卒業研究報告書

題 目

BSP モデル上での最短経路問題を解くアルゴリズム

指 導 教 員

石 水　隆 助手

報 告 者

02–1–47–103

水 口　貴 裕

近畿大学理工学部情報学科

平成 18 年 2 月 10 日提出

概要

本研究では BSP（Bulk Synchoronous Parallel）モデル上で全頂点最短路 (All-pairs Shortest Paths) 問題を解く並列アルゴリズムを提案する。

BSP（Bulk Synchoronous Parallel）モデルとは、Valiant により提案された分散メモリ型非同期式並列モデルのであり、PRAM を代表とする従来の並列計算モデルでは無かった、通信遅延および同期周期時間を考慮した新しい並列計算モデルのである。

BSP 上でシミュレートし、その実行時間を計測するプログラムを作成する。

本研究では、重み付連結無向グラフ G が与えられたとき、$p \times q$ プロセッサを用いて BSP モデル上で G の全頂点最短路 $O\left(\frac{n^s \log n}{pq} + \frac{g(p+q)n^2 \log n}{pq} + L \log n\right.$ 時間で求めるアルゴリズムを提案する。ただし、g は 1 メッセージ辺りの送信および受信時間、もしくは同期時間である。

また、そのアルゴリズムの BSP モデル上の動作をシミュレートするプログラムを作成し、アルゴリズムの BSP モデル上での時間計算量を実験的に評価する。

目 次

1 序論

1.1 並列処理

膨大な計算が必要とされている問題を解くために、コンピュータのアーキテクチャ、プロセッサは日々改良されているしかしながら 1 台のプロセッサによる計算量には限界があり、その限界量を超える計算量が必要とされている問題を解くために並列処理 (Parallel Processing) という概念が生み出された。

並列処理とは、複数の処理装置またはプロセッサからなり、ある問題を解く際にその問題の異なる部分問題を同時に解き全体の結果を導くのに協力し合うことであり、使用されるプロセッサ数は数個から多い場合数万にまでわたる。その結果、これまでの単一によるプロセッサで問題を解くのにかかる時間よりも非常に少ない時間で解くことができる。

複数のプロセッサを持つ並列計算機 (Parallel Computer) を用いて並列処理を行うことにより、単一プロセッサの逐次計算機よりも高速に問題を解くことができる。

並列計算機上で並列処理を行うアルゴリズムが並列アルゴリズム (Parallel Algorithm) である

1.2 並列処理の目的

本研究の要となる並列処理の目的は、次に挙げられる。

まず、複数のプロセッサによって処理手順を並列に実行することにより、解を得るための時間を短縮する。

次に再帰表現を多用するプログラムなど難解で本質的に並列性を持っている問題も、その問題が持っている表現の複雑さ困難さを解消するはたらきがある。

また、多重化と呼ばれシステムの一方の誤作動を検出し、もう一方がそれを訂正するという信頼性を向上させるシステムでもある。

本研究では、この中での解を得られるための時間を短縮するという目的に立って並列処理アルゴリズムを作成する。

1.3 並列計算モデル (Parallel Computing Model)

並列アルゴリズムの設計・解析は、並列計算機を抽象化した並列計算モデル (Parallel Computing Model) 上で行われる。代表的な並列計算モデルとして、PRAM(Parallel Random Access Machine)[4], Mesh[4], Hyper-cube[4], BSP (Bulk-Syncronous Parallel) モデル [6], などがある。

並列アルゴリズムの設計・解析は多くの場合 PRAM が使用される。しかししかし PRAM は通信・同期に掛かるコストを無視しているため、PRAM のアルゴリズムが必ずしも現実の計算機に効率良く適用できるとは限らない。そのため、より現実に近い並列計算モデルとして BSP モデルが注目されている。BSP モデルは、ネットワーク接続された複数のプロセッサから成る分散メモリ型並列計算モデルであり、メッセージの送受信やプロセッサ間での同期に掛かる時間を考慮することができる。本研究では BSP モデルを対象とし、BSP モデル上で効率の良い並列アルゴリズムの設計を行う。

1.4 全頂点最短路問題 (All-Pairs Shortest Psths)

各辺が非負の重みを持つ連結無向グラフ G に対して、ある頂点 u からある頂点 v までの経路 (Path) のうち、経路上の辺の重みの和が最小となる経路を u, v 間の最短路 (Shortest Path) と言う。最短路問題とは、G のある 2 頂点 u, v に対して、その最短路を求める問題であり、全頂点最短路問題とは全ての頂点間で最短路を求める問題である。

頂点数 n, 辺数 m のグラフに対し Dijkistra は $O((m+n)\log n)$ 時間で最短路問題を解くアルゴリズム [3] を提案した。また、Froyd は $O(n^3)$ 時間で全頂点最短路問題を解くアルゴリズム [3] を提案した。本研究では BSP モデル上で p プロセッサを用いて $O(\frac{n^3 \log n}{pq} + g\frac{(p+q)n^2 \log n}{pq} + L\log n)$ 時間で全頂点最短路問題を解く並列アルゴリズムを提案する。

2 準備

2.1 BSP モデル

初期の並列計算機ではプロセッサの処理能力は低く、プロセッサの内部演算時間に比べてプロセッサ間の通信はさほど考慮されていなかった。しかし、ここ数十年でプロセッサの処理能力は急激に向上したため、通信や同期にかかるコストとが処理時間において大きなウエイトを占めるようになってきた。そうした背景によって、Valiant により BSP(Bulk-Syncronous Parallel) モデル [6] が提案された。

BSP モデルは非同期式分散メモリ型の並列計算モデルである。BSP は局所メモリを持つ複数がプロセッサとそれらを結びつけるネットワークおよびプロセッサ間でバリア同期を取るための同期機構からなる。プロセッサ間の通信はネットワークを通して 1 対 1 でメッセージ交換をすることにより行われる。なお、バリア同期とは、協調して動作する多数のプロセッサの歩調を合わせることを目的とした同期プリミティブである。バリア同期を実行して同期を取る場合、全てのプロセッサがバリアに到達するまでどのプロセッサも

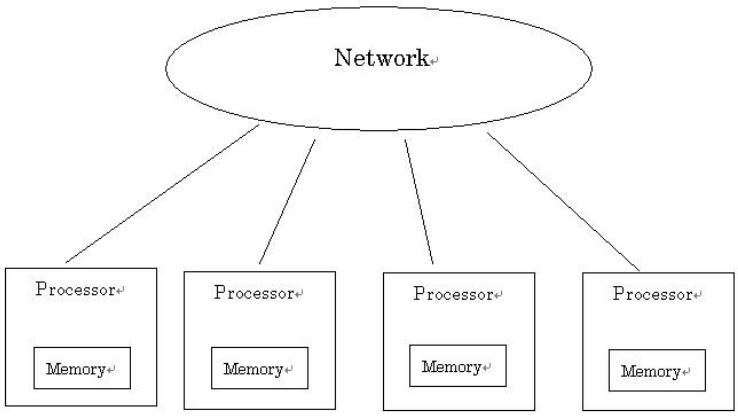

図 1: BSP (Bulk-Syncronous Parallel) モデル

実行を継続できず、封鎖される。BSP モデルはこのような理由から生まれたモデルである。図 1 に BSP モデルの概念図を示す。

BSP モデルは通信遅延や同期時間等を表す為に以下のパラメタを持つ。

- p, q：プロセッサ数。本研究では、$p \times q$ 台のプロセッサが 2 次元上に配置されていると仮定し、各プロセッサを $P_{i,j}$ $(0 \leq i < p, 0 \leq j < q)$ と表わす
- g：1 つのメッセージを送信あるいは受信するのにかかる時間。
- L：バリア同期を取るのにかかる時間　(通信遅延時間)。

