされているマシンならば，何処でも実行が可能な並列プログラムを容易に作成することができる．また，本稿では JPI を用いたプログラム例と性能評価の結果についても述べる．性能評価では JPI を用いたプログラムは最高で MPICH を用いたプログラムの 0.86 倍の実行速度を得ることができた．

Design and Implementation of Java Class Library for Object Passing

Daisuke Fujimura ,[†] Nobuya Watanabe [††] and Akira Masaki[††]

This paper introduces the design and implementation of JPI: *Java-object Passing Interface*, Java class library for the object passing. JPI is designed to be used in parallel Java programs using the object communication, and its design policy includes: (1) providing simple and clear program interfaces, (2) implementation of communication mechanisms using Java multi-threading features, and (3) program portability by architecture-independent of Java. It provides to users natural and acceptable Java APIs for the object communication which support parallel processing on multi-computers, and provides a parallel execution environment which requires no special-purpose servers. By using JPI, user can make and execute parallel programs easily that can run on any platforms where Java runtime is installed. This paper also presents an example of program using JPI and results of the performance evaluation. The results show that the program using JPI can run in maximum of 86% speed as compared to the program written in C using MPICH.

1. はじめに

大規模な科学技術計算には並列処理が不可欠であり，プログラムの記述には MPI (Message Passing Interface)[1] が広く使用されている．MPI はネットワークで接続されたマシン間において，(一般的にはプロセスやスレッドと呼ばれる) それぞれの実行単位が，実行に必要なデータを互いに送受信するために，MPI Forum によって標準化された API (Application Program Interface) である．MPI では実行単位がコミュニケータと呼ばれる集合に属し，その集合の中で一意に割振られたランクを識別子として，通信を行なう．

† 岡山大学大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama University
†† 岡山大学工学部
Faculty of Engineering, Okayama University

一方で，実行環境の OS やアーキテクチャに非依存で可搬性の高いプログラムの開発には Java が利用されており，Java による高性能計算の研究も広く行なわれている．Java は言語レベルでマルチスレッドをサポートしており，並行あるいは並列な動作を自然な形で記述することができる．しかし，高い並列度を有する高性能計算アプリケーションプログラムの実行はマルチプロセッサ環境でなければ，十分な性能は発揮できない．また，Java では分散処理をサポートする RMI (Remote Method Invocation)[2] も提供されているが，専用のコンパイラや，サーバをあらかじめ起動しておく前準備が必要となる．

最近では，MPI と Java のそれぞれの長所を活かすために，Java から利用する MPI の実装として，mpiJava[3] や JavaMPI[4]，MPIJ[5] が研究されており，国内でも関連した研究[6][7] が行なわれている．

mpiJava および JavaMPI は MPICH[8] などの既存

の実装をあらかじめマシンにインストールしておき，それらのネイティブなライブラリに JNI (Java Native Interface)[2] を使用したラッパーからアクセスする．しかし，これらの実装は使用しているライブラリがネイティブコードであるため，性能は向上する反面，使用できる OS やアーキテクチャが限定されてしまうため，Java の最大の特徴である可搬性は失われている．

MPIJ は 100% Java で実装されたメタコンピューティング環境 DOGMA を構成するコア API であるが，DOGMA サーバとの併用を前提としているため，単体での使用は困難である．

また，Java で MPI を利用できるようにする既存の研究全体に言えることだが，提供されている API には C/C++ 言語の色合いが強く残っており，それが Java 言語の柔軟性を損なっている．

本稿では，上述した問題の解決を目指した，オブジェクト通信のための Java クラスライブラリ JPI (Java-object Passing Interface) の設計と実装について述べる．JPI は Java のすべてのオブジェクトを通信の対象とし，利用者が実装したスレッド間でオブジェクトを通信する API と，それを用いた並列実行環境を提供する．

本稿は 2 節で JPI の設計と実装について述べ，3 節で JPI を使用するプログラム例を挙げる．4 節で性能評価の結果とそれについての考察を述べ，5 節で結論と今後の課題を述べる．

2. 設計と実装

2.1 設計方針

JPI は以下の設計方針に基づいて，設計を行なった．

- Java 言語に適した，簡潔で明瞭なインタフェース
- Java のマルチスレッドを活かしたメッセージ通信機構を提供する並列実行環境の実現
- 並列実行環境の準備の容易さ
- Java の可搬性の維持

