卒業研究報告書

題 目

MPIを用いた並列処理

指 導 教 員

石 水　隆 助教

報 告 者

06–1–037–0246

杉 所　拓 也

近畿大学理工学部情報学科

平成23年1月28日提出

概要

計算機は常に高い処理性能が求められている。処理性能の向上の手段として、限界まで 1 台の計算機の処理性能を向上させる方法と、複数のプロセッサを持つ並列計算機を使用しての並列計算がある。しかし、1 台の計算機の処理性能の向上には理論上の限界があり、また、性能の向上には時間と資金がかかってしまう。そこで、もう 1 つの手段である並列計算が重視されている。だが、高い処理性能を持つ並列計算機はとても高価なものであり、容易には使用できない。そのため、複数の安価な計算機をネットワーク接続し、計算機群全体を仮想的な並列計算機として利用できるクラスタ (Cluster) 環境が注目されている。

本研究では、クラスタによる仮想並列計算環境の構築がが可能となるソフトウェアの 1 つである MPI(Message Passing Interface) を用い、その性能を検証する。

MPI の有用性を検証するために、MPI 上で問題を解く時間の計測を行なう。検証を行なうための問題としては、本研究では最小全域木問題を用いる。最小全域木問題とは、重み付無向グラフが与えられたとき、全ての頂点を含む部分木のうち、辺の重みの総和が最小なものを求める問題である。本研究では、グラフ頂点数が 5、10、20、40、80、160 それぞれの場合で、計算機 1〜5 台を用いた場合の MPI 上で最小全域木問題を解く処理時間を計測した。

目次

1 **序論** 1
1.1 本研究の背景 1
1.2 本研究の目的 2
1.3 本報告書の構成 3

2 **準備** 4
2.1 MPI(Messahe Passing Interface) 4
2.2 使用機器 4
2.3 MPICH2 の導入 4
2.4 MPICH2 の実行方法 5
2.5 Visual C++2008Express Edition の導入 6
2.6 木 6
2.7 最小全域木問題を解く MPI プログラム 8

3 **計測結果** 9
3.1 内部処理時間 9
3.2 全体の処理時間 9
3.3 理論値との比較 10

4 **考察** 12

5 **結論・今後の課題** 13

謝辞 14

参考文献 15

付録 A 最小全域木問題を解く MPI **プログラム** 16

1　序論

1.1　本研究の背景

1.1.1　並列処理

あるタスクに対する処理において複数のプロセッサを用い、分割して処理を行なうことにより、単一のプロセッサでの処理よりも高速に計算処理を行なうことを並列処理 (Parallel Processing) という。

近年、計算機の性能の向上は著しいが、いずれ限界が来ることはすでに指摘されている。単一のプロセッサの性能に滞ることなく、処理速度を向上させる手段として並列処理が挙げられる。すでに並列処理は天体の軌道観測や地球の気象シミュレーションなどの逐次処理では膨大な時間がかかる問題において、逐次計算機よりも短時間で解けることから利用されている。

1.1.2　並列計算機

あるタスクに対する処理において複数のプロセッサを用いることで並列処理を行なうことが可能な計算機を並列計算機 (Parallel Computer) という。

並列計算機上で行なう並列処理には、共有メモリ型並列処理 (Shared Memory Parallel Processing) と分散メモリ型並列処理 (Distributed Memory Parallel Processing) の２つに大きく分けることが出来る。共有メモリ型並列処理とは、全てのプロセッサが１つのメモリを共有して使用する並列処理である。この方法ではプロセッサ間の同期の問題やデータの送受信などのオーバヘッドにはメモリを共有しているため容易に対処できる。しかし、プロセッサ数の増減などの変化に対して、全てのプロセッサをメモリに接続させることが困難である。共有メモリ型並列処理モデルとして PRAM(Parallel Random Access Machine)[1] などが挙げられる。分散メモリ型並列処理とは、それぞれのプロセッサが個々に局所メモリを持ち、プロセッサ間の通信にはネットワークを使用する並列処理である。この方法では各プロセッサの同期の問題やデータの送受信時間がそのままオーバヘッドとして処理時間に影響してしまう。しかし、共有メモリ型並列処理の効率的な実装が困難なことから、現在は実装が容易な分散メモリ型並列処理が主流となっている。また、分散メモリ並列処理モデルとして BSP(Bulk-Synchronous Parallel)[3][4]CGM(Coarse Grain Multi-Computer)[5] などが挙げられる。