BSP モデル上での並列アルゴリズムは各プロセッサが実行するプログラムにより表される。各プロセッサが実行するプログラムはスーパーステップの列からなる。各スーパーステップは内部計算命令の列から成る内部計算フェーズと、送信あるいは受信命令の列から成る通信フェーズで構成されており、各プロセッサは割り当てられたスーパーステップを非同期に実行する。スーパーステップの命令終了後、プロセッサ間でバリア同期を取り、次のスーパーステップの実行に移る。あるスーパーステップで各プロセッサが各々w 個の内部計算命令と各々h 個の通信命令を実行する場合、そのスーパーステップの時間計算量は $O(w + gh + L)$ となる。

2.2 全頂点最短路問題 (All-Pairs Shortest Paths)

各辺が非負の重みを持つ連結無向グラフ G に対して、ある頂点 u からある頂点 v までの経路 (Path) のうち、経路上の辺の重みの和が最小となる経路を u,v 間の最短路 (Shortest Path) と言う。全頂点最短路問題とは、重み付無向グラフ G が与えられたとき、全ての頂点間で最短路を求める問題である。頂点数 n, 辺数 m のグラフに対し Dijkistra は $O((m+n)\log n)$ 時間で最短路問題を解くアルゴリズム [3] を提案した。Froyd は $O(n^3)$ 時間で全頂点最短路問題を解くアルゴリズム [1] を提案した。

2.3 重み付グラフの隣接行列 (Adjacency Matrix)

頂点数 n の重み付グラフ G が与えられたとき、(u,v) $(0 \leq u,v < n)$ 成分 $a_{u,v}$ が辺 (u,v) の重みの値である行列 $A = \{a_{x,y}\}$ $(0 \leq x,y < n)$ をグラフ G の隣接行列 (Adjacency Matrix) と言う。ただし、辺 (u,v) が存在しないとき成分 $a_{u,v}$ は無限大の値を持ち A の対角成分 $a_{x,x}$ $(0 \leq x < n)$ は 0 の値を持つ。

3 全頂点最短路問題を解くアルゴリズム

3.1 アルゴリズム

頂点 u から頂点 v の経路が k 本の辺で構成されているとき、長さ k の経路と言う。重み付連結無向グラフ G の隣接行列 A は、長さ 1 の経路、すなわち u から v へ直接繋がる辺のみによる G の全頂点最短路を表している。

全頂点最短路は以下の手順で求めることができる。

$n \times n$ 行列 $A = \{a_{x,y}\}, B = \{b_{x,y}\} (0 \leq x,y < n)$ に対して、行列演算 $C = A \circ B$ を以下のように定義する

$$c_{x,y} = \min_{0 \leq k < n} \{a_{k,x} + b_{y,k}\}$$

グラフ G の隣接行列を A とすると、A^k は、長さ k 以下の経路のみによる G の全頂点最短路を表す。ただし、$A^2 = A \circ A, A^{i+1} = A^i \circ A$ である。頂点数 n の G の経路の長さは最大で $n-1$ である。よって、G の全頂点最短路は、A^{n-1} で求めることができる。

行列演算 $\circ$ は行列積と同形の演算である。従って行列積を求めるアルゴリズムを繰り返し用いることにより、全頂点最短路を求めることができる。また、行列演算 $\circ$ は結合則を満たす演算であるので、$A^{2i} = A^i \circ A^i$ が成立する。従って $logn$ 回の繰り返しにより A^{n-1} を求めることができる。

以下に本研究で提案した BSP モデル上で全頂点最短路問題を解く並列アルゴリズム **BSP-APSP** を示す。

(アルゴリズム **BSP-APSP**)

入力: 頂点数 n の重み付無向グラフ $G = (V, E)$ の隣接行列 A。プロセッサ $P_{i,j}$ $(0 \leq i < p, 0 \leq j < q)$ が A のサイズ $\frac{n}{p} \times \frac{n}{q}$ の部分行列 $A_{i,j}$ を持つ。

出力: G の全頂点最短路。プロセッサ $P_{i,j}$ $(0 \leq i < p, 0 \leq j < q)$ が A^n のサイズ $\frac{n}{p} \times \frac{n}{q}$ の部分行列 $A^n_{i,j}$ を持つ。

以下の操作を $\log n$ 回繰り返す。

手続き BSP-MM を用いて行列積 $A \circ A$ を計算し、その結果を A 自身に代入する。ただし行列積演算 $C = A \circ B$ を $c_{x,y} = \min_{0 \leq k < n}\{a_{k,x} + b_{y,k}\}$ と定義する。

(手続き **BSP-MM**) [入力:] サイズ $n \times n$ の行列 A, B および行列積演算子 $\circ$ プロセッサ $P_{i,j}$ $(0 \leq i < p, 0 \leq j < q)$ が A, B のサイズ $\frac{n}{p} \times \frac{n}{q}$ の部分行列 $A_{i,j}, B_{i,j}$ を持つ。[出力:] 行列積 $C = A \circ B$。プロセッサ $P_{i,j}$ $(0 \leq i < p, 0 \leq j < q)$ が C のサイズ $\frac{n}{p} \times \frac{n}{q}$ の部分行列 $C_{i,j}$ を持つ

1. プロセッサ $P_{i,j}$ は保持する部分配列 $A_{i,j}$ を横方向のプロセッサ $P_{i,0}, P_{i,1}, ..., P_{i,q-1}$ に送信する。
2. プロセッサ $P_{i,j}$ は保持する部分配列 $B_{i,j}$ を縦方向のプロセッサ $P_{0,j}, P_{1,j}, ..., P_{p-1,j}$ に送信する。
3. (1),(2) で受信したデータを用いてプロセッサ $P_{i,j}$ は $C = A \circ B$ のサイズ $\frac{n}{p} \times \frac{n}{q}$ の部分行列 $C_{i,j}$ を計算する。

3.2 計算量

この章では本研究で提案した BSP モデル上で全頂点最短路問題を解く並列アルゴリズムの時間計算量を検証する。

BSP モデル上の行列積については以下の補題が成り立つ。

補題 1 サイズ $n \times n$ の行列 A, B および行列積演算子 $\circ$ が与えられたとき、$p \times q$ $(p, q \leq n)$ プロセッサを用いて BSP モデル上で $O(\frac{n^3}{pq} + g\frac{(p+q)n^2}{pq} + L)$ 時間で行列積 $C = A \circ B$ を計算できる。

(証明)

手続き BSP-MM の 1. において各プロセッサはサイズ $\frac{n}{p} \times \frac{n}{q}$ のデータを q プロセッサに送信し、また、2. において各プロセッサはサイズ $\frac{n}{p} \times \frac{n}{q}$ のデータを p プロセッサに送信する。従って 1. および 2. の時間計算量は $O(g\frac{(p+q)n^2}{pq} + L)$ である。3. において行列 C の各成分 $c_{x,y}$ は $O(n)$ 時間で計算できる。従ってサイズ $\frac{n}{p} \times \frac{n}{q}$ の部分行列は $O(\frac{n^3}{pq})$ 時間で計算できる。 □