2.2 インタフェース

メッセージの送信に必要な情報は (1) 送信するメッセージ自身，(2) メッセージの宛て先，(3) メッセージを識別するためのタグの 3 つである．MPI のメッセージ送信ルーチン `MPI_Send` では，上述の (1) 〜 (3) の他に (4) メッセージが格納されているメモリの長さや (5) データの型を引数として与える必要がある．また，C 言語や Fortran 言語には例外処理機構がないため，返値でルーチンの終了状態を通知するようになっている．

JPI では上述の Java 言語にそぐわない性質を廃したインタフェースを提供する．Java 言語ではオブジェクトは必要な情報を内部に保持しており，メモリに関する概念はクラスに隠蔽されている．よって，Java 言語のためのルーチンは上述の (4) および (5)，すなわちメモリに関する明示的な情報を必要としない．また，メッセージ送信ルーチンをコミュニケータを表現するクラスのメソッドとして提供することで，コミュニケータを引数として渡す必要もない．この手法は MPIJ[5] を参考にした．返値に関しては，Java には例外処理機構があるため，メソッドが例外をスローする可能性があることを宣言すれば良い．よって，返値も必要ない．

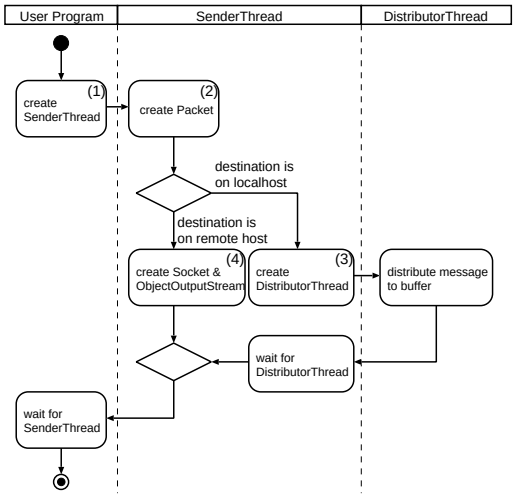

図 1 メッセージ送信の流れ

メッセージ受信ルーチンにも同様のことが当てはまる．加えて，Java 言語にはプリミティブに対するポインタがないため，プリミティブに対して，メッセージ受信ルーチンの引数にアドレスを渡して，そのアドレスにメッセージを格納するという，オブジェクトと同様のインタフェースを提供することはできない．そのため，受信したメッセージをルーチンの返値から受け取るようにすれば，格納先の準備の必要のない簡潔で一意な記述となる．

2.3 メッセージ通信のメカニズム

JPI を使用したプログラムは 2 種類のスレッド，**ユーザスレッド** と **JPI スレッド** によって，実行される．前者は利用者によって実装された並列処理プログラムを実行するスレッドであり，後者は JPI のクラスライブラリ内の処理を行なうスレッドである．メッセージの送受信は後者のスレッドが行なう．

初めにメッセージ送信の流れを図 1 に示す．

(1) ユーザスレッドからメッセージ送信メソッドが呼ばれると，メッセージ送信用の JPI スレッド

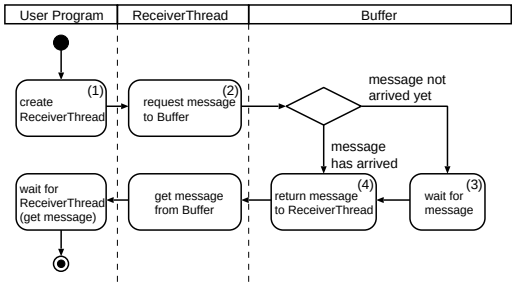

図 2 メッセージ受信の流れ

SenderThread が生成される．

(2) SenderThread はメッセージやタグなどをひとまとめにしたパケットを作成する．

(3) 送信先がローカルホスト上のユーザスレッドの場合は，バッファ格納用 JPI スレッド DistributorThread にパケットを渡す．

(4) 送信先がリモートホスト上のユーザスレッドの場合は，そのリモートホストの受信サーバ用 JPI スレッドに接続し，ソケットを通して，パケットを送信する．

続いて，メッセージ受信の流れを図 2 に示す．

(1) ユーザスレッドからメッセージ受信メソッドが呼ばれると，メッセージ受信用の JPI スレッド ReceiverThread が生成される．

(2) ReceiverThread はメッセージが既に到着しているかどうかを Buffer に問い合わせる．

(3) まだメッセージが到着していなければ，メッセージの到着を待つ．

(4) メッセージが到着したら， ReceiverThread は Buffer からメッセージを取り出す．そして，最終的にメソッドの返値として，メッセージが返される．