共有型メモリ並列計算機、分散メモリ型並列計算機共に現存する様々な並列計算機が存在する。しかし、一般的に並列計算機は非常に高価であり、並列計算機を持つことが出来るのは一部の大学、研究機関、そして企業など、資金力を持つ組織に限られる。このため、多くのユーザは手軽に並列計算機を使うことができず、並列計算機は注目されるものの広く普及されることはなかった。だが今日、複数の計算機をネットワーク接続し計算機群全体を１台の仮想並列計算機とするクラスタ (Cluster) 環境が、注目されておりクラスタ環境を構築するソフトウェアも開発されている。

クラスタ環境を構築するソフトウェアの中には、無料で提供されているものもあるため、現在では多くのユーザが並列計算機を身近に利用できるようになった。そのようなソフトウェアとしては共有メモリ型並列処理用として OpenMP[6]、分散メモリ型並列処理用として MPI[2] や PVM(Parallel Virtual Machine)[14]、OpenMosix[15]、Score[16] などがある。

1.1.3　仮想並列計算機を構築するソフトウェア

MPI(MJassage Passing Interface)[2][7] は並列計算におけるメッセージ (データ) 通信のプログラムを記述するための規約を設けるために開発された規格である。MPI は 1992 年に結成された MPI フォーラムにより、並列処理に関する標準的使用の定義、および検討をされることで具体化された。MPI は MPI 標準と呼ばれるインタフェース規格であり、この MPI 標準に準じて実装されたライブラリを MPI ライブラリと呼ぶ。MPI の開発には欧米の 40 近い組織から 60 人ほどの人間が関わっており、研究者や主な並列計算機ベンダのほとんどが参加した。

本来、計算機間で通信を行なう場合、計算機のアーキテクチャによりプロトコルの違いが応じ、計算機ごとにプログラムを書き分ける必要があった。しかし、MPI ライブラリには様々な通信関数が実装されているため、MPI 規約を用いて作成したプログラムは MPI によりプロトコルの障害を考慮することなく、ユーザは並列処理アルゴリズムに集中してプログラム作成することが出来る。

PVM(Parallel Virtual Machine)[14] は、ネットワークに接続された複数台の計算機を、単一の計算機として利用することができるようになるものである。このことにより、複数台の計算機が持つ計算能力を 1 つの大規模計算問題に結集して処理を行なうことができる。

OpenMP[6] は、共有メモリ型並列計算機を用いて並列を行なうものである。また、OpenMP が使用できない環境では無視されるディレクティブを挿入することによって並列化を行なう。このため並列環境と非並列環境でほぼ同一のソースコードを使用できるという利点がある。

OpenMosix[15] は Linux プロセスをネットワーク経由で他のクラスタノードに実行させる仕組であり、動的な負荷分散を自動的に行なってくれるので、複数の PC をあたかも 1 台の PC として利用し、最大限に性能を引き出すことができる。

Score[16] は、Linux 用クラスタ計算機用超並列プログラム実行環境のことである。実行環境とは、並列プログラムが動作するための共通 API 使用に基づいたライブブリ群や補助ツール群を動作させる基盤のことである。

上記に挙げた仮想並列計算環境を構築するソフトウェアの中では、現在 MPI が主流となっている。そこで本研究では、MPI を用いる。MPI は専用の並列計算機からワークステーションやパーソナルコンピュータなど多様な機種をサポートしており、無料で提供されているものも多く、主な実装には MPICH[8][9] や LAM[12][13]、OpenMPI[17] などがある。また、MPI を使用できるプログラミング言語も多く、C 言語や Java など多様に対応している。

1.2　本研究の目的

本研究では、仮想並列計算機の有用性を検証するために MPI を用いて、複数台の計算機をネットワーク上において構成し、仮想並列計算機の性能を実験的に評価する。本研究における評価方法としては、MPI を構築するソフトウェアの 1 つである MPICH2[8] を用いて、MPI 環境を構築し、MPI 上で問題を解く時間を計測することによって、MPI による時間短縮効果の検証を行なっている。検証を行なうための問題としては、最小全域木問題を用いる。

1.2.1 最小全域木問題

本研究では、MPI の性能を検証するための対象問題として最小全域木問題を用いる。最小全域木問題は、重み付無向グラフ $G = (V, E)$ が与えられたとき、G の全ての頂点を含む部分木のうち、辺の重みの総和が最小なものである「最小全域木」を求めるものである。最小全域木問題に対して、頂点数 $|V| = n$、辺数 $|E| = m$ の重み付無向グラフに対し、Prim は $O(m + n \log n)$、Kruskal は $O(m \log m)$、Sollin は $O(n^2)$ の逐次アルゴリズムを提案した [1]。また、Sollin のアルゴリズムからは、CREW PRAM 上で、p プロセッサ $O(\frac{n^2}{p} + \log^2 n)$ 時間で解く並列アルゴリズムアルゴリズムが得られる [1]。