補題 1 より、以下の定理が成り立つ。

定理 1 頂点数 n の重み付連結無向グラフ G が与えられたとき BSP モデル上で $p \times q$ ($p, q \leq n$) プロセッサを用いて $O(\frac{n^3 \log n}{pq} + g\frac{n^2 \log n}{pq} + L \log n)$ 時間で全頂点最短路問題を解ける。

(証明)

全頂点最短路問題は行列積を $\log n$ 回繰り返すことにより求めることができる。よって補題 1 より題意が成り立つ。 □

3.3 シミュレーションアルゴリズム

本研究で提案した BSP モデル上で全頂点最短路問題を解く並列アルゴリズムの時間計算量を実験的に評価するため、シミュレートプログラムを用いてその実行時間を検証する。

付録に本研究で作成したアルゴリズム BSP-APPS の BSP モデル上での実行をシミュレートするシミュレートプログラムを示す。

4 結果

本研究で作成したシミュレートプログラムを用いて、頂点数 n のグラフに対してプロセッサ数を変化させたときの実行時間の変化を示す。

頂点数 n を 64、同期周期を 16、通信遅延を 16 とした場合において、縦方向のプロセッサ数を 1 にして横方向のプロセッサ数を変えた場合の実行時間を測定した結果を図 2 におよび、64 個のプロセッサ 1×64、2×32、4×16、8×8 で、縦横への割り振り方を変えて実行時間を測定した結果を図 3 に示す。

さらに頂点数 n を 64　縦横方向プロセッサをそれぞれ 8 通信コストを 16 に固定した時の通信遅延を 16 に固定して、通信コスト値を変えた値を図 4 に示す。

5 考察

図 2 はプロセッサ数と実行時間の関係である。ただしプロセッサの数は縦行列分に分割せず、横にのみ分割したものである。プロセッサの数が増えるにつれ、実行時間が減少した。また、縦と横を反対にした場合でも同じ結果が導かれたことを確認した。これは、行列計算を逐次処理することによって、作業時間が短縮されたためだと考えられる。

図 3 同じプロセッサ数で、分割形態によっての実行時間の変遷を示したグラフである。プロセッサの縦横の台数が近づくにつれ実行時間が減少した。

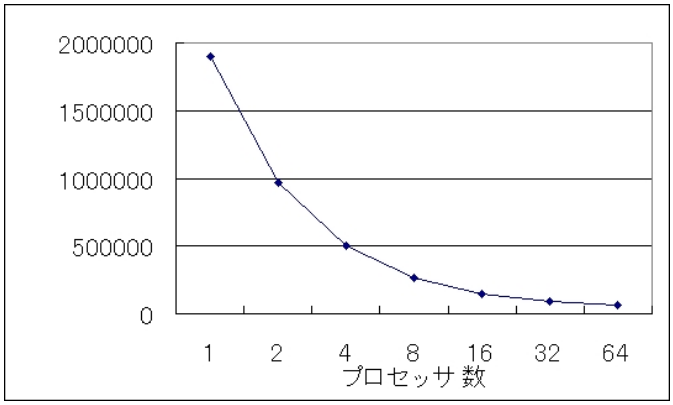

図 2: プロセッサと実行時間

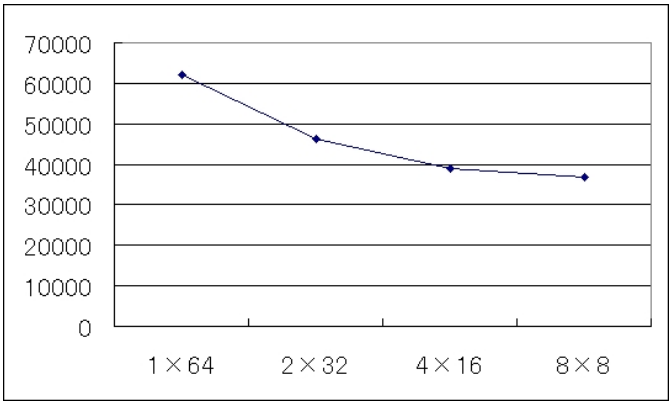

図 3: プロセッサの分割形態

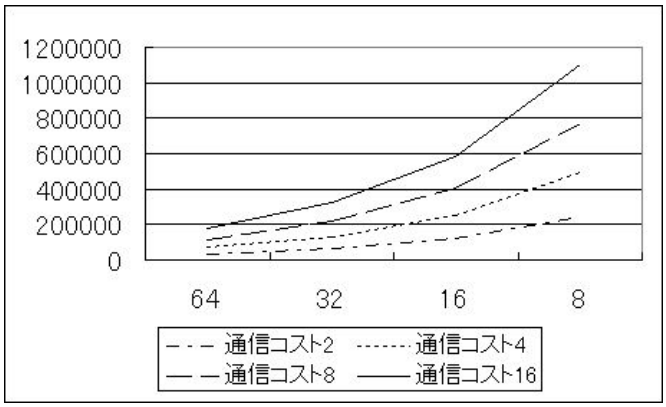

図 4: 通信コスト値の変化

逐次での行列計算では、他のプロセッサから計算に必要な値を受け取り、行列積の演算では通信されるメッセージ数は部分行列の縦の積の長さ、および横の積の長さに比例する。よって、行列に対してプロセッサを割り当てるときには、部分行列の縦横の長さが等しくなるようにしなければならない。

通信遅延値が増加しても、全体の実行時間に大きな影響は受けない。図 4 より通信コストの値は、全体の実行時間に大きな影響を与えることが分かる。

6　結論・今後の課題

BSP モデル上では、通信遅延および同期周期時間を考慮したアルゴリズム構築をしなければならない。本研究の全頂点最短路を求めるアルゴリズムにおいて実行時間に通信コストが大きく関係してきたように、問題ごとに通信コストを減らすよう考慮しなければならない。

7　謝辞

本研究報告書を作成するに当たり、さまざまな御指導や御助言など大変尽力を尽くしていただいた石水隆助手には深く感謝申し上げます。また、同じ研究室の皆には色々とお世話になり感謝申し上げます。

参考文献

[1] 浅野孝夫・今井浩　共著：計算とアルゴリズム, オーム社出版 (2000).

[2] 石水隆・藤原暁宏・井上美智子・増沢利光・藤原秀雄：論文「選択問題を解く BSP モデル及び BSP モデル上での並列アルゴリズム」, 電子情報通信学会論文誌　D-I Vol. J82-D-I No.4 pp.533-542 1999 4 月 (1999).

[3] 岩畑清: ”アルゴリズムとデータ構造, ” 岩波書店, (1989).

[4] J.J á J á: ”An Introduction to Parallel Algorithms,” Addison-Wesley Publishing Company (1992).

[5] 渋沢進：並列分散処理入門, 培風館 (1998).

[6] L.G.Valiant: ”A Bridging Model for Parallel Computing,” Comm. Of the ACM (1990).