メッセージの送受信に必要なそれぞれの処理はユーザスレッドとは別の JPI スレッドで処理する．これにより，ユーザプログラムをより自然に記述することが可能となり，さらに Java 仮想マシンのマルチスレッドスケジューリングによる，効率の良い実行が期待できる．

2.4 クラス

JPI を使用したプログラムでは， JApplication クラスを継承したクラスのオブジェクトを JRun クラスがユーザスレッド化することで並列実行される．各ユーザスレッドは JComm および JHandle クラスの提供するメソッドやフィールドを使用して，オブジェクトの送受信を行ないながら，プログラムを実行する． JPI クラスライブラリが提供するクラスを以下に述べる．

```
% java jpi.JRun options classname arguments
```

図 3 JRun の コマンドラインからの入力フォーマット

```
public class JApplication {
 JHandle JPI;
 abstract public void Jmain(String[]);
}
```

図 4 JApplication クラス

2.4.1 JRun クラス

JPI を使用したプログラムをブートするためのクラスである．図 3 にコマンドラインからの入力フォーマットを示す． *options* には並列処理に関するオプションを与える． *classname* には並列処理させたいプログラムを定義したクラスの名前を与える． *arguments* にはユーザスレッドに渡す引数列を与える．この引数列は後述の `Jmain()` メソッドの引数として，渡される．

2.4.2 JApplication クラス

JApplication のクラス宣言を図 4 に示す．利用者はこのクラスを継承したサブクラスの `Jmain()` メソッドに実行させたい内容を記述して，オーバーライドすることで，並列処理を行なうことができる．また，このメソッドは `String[]` クラスの引数として， 2.4.1 で述べた *arguments* を受け取る．

2.4.3 JComm クラス

各ユーザスレッド間でオブジェクトの通信を行なうために必要なメソッド群を提供する．提供するメソッドの代表的なものを以下に示す．

`void send(Object msg, int dst, int tag)`
: オブジェクト `msg` に識別用のタグ `tag` を付けて，ランク `dst` のユーザスレッドへ送信する．

`Object recv(int src, int tag)`
: ランク `src` のユーザスレッドから受信した，識別用のタグ `tag` を持つオブジェクトを返す．

`int rank()`
: そのコミュニケータにおけるユーザスレッドのランクを返す．ランクはスレッド毎の処理を分けるための条件分岐などで使用される．

`int size()`
: そのコミュニケータ内のユーザスレッドの数を返す．データを各スレッドに均等に分割したい時などに利用される．

以上のように，メソッドの API は必要最小限の引数のみで構成している．また， JPI の起動時から利用できるコミュニケータとしては，すべてのユーザスレッドが属している `COMM_WORLD` が提供されている．

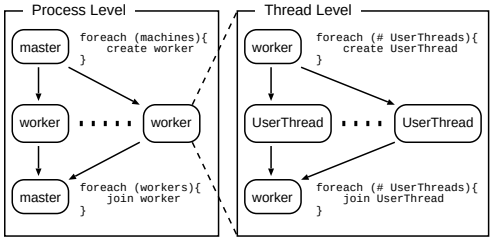

図 5 プログラムの並列実行

2.4.4 JHandle クラス

各ユーザスレッドが属しているコミュニケータや通信を制御する特別なタグなどの静的な情報を提供する．ユーザスレッドからはこのクラスのインスタンス JPI にアクセスして，情報を取り出す．

2.4.5 その他のクラス

上述のクラスの他に，JPI クラスライブラリが提供するメソッドがスローする例外を示す JException や，非同期通信に関する情報を保持する JRequest，リダクション操作を定義する JOperation クラスが用意されている．