1.3 本報告書の構成

本報告者の構成を以下に述べる。2 章に本研究での環境と、MPI の導入方法と使用機器、および検証用の問題である最小全域木問題について述べる。3 章に並列計算による処理時間に関する計測結果を示す。4 章では考察を行ない、5 章では本研究の結論と今後の課題を述べる。

2 準備

2.1 MPI(Messahe Passing Interface)

MPI (Message Passing Interface)[2] は 1991 年に計算機間のメッセージ通信の標準規格として開発され、並列処理利用をするための標準化された規格である。複数の計算機が情報をバイト列からなるメッセージとして送受信することで協調動作を行なうことができる。

無料で提供されている MPI の主な実装として、MPICH2[8] や LAM[12][13]、OpenMPI[17] 等がある。

MPICH は MPI を実装するためのソフトウェアとして Argonne National Laboratory[8][9] で開発された。2005 年には MPICH の後継として MPICH2 が開発され MPICH よりも故障に強く、効果的で、使いやすい。MPICH2 は、プロセス管理とコミュニケーションを完全に分離している。デフォルトの実行環境は mpd と呼ばれるデーモンより成り、これはジョブが実行される前にノード間で接続を確立するプロセスである。このプロセスにより、高速でスケーラブルな実行環境の確率を可能にし、問題発生時にも問題の原因究明が容易に行なえる。

LAM は、(Language Analysis Manager) の略であり各国語版の解析ライブラリと共に用いる。日本語版では「日本語解析ライブラリ」と共に形態素解析を行なう。

OpenMPI は高性能性能メッセージパッシングライブラリである。維持されているオープンソースを実装する。

本研究では、MPI を実装するソフトウェアとして、MPICH2 を用いる。MPICH2 を使用するのは、無料であり広く普及しているソフトウェアであること、MPI の特徴でもある移植性の高さをそのまま利用できる、MPI-2 の機能に対するサポートが充実している、Windows のオペレーションシステムシリーズでも利用することができる、などの理由からである。

2.2 使用機器

本研究で使用する計算機は、オペレーションシステムとしてマイクロソフト社の提供販売しているオペレーションシステムである Windows Vista と Windows XP を使用する。Windows Vista と Windows XP を使用する理由は、一般家庭や公共機関、企業などの場所で他のオペレーションシステムより広く普及しているからである。また、MPI を使用して仮想並列計算機を構築する場合に、全ての計算機が同じオペレーションシステムを実装しておかなければ実行できないこともあり、オペレーションシステムの統一が容易であるという理由もある。

本研究では、計算機 5 台を 100Base-TX により LAN 接続し MPI 環境を構築する。本研究において、使用する計算機の名前、性能などを表 1、図 1 に記す。

2.3 MPICH2 の導入

本節では MPICH2 のインストール手順と環境設定について述べる。

まず MPI 環境を構築する全ての計算機に MPICH2[8] のページから使用するオペレーティングシステムに合うインストール用ファイルをダウンロードする。本研究では WindowsOS を使用したので、Windows 用の最新バージョンを用いた。ダウンロード後、実行形式のインストール用ファイルを実行することにより、イン

表 1　本研究で使用した計算機一覧

	メイン PC	サブ PC1	サブ PC2	サブ PC3	サブ PC4
OS	Windows Vista	Windows Vista	Windows Vista	Windows XP	Windows Vista
CPU	Core2 1.4GHz	Core2 1.4GHz	Core2 1.4GHz	Pentium 1.6GHz	Core2 1.4GHz
RAM	1GB	1GB	1GB	512MB	1GB