A　付録

以下に本研究で作成したBSPモデル上での最短経路問題を解くアルゴリズムの実行をシミュレートするプログラムを示す。

本研究で作成したプログラムは以下の4つから成る。

- BSPMatrix.java　:メインプログラムである。どのプロセッサーがどのような作業を行うかを記述している。
- BSPProcessor.java　:各プロセッサが行う作業を具体的に記述している。
- BSPNetwork.java　:BSPモデルのネットワークの動作を記述している。
- Matrix.java　:入力行列を作成する。

BSPMatrix.java

```
import ioTools.*;  //ファイル入出力用
import java.io.*;  //ファイル入出力用

public class BspMatrix {
  static final int[][] matrixSize = {{16,16},{16,16},{16,16}};
  // 入力行列のサイズ {{行列 A の行, 行列 A の列}, {行列 B の行, 行列
B の列}, {行列 C の行, 行列 C の列}}
  static final int gap = 4;                                  // 通信
コスト
  static final int latency = 16;                             // 通信
遅延
  static final int columnOfProcessors = 2;                   // 縦方
向のプロセッサ数
  static final int rowOfProcessors = 2;                      // 横方
向のプロセッサ数
  static final int matrixsize = 16;

  static BspProcessor[][] processor;                         // プロ
セッサ
  static BspNetwork network;                                 // ネット
ワーク

  static boolean debugSW = false;                            // デバ
グ用スイッチ
```

```
  static boolean traceSW = false;                        // トレ
ス用スイッチ

  static PrintWriter output;                             // 出力
用ファイル
  static PrintWriter log;                                // ログ
出力用ファイル

  public static void main(String[] args) {

    // ネットワークを作る
    network = new BspNetwork(columnOfProcessors, rowOfProcessors);

    // プロセッサを作る
    processor = new BspProcessor[columnOfProcessors][rowOfProcessors];
    for (int p=0; p<columnOfProcessors; p++)
      for (int q=0; q< rowOfProcessors; q++) {
        // プロセッサ(i,j)を作る
        processor[p][q] = new BspProcessor(p, q, columnOfProcessors, rowOfProcessors);
         processor[p][q].setNetwork (network);
        processor[p][q].setMatrixSize (matrixSize[0][0], matrixSize[0][1], matrixSize[1][
      }

    // 入力行列を作る
    Matrix[] matrix = new Matrix[2];
    matrix[0] = new Matrix();
    matrix[0].makenewMatrix(matrixSize[0][0], matrixSize[0][1]);
    //matrix[1] = new Matrix();
    //matrix[1].makenewMatrix(matrixSize[1][0], matrixSize[1][1]);
    matrix[1] = matrix[0];

    // 行列を表示
    System.out.println("Input Matrix");
    matrix[0].dump();
    matrix[1].dumpToFile("InputMatrixA.txt");
    //System.out.println("MatrixB");
    //matrix[1].dump();
    //matrix[1].dumpToFile("InputMatrixB.txt");
```

```
// プロセッサ(i,j)に行列A,行列Bの部分行列割り当て
int[] columnSize = new int[2];
int[] rowSize = new int[2];
columnSize[0] = matrixSize[0][0] / columnOfProcessors;
rowSize[0] = matrixSize[0][1] / rowOfProcessors;
columnSize[1] = matrixSize[1][0] / columnOfProcessors;
rowSize[1] = matrixSize[1][1] / rowOfProcessors;

for (int p=0; p<columnOfProcessors; p++)
  for (int q=0; q<rowOfProcessors; q++) {
    int[][] m0 = matrix[0].getSubMatrix(p*columnSize[0], q*rowSize[0], columnSize[0],
    int[][] m1 = matrix[1].getSubMatrix(p*columnSize[1], q*rowSize[1], columnSize[1],
     processor[p][q].setSubMatrix(m0, m1);
  }

// 時計初期化
for (int p=0; p<columnOfProcessors; p++)
  for (int q=0; q<rowOfProcessors; q++)
    processor[p][q].setTime(0,0,0);

// 出力用ファイル設定
output = FileIo.fWrite("OutputMatrix.txt", false);
log = FileIo.fWrite("BspMatrix.log", false);
network.setLogFile(log);
for (int p=0; p<columnOfProcessors; p++)
  for (int q=0; q<rowOfProcessors; q++) {
    processor[p][q].setOutputFile(output);
    processor[p][q].setLogFile(log);
  }

// 入力行列表示
if (traceSW) {
  System.out.println("MatrixA");
  showMatrix(0);
  System.out.println("MatrixB");
  showMatrix(1);
}
log.println("MatrixA");
logMatrix(0);
```

```
  log.println("MatrixB");
  logMatrix(1);

  //行列演算を実行させる
  if (debugSW) System.out.println("matrixMultiplication");
  log.println("matrixMultiplication");
  matrixMultiplication();



  // 解行列表示
  System.out.println("Output Matrix");
  showMatrix(2);
  log.println("Matrix_out");
  logMatrix(2);
  outputMatrix(2);

  // 実行時間表示
  showTime();

  output.close();
  log.close();
}

public static void matrixMultiplication() {
  int[] columnSize = new int[2];
  int[] rowSize = new int[2];
  columnSize[0] = matrixSize[0][0] / columnOfProcessors;
  rowSize[0] = matrixSize[0][1] / rowOfProcessors;
  columnSize[1] = matrixSize[1][0] / columnOfProcessors;
  rowSize[1] = matrixSize[1][1] / rowOfProcessors;

  // 行列 A の部分行列を横方向のプロセッサに送る
  if (debugSW) System.out.println("sendRowMatrix");
  log.println("sendRowMatrix");
  for (int p=0; p<columnOfProcessors; p++)
    for (int c=0; c<columnSize[0]; c++)
      for (int q=0; q<rowOfProcessors; q++)
        processor[p][q].sendRowMatrix (c);
```

```
synchronous(); // 同期

// 行列積計算に必要な行列 B の横方向の部分行列を受信する
if (debugSW) System.out.println("receiveRowMatrix");
log.println("receiveRowMatrix");
for (int p=0; p<columnOfProcessors; p++)
  for (int q=0; q<rowOfProcessors; q++)
    processor[p][q].receiveRowMatrix ();


// 行列 B の部分行列を縦方向のプロセッサに送る
if (debugSW) System.out.println("sendRowMatrix");
log.println("sendRowMatrix");
for (int p=0; p<columnOfProcessors; p++)
  for (int c=0; c<columnSize[1]; c++)
    for (int q=0; q<rowOfProcessors; q++)
      processor[p][q].sendColumnMatrix (c);

synchronous(); // 同期


// 行列積計算に必要な行列 B の縦方向の部分行列を受信する
if (debugSW) System.out.println("receiveRowMatrix");
log.println("receiveRowMatrix");
for (int p=0; p<columnOfProcessors; p++)
  for (int q=0; q<rowOfProcessors; q++)
    processor[p][q].receiveColumnMatrix ();
if (traceSW) {
  for (int p=0; p<columnOfProcessors; p++)
    for (int q=0; q<rowOfProcessors; q++) {
     System.out.println (processor[p][q].formatProcessorNumber());
      processor[p][q].showTmpMatrix(0);
      processor[p][q].showTmpMatrix(1);
    }
}
for (int p=0; p<columnOfProcessors; p++)
  for (int q=0; q<rowOfProcessors; q++) {
    log.println (processor[p][q].formatProcessorNumber());
```

```
      processor[p][q].logTmpMatrix(0);
      processor[p][q].logTmpMatrix(1);
    }


  if (debugSW) System.out.println("saitankeiro");
  log.println("saitankeiro");

  double k= Math.log(matrixsize -1)/Math.log(2);
  double m = Math.ceil(k)+1;

  for (double i=0; i<m ; i++){

  for (int p=0; p<columnOfProcessors; p++)
    for (int q=0; q<rowOfProcessors; q++)
      processor[p][q].saitankeiro();
    }
  }


// 全てのプロセッサの同期
 public static void synchronous() {
  if (traceSW) {
    System.out.println("Network");
    for (int p=0; p<columnOfProcessors; p++)
      for (int q=0; q<rowOfProcessors; q++)
      network.showSendQueue (p, q);
  }
  log.println("Network");
  for (int p=0; p<columnOfProcessors; p++)
    for (int q=0; q<rowOfProcessors; q++)
      network.logSendQueue (p, q);

  // ネットワークの送信キュー内のデータを受信キューに移す
  network.synchronous();

  int timeI = 0;
  int timeG = 0;
  int timeL = 0;
```

```
    for (int p=0; p<columnOfProcessors; p++) // 全てのプロセッサ
の時間を最大値に揃える
      for (int q=0; q<rowOfProcessors; q++) {
       if (processor[p][q].timeI > timeI) timeI = processor[p][q].timeI;
       if (processor[p][q].timeG > timeG) timeG = processor[p][q].timeG;
       if (processor[p][q].timeL > timeL) timeL = processor[p][q].timeL;
    }

    timeL++;

    for (int p=0; p<columnOfProcessors; p++)
      for (int q=0; q<rowOfProcessors; q++)
        processor[p][q].setTime(timeI, timeG, timeL);
  }

  // 行列 [i] を表示
  public static void showMatrix(int i) {
    int columnSize = matrixSize[i][0] / columnOfProcessors;
    for (int p=0; p<columnOfProcessors; p++) {
      for (int c=0; c<columnSize; c++) {
        for (int q=0; q<rowOfProcessors; q++) {
          processor[p][q].showMatrix(i, c);
          System.out.print(" ");
        }
        System.out.println();
      }
      System.out.println();
    }
  }

  // 行列 [i] を出力 (ログ出力用)
  public static void logMatrix(int i) {
    int columnSize = matrixSize[i][0] / columnOfProcessors;
    for (int p=0; p<columnOfProcessors; p++) {
      for (int c=0; c<columnSize; c++) {
        for (int q=0; q<rowOfProcessors; q++) {
          processor[p][q].logMatrix(i, c);
          log.print(" ");
        }
```

```
        log.println();
      }
      log.println();
    }
  }

  // 行列 [i] を出力
  public static void outputMatrix(int i) {
    int columnSize = matrixSize[i][0] / columnOfProcessors;
    for (int p=0; p<columnOfProcessors; p++) {
      for (int c=0; c<columnSize; c++) {
        for (int q=0; q<rowOfProcessors; q++)
          processor[p][q].outputMatrix(i, c);
        output.println();
      }
    }
  }

  // 実行時間表示
  public static void showTime() {
    System.out.println("time : " + processor[0][0].timeI + " + g*" + processor[0][0].time
  }
}
```