2.5 プログラムの並列実行

JPI を使用したプログラムの並列実行を図 5 に示す．

並列実行環境は master と worker の 2 種類の Java 仮想マシンプロセスから構成される．master はコマンドラインから起動されるプロセスで，worker は master から起動されるプロセスである．

並列実行環境として利用するマシンはすべて NFS (Network File System) によって，カレントディレクトリをマウントしておく必要がある．これは worker が初期化に必要な情報を，master が生成した一時ファイルから読み出すためである．NFS は一般的な LAN では既に設定されている可能性が高いため，ほとんどの場合，前準備として設定する必要はないと考えられる．

また，Java ではリモートのマシンで OS のシステムコールを経由せずに直接 Java 仮想マシンを起動する API は提供していない．本研究では Java から利用できるセキュアシェルクライアントとして，JSSH (A Java Secure Shell client library)[9] を使用することで，リモートのマシンに Java 仮想マシンを起動させる実装としている．セキュアシェルもまた，NFS と同様の理由から，前準備として設定する必要はないと考えられる．

続いて，JPI プログラムの実行の手順について述べる．JPI プログラムの実行を開始するには，コマンドラインから図 3 のフォーマットに従って，master を起動する．

master はコマンドラインからオプションとして受け取った情報を一時ファイルに書き出した後，上述のセキュアシェルを使用して，使用するマシンにそれぞれ worker を起動させる．master と worker は同一マシン上で起動させることもできる．worker は master が書き出した一時ファイルを読み込み，初期化に必要な情報を抽出する．

また，各 worker は他の worker が起動する前に通信を開始してしまうことを防ぐために，すべての worker との間で同期をとる．同期がとれると，各 worker は読み出した情報に基づいて，ユーザスレッドを生成する．各ユーザスレッドには，一意に識別できるようにするためのランクが割り振られる．ランクは int 型の 32 bit 整数値である．

worker は自身が生成したユーザスレッドがすべて終了したことを確認してから，終了する．そして，master はすべての worker が終了したことを確認してから，終了する．並列実行環境のための後始末は，前準備と同様に必要ない．

3. JPI の使用例

3.1 プログラムの例

JPI を使用したプログラムの例として，ping-pong 性能測定プログラムを図 6 に示す．

このプログラムは 2 つのユーザスレッドの間でバイト列を往復させ，バイト列の大きさと往復に要した時間を表示する．プログラムでは初めに 5 行目で自身のユーザスレッドのランク，6 行目で起動したユーザスレッドの数を入手し，ユーザスレッドの数が 2 でなければ，直ちに終了する．14 行目から 22 行目まではランク 0，24 行目から 26 行目まではランク 1 のユーザスレッドがそれぞれ行なう．

また，通信時間の測定中に GC が発生しないようにするため，性能測定に無関係な部分でデータへの参照を破棄 (28 行目) し，強制的に GC (29 行目) を実行している．

3.2 プログラムの実行

このプログラムを実行させるには，図 7 のように指定する．`-mf` は並列実行させるために worker を起動させるマシンの一覧を指定するオプションであり，その次に与えられた `machines.smp` には worker を起動させるマシンの名前が列挙されている．`-nt` は起動させるユーザスレッドの総数を指定するオプションである．`-J` は worker を起動する際の Java 仮想マシンに与えるオプションであり，ここでは `-Xmx`[2] でメモリ割り当てプールの最大サイズを 1024M バイトに指定している．

```
1  import jpi.*;
2  public class PingPong extends JApplication {
3   public void Jmain(String[] args) {
4    try {
5     int rank = JPI.COMM_WORLD.rank();
6     int size = JPI.COMM_WORLD.size();
7     if (size != 2) System.exit(1);
8     System.err.println("Thread " + rank +
9       " on " + JPI.getProcessorName());
10    long start = 0, end = 0;
11    int len = 0, max = 1 << 28;
12    byte[] data = null;
13    for (len = 1; len <= max; len <<= 1) {
14     if (rank == 0) {
15      data = new byte[len];
16      start = System.currentTimeMillis();
17      JPI.COMM_WORLD.send(data, 1, 100);
18      data =
19        (byte[])JPI.COMM_WORLD.recv(1, 200);
20      end = System.currentTimeMillis();
21      System.out.println("length=" + len*2 +
22        " time=" + (end - start));
23     } else if (rank == 1) {
24      data =
25        (byte[])JPI.COMM_WORLD.recv(0, 100);
26      JPI.COMM_WORLD.send(data, 0, 200);
27     }
28     data = null;
29     System.gc();
30    }
31   } catch (JException e) {
32    e.printStackTrace();
33    System.exit(1);
34   }
35  }
36 }
```