BSPProcessor.java

```
import ioTools.*;  //ファイル入出力用
import java.io.*;  //ファイル入出力用

public class BspProcessor {
  int columnNumber;          // 縦方向のプロセッサ番号
  int rowNumber;             // 横方向のプロセッサ番号
  BspNetwork network;        // ネットワーク
  int columnOfProcessors;    // 縦方向のプロセッサ数
  int rowOfProcessors;       // 横方向のプロセッサ数
  int timeI;                 // 内部計算時間
  int timeG;                 // 通信時間
  int timeL;                 // 同期回数
  int[] columnOfMatrix;      // 行列の行数 columnOfMatrix[0],columnOfMatrix[1]
に入力行列の行数, columnOfMatrix[2] に解行列の行数が入る
```

```
  int[] rowOfMatrix;        // 行列の列数 rowOfMatrix[0],rowOfMatrix[1]
に入力行列の列数, rowOfMatrix[2] に解行列の列数が入る
  int[][][] subMatrix;      // 部分行列 subMatrix[0],subMatrix[1]
に入力行列, subMatrix[2] に解行列が入る
  int[][][] tmpMatrix;      // 行列積計算に必要な入力行列の部分行
列

  PrintWriter output;       // 出力用ファイル
  PrintWriter log;          // ログ出力用ファイル

  // プロセッサ (p,q) を作る
  public BspProcessor (int p, int q, int cp) {
    columnNumber = p;
    rowNumber = q;
    columnOfProcessors = cp;
    rowOfProcessors = cp;
  }

  // プロセッサ (p,q) を作る
  public BspProcessor (int p, int q, int cp, int rp) {
    columnNumber = p;
    rowNumber = q;
    columnOfProcessors = cp;
    rowOfProcessors = rp;
  }

  // ネットワークを設定する
  public void setNetwork (BspNetwork net) {
    network = net;
  }

  // 入力行列のサイズを設定する
  public void setMatrixSize (int c0, int r0, int c1, int r1) {
    if (r0 != c1) executeError ("Illegal matrix size : ("+c0+"
×"+r0+") × ("+c1+"×"+r1+")");
    columnOfMatrix = new int[3];
    rowOfMatrix = new int[3];
    columnOfMatrix[0] = c0;
    rowOfMatrix[0] = r0;
```

```
  columnOfMatrix[1] = c1;
  rowOfMatrix[1] = r1;
  columnOfMatrix[2] = c0;
  rowOfMatrix[2] = r1;
}

// データをセットする
public void setSubMatrix(int[][] m0, int[][] m1) {
  subMatrix = new int[3][][];
  subMatrix[0] = m0;
  subMatrix[1] = m1;
  tmpMatrix = new int[2][][];
}

// 時間をセット
public void setTime(int i, int g, int l) {
  timeI = i;
  timeG = g;
  timeL = l;
}

// 出力用ファイルをセット
public void setOutputFile(PrintWriter o) {
  output = o;
}

// ログ出力用ファイルをセット
public void setLogFile(PrintWriter l) {
  log = l;
}

// 行列 [i] の c 行目を表示
public void showMatrix (int i, int c) {
  for (int r=0; r<subMatrix[i][c].length; r++)
    System.out.print(formatData(subMatrix[i][c][r]));
}

// 行列 [i] の c 行目を出力
public void outputMatrix (int i, int c) {
```

```
  for (int r=0; r<subMatrix[i][c].length; r++)
    output.print(formatData(subMatrix[i][c][r]));
}

// 行列 [i] の c 行目を出力 (ログ出力用)
public void logMatrix (int i, int c) {
  for (int r=0; r<subMatrix[i][c].length; r++)
    log.print(formatData(subMatrix[i][c][r]));
}

// 行列 [0] の c 行目を横方向のプロセッサに送る
public void sendRowMatrix (int c) {
  for (int r=0; r<subMatrix[0][c].length; r++)
    for (int p=0; p<rowOfProcessors; p++)
      network.put(columnNumber, p, subMatrix[0][c][r]);
  timeG += subMatrix[0][c].length * rowOfProcessors;
}

// 行列 [1] の c 行目を縦方向のプロセッサに送る
public void sendColumnMatrix (int c) {
  for (int r=0; r<subMatrix[1][c].length; r++)
    for (int p=0; p<columnOfProcessors; p++)
      network.put(p, rowNumber, subMatrix[1][c][r]);
  timeG += subMatrix[1][c].length * columnOfProcessors;
}

// 行列積計算に必要な入力行列の横方向の部分行列を受信する
public void receiveRowMatrix () {
  tmpMatrix[0] = network.getMatrix (columnNumber, rowNumber, subMatrix[0].length, rowO
  timeG += tmpMatrix[0].length * tmpMatrix[0][0].length;
}

// 行列積計算に必要な入力行列の縦方向の部分行列を受信する
public void receiveColumnMatrix () {
  tmpMatrix[1] = network.getMatrix (columnNumber, rowNumber, columnOfMatrix[1], subMatr
  timeG += tmpMatrix[1].length * tmpMatrix[1][0].length;
}
```

```
//行列から最短経路を求める
public void saitankeiro () {
  int columnSize = columnOfMatrix[2] / columnOfProcessors;
  int rowSize = rowOfMatrix[2] / rowOfProcessors;
  subMatrix[2] = new int [columnSize][rowSize];
  //int element=0;
  //int minkeiro=100000;

  for (int i=0; i<columnSize; i++){
    for (int j=0; j<rowSize; j++) {
      int element=Integer.MAX_VALUE;
      int minkeiro =Integer.MAX_VALUE;

        for (int k=0; k<rowOfMatrix[0]; k++){


         if (tmpMatrix[0][i][k] != Integer.MAX_VALUE && tmpMatrix[1][k][j] != Integer.M

            element =tmpMatrix[0][i][k]+tmpMatrix[1][k][j];
            if (element < minkeiro){
              minkeiro = element;
              subMatrix[2][i][j] = minkeiro;
             }
           }
           else if  (subMatrix[2][i][j] != Integer.MAX_VALUE)
            subMatrix[2][i][j] = minkeiro;

            else
            subMatrix[2][i][j] = Integer.MAX_VALUE;

           }
       }
  }
  timeI += columnSize * rowSize * rowOfMatrix[0];
}

// 入力行列の部分行列を表示する (デバグ用)
public void showSubMatrix (int i) {
  for (int c=0; c<subMatrix[i].length; c++) {
```

```
    for (int r=0; r<subMatrix[i][c].length; r++)
      System.out.print (formatData (subMatrix[i][c][r]));
    System.out.println();
  }
}

// 行列積計算に必要な入力行列の部分行列を表示する(デバグ用)
public void showTmpMatrix (int i) {
  for (int c=0; c<tmpMatrix[i].length; c++) {
    for (int r=0; r<tmpMatrix[i][c].length; r++)
      System.out.print (formatData (tmpMatrix[i][c][r]));
    System.out.println();
  }
}

// 入力行列の部分行列を出力する(ログ出力用)
public void logSubMatrix (int i) {
  for (int c=0; c<subMatrix[i].length; c++) {
    for (int r=0; r<subMatrix[i][c].length; r++)
      log.print (formatData (subMatrix[i][c][r]));
    log.println();
  }
}

// 行列積計算に必要な入力行列の部分行列を出力する(ログ出力用)
public void logTmpMatrix (int i) {
  for (int c=0; c<tmpMatrix[i].length; c++) {
    for (int r=0; r<tmpMatrix[i][c].length; r++)
      log.print (formatData (tmpMatrix[i][c][r]));
    log.println();
  }
}

// プロセッサ番号を出力用に整形する
String formatProcessorNumber () {
  String str;
  if (columnNumber < 10)
    str = "Pr.( "+columnNumber;
  else str = "Pr.("+columnNumber;
```

```
    if (rowNumber <10)
      str += ", "+rowNumber+"):";
    else str += ","+rowNumber+"):";
    return str;
  }


  // データを出力用に整形する
  String formatData (int d) {
    if (d == Integer.MAX_VALUE)
      return "  --";                    // 無限大は"--"を出力
    else if (d < 0) {
      if (d > -10) return "  "+d;
      else if (d > -100) return " "+d;
      else return ""+d;
    } else {
      if (d < 10) return "   "+d;
      else if (d < 100) return "  "+d;
      else return " "+d;
    }
  }

  void executeError (String err_mes) { /* 実行時エラー */
    System.out.println("Execute error");
    System.out.println(err_mes);
    System.exit(1);
  }
}
```

BSPNetwork.java

```
import java.util.ArrayList;  // ArrayList 処理用
import ioTools.*;  //ファイル入出力用
import java.io.*;  //ファイル入出力用

public class BspNetwork {
  ArrayList[][] sendQueue;          // プロセッサ (i,j) へ送信中
のデータ
```

```
  ArrayList[][] receiveQueue;        // プロセッサ (i,j) が受信中
のデータ

  PrintWriter log;                   // ログ出力用ファイル

  // p × p 台のプロセッサを結ぶネットワークを作る
  public BspNetwork(int p) {
    sendQueue = new ArrayList[p][p];
    receiveQueue = new ArrayList[p][p];
    for (int i=0; i<p; i++)
      for (int j=0; j<p; j++) {
        sendQueue[i][j] = new ArrayList();
        receiveQueue[i][j] = new ArrayList();
    }
  }

  // p × q 台のプロセッサを結ぶネットワークを作る
  public BspNetwork(int p, int q) {
    sendQueue = new ArrayList[p][q];
    receiveQueue = new ArrayList[p][q];
    for (int i=0; i<p; i++)
      for (int j=0; j<q; j++) {
        sendQueue[i][j] = new ArrayList();
        receiveQueue[i][j] = new ArrayList();
    }
  }

  // int 型データ n をプロセッサ (p,q) への送信キューに置く
  public void put (int p, int q, int n) {
    sendQueue[p][q].add(new Integer(n));
  }

  // int 型配列データ a をプロセッサ (p,q) への送信キューに置く
  public void putArray (int p, int q, int[] a) {
    for (int i=0; i<a.length; i++)
      sendQueue[p][q].add(new Integer(a[i]));
  }

  // 2 個組 int 型データ (m,n) をプロセッサ (p,q) への送信キューに置く
```

```
  public void putPair(int p, int q, int m, int n) {
    sendQueue[p][q].add(new Integer(m));
    sendQueue[p][q].add(new Integer(n));
  }

  // int 型配列データ a の a[l]〜a[h] をプロセッサ (p,q) への送信キュー
に置く
  // l,h が 0 未満または a.length 以上のときはエラー
  public void putPartOfArray (int p, int q, int[]a, int l, int h) {
    if (l<0 || l>=a.length || h<0 || h>=a.length)
      executeError("Illegal index of array at Queue "+p);
    for (int i=l; i<=h; i++) {
      sendQueue[p][q].add(new Integer(a[i]));
    }
  }

  // int 型行列データ m をプロセッサ (p,q) への送信キューに置く
  public void putMatrix (int p, int q, int[][]m) {
    for (int i=0; i<m.length; i++)
      for (int j=0; j<m[i].length; j++)
         sendQueue[p][q].add(new Integer(m[i][j]));
  }

  // プロセッサ (p,q) への受信キューから int 型データを取り出す
  // 受信キューにデータが入っていない場合、取り出したデータが Integer
型でない場合はエラー
  public int get (int p, int q) {
    if (receiveQueue[p][q].isEmpty()) executeError("Queue "+p+" Underflow");
    if ((receiveQueue[p][q].get(0)).getClass() !=  Integer.class) executeError("Queue "+p
    int n = ((Integer) receiveQueue[p][q].get(0)).intValue();
    receiveQueue[p][q].remove(0);
    return n;
  }

  // プロセッサ (p,q) への受信キューから int 型配列データを取り出す
  // 受信キューにデータが入っていない場合、取り出したデータが Integer
型でない場合はエラー
  public int[] getArray (int p, int q) {
    if (receiveQueue[p][q].isEmpty()) executeError("Queue "+p+" Underflow");
```

```
    int[] a = new int[receiveQueue[p][q].size()];
    for (int i=0; i<a.length; i++) {
      if ((receiveQueue[p][q].get(0)).getClass() !=  Integer.class) executeError("Queue "
      a[i] = ((Integer) receiveQueue[p][q].get(0)).intValue();
      receiveQueue[p][q].remove(0);
    }
    return a;
  }

  // プロセッサ (p,q) への受信キューから 2 個組 int 型データを取り出す
  // 受信キューに 2 個以上データが入っていない場合、取り出したデータ
が Integer 型でない場合はエラー
  public int[] getPair (int p, int q) {
    if (receiveQueue[p][q].size() < 2) executeError("Queue "+p+" Underflow");
    if ((receiveQueue[p][q].get(0)).getClass() !=  Integer.class) executeError("Queue "+p
    int m = ((Integer) receiveQueue[p][q].get(0)).intValue();
    receiveQueue[p][q].remove(0);
    if ((receiveQueue[p][q].get(0)).getClass() !=  Integer.class) executeError("Queue "+p
    int n = ((Integer) receiveQueue[p][q].get(0)).intValue();
    receiveQueue[p][q].remove(0);
    int[] intPair = {m,n};
    return intPair;
  }

  // プロセッサ (p,q) への受信キューから長さ s の int 型配列データを取
り出す
  // 受信キューに s 個以上データが入っていない場合、取り出したデータ
が Integer 型でない場合はエラー
  public int[] getArray (int p, int q, int s) {
    if (receiveQueue[p][q].size() < s) executeError("Queue "+p+" Underflow");
    int[] a = new int[s];
    for (int i=0; i<s; i++) {
      if ((receiveQueue[p][q].get(0)).getClass() !=  Integer.class) executeError("Queue "
      a[i] = ((Integer) receiveQueue[p][q].get(0)).intValue();
      receiveQueue[p][q].remove(0);
    }
    return a;
  }
```

```
  // プロセッサ (p,q) への受信キューからサイズ s × s の int 型行列デー
タを取り出す
  // 受信キューに s*s 個以上データが入っていない場合、取り出したデー
タが Integer 型でない場合はエラー
  public int[][] getMatrix (int p, int q, int s) {
    if (receiveQueue[p][q].size() < s*s) executeError("Queue "+p+" Underflow");
    int[][] m = new int[s][s];
    for (int i=0; i<s; i++)
      for (int j=0; j<s; j++) {