図 6 ping-pong 性能測定プログラム

```
% java jpi.JRun -mf machines.smp -J-Xmx1024m \
  -nt 2 PingPong
```

図 7 ping-pong 性能測定プログラムの JRun による起動

4. 性能評価

4.1 性能評価プログラム

性能評価には 3 節で述べた ping-pong 性能測定プログラムと Jacobi 法を用いて熱平衡状態を求めるプログラム の 2 種類のプログラムを使用した．性能評価に利用したマシンのスペックを表 1 に，マシン間のネットワーク構成を図 8 に示す．

ping-pong 性能測定プログラムでは node0 と node1 でユーザスレッドを 1 つずつ生成して，スループットを測定した．また，熱平衡状態を求めるプログラムで解く問題のサイズは 512 × 512 の double 型 2 次元配列とした．

表 1 性能評価環境のスペック

ホストマシン host	
CPU	Pentium 4 (2.0AGHz) (FSB 400MHz)
MEM	PC800 RAMBUS 1GB (256MB RIMM×4)
NIC	Intel Pro 100
OS	Red Hat Linux 8.0 (kernel-2.4.18)

ノードマシン node0 ～ node3	
CPU	Pentium III (1.0GHz)×2 (FSB 133MHz)
MEM	PC133 SDRAM 1GB (256MB DIMM×4)
NIC	Intel Pro 100, Intel PRO/1000
OS	Red Hat Linux 7.3 (kernel-2.2.21smp)

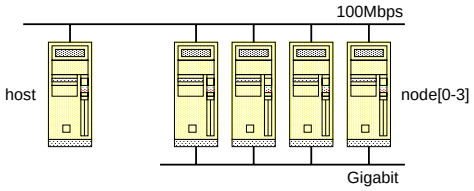

図 8 性能評価環境のネットワーク構成

JPI を利用した解法プログラムの実行には J2SE 1.4.1_01 を使用した．また，MPICH-1.2.2.3 を比較対象とし，それを利用した解法プログラムのコンパイルには GCC 3.2 20020903 を使用した．

測定は使用するノード数 n と使用するノードでそれぞれ起動するユーザスレッド数 m の組合せ $(n \times m)$ を (1×1)，(1×2)，(2×1)，(2×2)，(4×1)，(4×2) の 6 パターンで行なった．JPI は各 worker で生成したユーザスレッドで計算を行なったが，MPICH はマルチスレッドセーフではないので，ユーザスレッドの代わりにプロセスを同じ数だけ起動させた場合と比較した．各ノードは表 1 に示すように 2 個の CPU を搭載しているため，全体で同じユーザスレッド数の場合でも，メモリが共有されている場合と分散している場合の差異も測定することができた．

4.2 実験結果と考察

ping-pong 性能測定プログラムの結果を図 9 に，Jacobi 法を用いて熱平衡状態を求めるプログラムの結果を図 10 に示す．

前者からは，通信データサイズが 1M バイトまでは JPI のスループットは MPICH の 1/100 程度しか得られていないが，通信データサイズがそれ以上になると，差異はほとんどなくなるという結果が得られた．一方，後者からは，JPI を使用したプログラムと同じアルゴリズムで実装した MPICH プログラムと比較すると，0.16 ～ 0.86 倍の実行速度であるという結果が得られた．両者の結果からは以下のような傾向が見られた．