        if ((receiveQueue[p][q].get(0)).getClass() !=  Integer.class) executeError("Queue
        m[i][j] = ((Integer) receiveQueue[p][q].get(0)).intValue();
         receiveQueue[p][q].remove(0);
    }
    return m;
  }

  // プロセッサ p への受信キューからサイズ s × t の int 型行列データを
取り出す
  // 受信キューに s*t 個以上データが入っていない場合、取り出したデー
タが Integer 型でない場合はエラー
  public int[][] getMatrix (int p, int q, int s, int t) {
    if (receiveQueue[p][q].size() < s*t) executeError("Queue "+p+" Underflow");
    int[][] m = new int[s][t];
    for (int i=0; i<s; i++)
      for (int j=0; j<t; j++) {
        if ((receiveQueue[p][q].get(0)).getClass() !=  Integer.class) executeError("Queue
        m[i][j] = ((Integer) receiveQueue[p][q].get(0)).intValue();
         receiveQueue[p][q].remove(0);
    }
    return m;
  }

  // 全てのプロセッサで同期を取る
  // 全てのプロセッサの送信キューのデータを受信キューに移す
  //（この操作を行わないと受信キューからデータを取り出せない）
  public void synchronous() {
    for (int i=0; i<sendQueue.length; i++)
      for (int j=0; j<sendQueue[i].length; j++)
        while (!(sendQueue[i][j].isEmpty())) { // 送信キューが空
```

```
になるまで
          receiveQueue[i][j].add(sendQueue[i][j].get(0));
          sendQueue[i][j].remove(0);
        }
  }

  // 横方向ののプロセッサで同期を取る
  // プロセッサ (p,0)〜プロセッサ (p, rowSize-1) の送信キューのデータを受信キューに移す
  //（この操作を行わないと受信キューからデータを取り出せない）
  public void rowSnchronous(int p) {
    for (int i=0; i<sendQueue[p].length; i++)
      while (!(sendQueue[p][i].isEmpty())) { // 送信キューが空になるまで
        receiveQueue[p][i].add(sendQueue[p][i].get(0));
        sendQueue[p][i].remove(0);
      }
  }

  // 縦方向ののプロセッサで同期を取る
  // プロセッサ (0,q)〜プロセッサ (columnSize-1, q) の送信キューのデータを受信キューに移す
  //（この操作を行わないと受信キューからデータを取り出せない）
  public void columnSnchronous(int q) {
    for (int i=0; i<sendQueue.length; i++)
      while (!(sendQueue[i][q].isEmpty())) { // 送信キューが空になるまで
        receiveQueue[i][q].add(sendQueue[i][q].get(0));
        sendQueue[i][q].remove(0);
      }
  }

  // プロセッサ (p,q) への受信キューをクリアする
  public void clear (int p, int q) {
    receiveQueue[p][q].clear();
  }

  // プロセッサ (p,q) への受信キューが空であるか？
  public boolean isEmpty (int p,int q) {
```

```
  return receiveQueue[p][q].isEmpty();
}

// プロセッサ (p,q) への受信キューに置かれたデータの数を返す
public int size (int p, int q) {
  return receiveQueue[p][q].size();
}

// ログ出力用ファイルをセットする
public void setLogFile(PrintWriter l) {
  log = l;
}

// プロセッサ (p,q) への送信キューの内容を表示する (デバグ用)
public void showSendQueue (int p, int q) {
  System.out.print(formatProcessorNumber(p, q));
  if (sendQueue[p][q].isEmpty())
    System.out.println(" Empty");
  else {
    int n = sendQueue[p][q].size();
    for (int i=0; i<n; i++)
      System.out.print (formatData (sendQueue[p][q].get(i)));
    System.out.println();
  }
}

// プロセッサ p への受信キューを表示する (デバグ用)
public void showReceiveQueue (int p, int q) {
  System.out.print(formatProcessorNumber(p, q));
  if (receiveQueue[p][q].isEmpty())
    System.out.println(" Empty");
  else {
    int n = receiveQueue[p][q].size();
    for (int i=0; i<n; i++)
      System.out.print (formatData (receiveQueue[p][q].get(i)));
    System.out.println();
  }
}
```

```
// プロセッサ p への送信キューをログファイルに出力する
public void logSendQueue (int p, int q) {
  log.print(formatProcessorNumber(p, q));
  if (sendQueue[p][q].isEmpty())
    log.println(" Empty");
  else {
    int n = sendQueue[p][q].size();
    for (int i=0; i<n; i++)
      log.print (formatData (sendQueue[p][q].get(i)));
    log.println();
  }
}

// プロセッサ p の受信キューをログファイルに出力する
public void logReceiveQueue (int p, int q) {
  log.print(formatProcessorNumber(p, q));
  if (receiveQueue[p][q].isEmpty())
    log.println(" Empty");
  else {
    int n = receiveQueue[p][q].size();
    for (int i=0; i<n; i++)
      log.print (formatData (receiveQueue[p][q].get(i)));
    System.out.println();
  }
}

// プロセッサ番号を出力用に整形する
String formatProcessorNumber (int p, int q) {
  String str;
  if (p < 10)
    str = "Pr.( "+p;
  else str = "Pr.("+p;
  if (q <10)
    str += ", "+q+"):";
  else str += ","+q+"):";
  return str;
}

// データを出力用に整形する
```

```
  String formatData (Object o) {
    if (o.getClass() ==  Integer.class) {
      int val = ((Integer) o).intValue();
      if (val == Integer.MAX_VALUE) return " --";          // 無
限大は"--"を出力
      else if (val < 0) {
        if (val > -10) return " "+val;
        else return ""+val;
      } else {
        if (val < 10) return "  "+val;
        else return " "+val;
      }
    } else return " ??";
  }

  static void executeError (String err_mes) { /* 実行時エラー */
    System.out.println("Execute error");
    System.out.println(err_mes);
    System.exit(1);
  }
}
```

BSPNetwork.java

```
import ioTools.*;           // ファイル入出力用
import java.io.*;           // ファイル入出力用
import java.util.*;         // 文字列処理用

public class Matrix {
  static final int range = 9;        // 値の範囲
                 // 各要素には-range+1〜range-1 の値が与えられる
                 // 隣接行列の場合は各要素には 0〜range-1 の重みが
与えられる
  final double probability = 0.3;     // 隣接行列の辺の存在確率 (0:
辺存在せず 1:辺が必ず存在)
  int columnSize;                     // 行列の行数
  int rowSize;                        // 行列の列数
  int[][] element;                    // 行列の要素
```

```
public Matrix () {
}
//最短経路問題を解くためのｎ×ｎの行列
public void makenewMatrix(int n) {
  columnSize = n;
  rowSize = n;
  element = new int[n][n];
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      //int v = randomValue();
      int v = randomWeight();
      if (i == j)
        element[i][i] = 0;
      else
      element[i][j] = v;
    }
  }
}

  public void makenewMatrix(int n, int m) {
  columnSize = m;
  rowSize = n;
  element = new int[m][n];
  for (int i=0; i<m; i++) {
    for (int j=0; j<n; j++) {
      //int v = randomValue();
      int v = randomWeight();
      if (i == j)
      element[i][i] = 0;
      else
      element[i][j] = v;
    }
  }
}


// デフォルトファイルからデータを読み込む
public void readMatrix () {
  readMatrix("Matrix.txt");
```

```
  }

  // 指定したファイルからデータを読み込む
  public void readMatrix (String fileName) {
    BufferedReader buffer = FileIo.fRead(fileName);
    String line = readLine(buffer);
    StringTokenizer st = new StringTokenizer(line);
    String s = st.nextToken();
    columnSize = Integer.parseInt(s);          // ファイルの先頭
行に書かれた最初の値を行列の列数とする
    if (st.hasMoreTokens()) {
      s = st.nextToken();
      rowSize = Integer.parseInt(s);         // ファイルの先頭行に
書かれた次の値を行列の行数とする
    } else rowSize = columnSize;               // ファイルの先頭行
に 1 つしか値が無ければ正方行列とする
    element = new int[columnSize][rowSize];
    for (int i=0; i<columnSize; i++) {
      line = readLine(buffer);
      st = new StringTokenizer(line);
      for (int j=0; j<rowSize; j++) {
        s = st.nextToken();
        if (s.equals("--"))             // "--" は無限大の値とする
          element[i][j] = Integer.MAX_VALUE;
        else element[i][j] = Integer.parseInt(s);
      }
    }
  }


  // ファイルから 1 行読み込む
  String readLine(BufferedReader buffer) {
    String line = "";
    try {
      line = buffer.readLine();
    } catch(IOException error_report) {
            /* 読込みエラーが発生したら，キャッチした例外を表示し，
               ファイルを閉じ，処理系を終了させる */
      System.out.println(error_report);
```

```
      System.exit(1);
    }
    return line;
  }

  // 確率 probability で 0〜range-1 を返し、確率 1-probability で無
限大を返す
  int randomWeight () {
    if (Math.random() < probability)
      return (int) (range*Math.random()+1); // 0〜range-1 までの
乱数を返す
    else return Integer.MAX_VALUE;          // 辺が存在しない場
合は無限大を返す
  }

  // 行列を返す
  public int[][] getMatrix () {
    return element;
  }

  // (x,y) を始点とするサイズ s × s の部分行列を返す
  // x+s>columnSize または y+s>rowSize のときはエラー
  public int[][] getSubMatrix (int x, int y, int s) {
    if (x+s>columnSize || y+s>rowSize) executeError("Illegal index ("+(x+s)+","+(y+s)+")"
    int[][] m = new int[s][s];
    for (int i=0; i<s; i++)
      for (int j=0; j<s; j++)
        m[i][j] = element[x+i][y+j];
    return m;

  }

  // (x,y) を始点とするサイズ s × s の部分行列を返す
  // x+s>columnSize または y+s>rowSize のときはエラー
  public int[][] getSubMatrix (int x, int y, int s, int t) {
    if (x+s>columnSize || y+t>rowSize) executeError ("Illegal index ("+(x+s)+","+(y+t)+")
    int[][] m = new int[s][t];
    for (int i=0; i<s; i++)
      for (int j=0; j<t; j++)
```

```
        m[i][j] = element[x+i][y+j];
    return m;

  }

  // 行列を表示する
  public void dump () {
    for (int i=0; i<columnSize; i++) {
      for (int j=0; j<rowSize; j++)
        System.out.print(formatData (element[i][j]));
      System.out.println();
    }
  }

  // 行列をデフォルトファイルに出力する
  public void dumpToFile () {
    dumpToFile ("Matrix.txt");
  }

  // 行列を指定したファイルに出力する
  public void dumpToFile (String fileName) {
    PrintWriter outputFile = FileIo.fWrite(fileName, false);
    if (columnSize == rowSize)     // ファイルの先頭に行列のサイ
ズを出力する
      outputFile.println(columnSize);
    else outputFile.println(columnSize+" "+rowSize);
    for (int i=0; i<columnSize; i++) {
      for (int j=0; j<rowSize; j++)
        outputFile.print(formatData (element[i][j]));
      outputFile.println();
    }
    outputFile.close();
  }

  // データを出力用に整形する
  String formatData (int d) {
    if (d == Integer.MAX_VALUE)
      return "  --";                    // 無限大は"--"を出力
    else if (d < 0) {
```

```
      if (d > -10) return "  "+d;
      else if (d > -100) return " "+d;
      else return ""+d;
    } else {
      if (d < 10) return "   "+d;
      else if (d < 100) return "  "+d;
      else return " "+d;
    }
  }

  static void executeError (String err_mes) { /* 実行時エラー */
    System.out.println("Execute error");
    System.out.println(err_mes);
    System.exit(1);
  }

  public static void main (String[] args) {
    System.out.print("Size ? ");
    int n = Console.ReadInteger();

    // 行列を作る
    Matrix matrix = new Matrix();
    //matrix.makeMatrix(n);
    matrix.makenewMatrix(n);

    // 行列を表示
    matrix.dump();
    matrix.dumpToFile();
  }
}
```