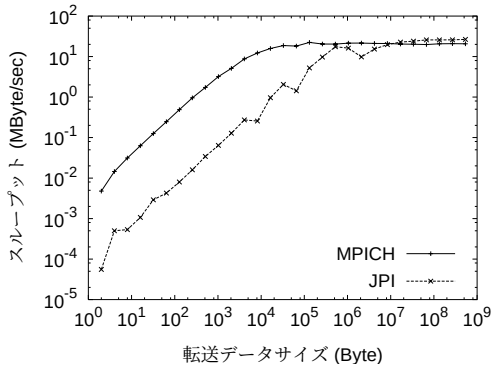

図 9 ping-pong 性能測定プログラムの実行結果

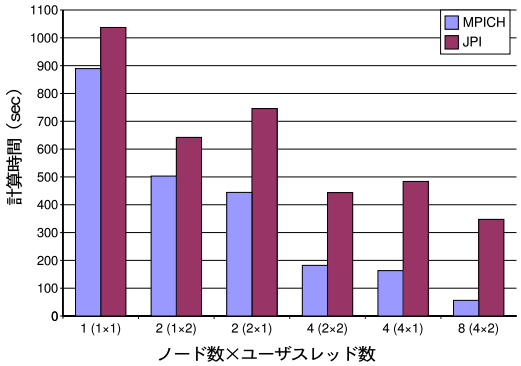

図 10 熱分散問題の解法プログラムの実行結果

- ユーザスレッド数が増加するにつれて，MPICH との実行速度に差が開いている．これは解法プログラムが 1 回に行なう通信量が 512 × 8 バイトであり，図 9 より，この通信データサイズでの JPI のスループットは MPICH よりも大幅に劣っているためであると考えられる．
- 全体のユーザスレッド数が同じ時はノード数が少ない方が性能が良い．これはローカルのユーザスレッドにメッセージを送信する場合には，オブジェクトの参照を渡すだけで済むのに対して，リモートのユーザスレッドにメッセージを送信する場合には，受信したメッセージのオブジェクトを新たに生成するコストと，オブジェクトの生成を繰り返すことで発生する GC の影響と考えられる．

5. おわりに

本稿では，オブジェクト通信のための Java クラスライブラリ JPI の設計と実装について述べた． JPI は Java と MPI のそれぞれの長所を活かした上で， Java に適した簡潔で明瞭なインタフェースを利用者に提供することを目標として，設計を行なったため，従来の研究に比べ， Java プログラマにとって，より自然なインタフェースを提供している．

一方で，プログラムの可搬性とプラットフォーム非依存性を重視したため，実行性能が問題であったが， Java のマルチスレッド機構を活用した実装により，マルチプロセッサマシン上で効率の良い通信処理を行なえるように工夫をした．それにより，性能評価実験では MPICH を利用して書かれた C 言語のプログラムと比較して，最高で 0.86 倍の実行速度を得ることができた．

プログラムや実行環境によっては，性能的に問題になることもあるが，これらの解決は今後の課題である．その他の課題としては， (1) より大規模なアプリケーションによる性能評価実験， (2) プラットフォーム非依存性を向上するために NFS を必要としない並列実行環境の提供， (3) 耐故障性を向上するための手法の検討， (4) 様々な実行環境に適応可能なチューニング手法の検討，を考えている．

謝辞 本研究は，文部科学省研究費補助金 (若手研究 (B) 課題番号 13780240) による．

参 考 文 献

1) Message Passing Interface (MPI) Forum
http://www.mpi-forum.org/
2) Java™ 2 Platform, Standard Edition version 1.4
http://java.sun.com/j2se/1.4/ja/
3) Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim : “mpiJava: An Object-Oriented Java interface to MPI”, International Workshop on Java for Parallel and Distributed Computing (1999).
4) S. Mintchev : “Writing Programs in JavaMPI”, Technical Report MAN-CSPE-02 School of Computer Science, University of Westminster, London, UK (1997).
5) MPIJ
http://dogma.byu.edu/OnlineDocs/docs/mpij/MPIJ.html
6) 日下部 明, 廣安 知之, 三木 光範 : “Java による MPI の実装と評価,” 2000 年記念並列処理シンポジウム JSPP2000 論文集, pp. 269-276 (2000).
7) 坂口 聡, 小畑 正貴 : “Java による PC クラスタソフトウェアの開発と評価,”情報処理学会研究報告 HPC-142, pp.151-156 (2001).
8) MPICH
http://www-unix.mcs.anl.gov/mpi/mpich/
9) JSSH
http://www.pitman.co.za/projects/jssh/