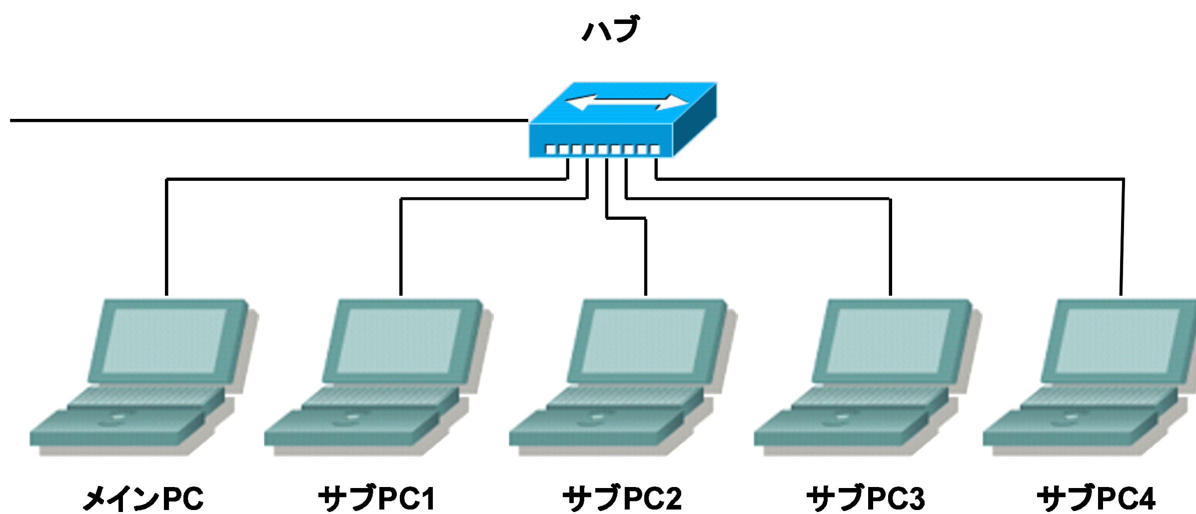

図 1　MPI 環境を構築する計算機

ストーラが起動する。デフォルトでは C:¥Program Files¥MPICH2 フォルダにインストールを行なわれる設定になっているので、インストール用実行形式ファイルの実行時に環境に合わせて適当なフォルダを指定する。

インストール後は環境設定を行なわなければならない。この設定は環境変数 PATH の指定を、MPICH2 のインストールしたフォルダの下にある bin(本研究では "C:¥Program Files¥MPICH2¥bin") フォルダに対して設定すればよい。この設定の確認は PATH 指定を行なった後、コマンドプロンプト上で mpiexec コマンドを実行したときに "Usage" と表示されれば、PATH 指定が成功したことがわかる。

また、使用する計算機に共通のアカウント名とパスワードを持つユーザを用意する必要がある。加えてプログラムの実行には、全ての計算機に共通したファイル構成で、実行ファイルを置く必要がある。そのため、実行ファイルの受け渡しなどをネットワーク上で行えるようにするために、共有フォルダを用意することが望ましい。

2.4　MPICH2 の実行方法

始めに "C:¥Program Files¥MPICH2¥bin" 内にある wmpiconfig.exe を実行することにより参加する計算機の接続を行う。さらに使用する Domain を指定し、次に Get Hosts を行なうと、現在その Domain に参加している計算機が表示される。次に Scan Hosts を行ない、計算機同士が繋がっている事を確認した後、Apply All を行なうことで、この wmpiconfig 内での作業は終了となる。

次はコマンドプロンプトでの実行方法である。1 台だけで動かす場合は mpiexec -localonly と指定することで実行できる。複数台で動かす場合は mpiexec -hosts プロセッサ数計算機 1 の名前、計算機 2 の名前、… というように指定することで実行することができる。

2.5 Visual C++2008Express Edition の導入

本研究で作成する並列計算処理に使用する MPI プログラムは C++ 言語を用いる。C++ 言語のコンパイルツールには Microsoft 社製の Visual C++2008Express Edition[10] を使用する。プログラミングをするにあたって Visual C++ から MPICH2 を使用できるように設定しなければならない。まず、Visual C++ 上で空のプロジェクトを作成し、ツールオプションからインクルードファイルとライブラリファイルを MPICH2 フォルダにある lib と include フォルダを指定し追加を行なう。次にプロジェクトの設定でリンカ入力の依存ファイルである mpi.lib を追加する。これらの設定を行なうことで MPICH2 による並列プログラミングが可能となる。

2.6 木

2.6.1 重み付無向グラフ

グラフ $G = (V, E)$ において、各頂点間に存在する辺 $(u, v) \in E$ $(u, v \in V)$ にコスト、すなわち重みが付いているグラフを、重み付グラフという。例を出すとするならば、電車の乗換案内図において、駅を頂点、駅と駅を繋ぐ線路を辺とした場合に、駅間の所要時間が「重み」にあたる。

また、グラフ G において、辺 (u, v) が存在するならば辺 (v, u) も存在するとき、G を無向グラフという。すなわち、無向グラフと各辺の始点と終点がどちらであるかを重視しないグラフである。このようなとき、平面上の幾何学的表現では各辺を表現する矢印から矢印を取り除き、そのグラフを表現する。

この 2 つの特徴、「重み付グラフ」と「無向グラフ」を持ち合わせたグラフが「重み付無向グラフ」である。辺 (u, v) に付加された重みを $w(u, v)$ とするとき、重み付無向グラフ G の各辺において、$w(u, v) = w(v, u)$ となる。

2.6.2 最小全域木問題

本研究では、MPI の性能を検証するための対象問題として最小全域木問題を用いる。

最小全域木問題とは、重み付無向グラフ $G = (V, E)$ が与えられたとき、G の全ての頂点を含む部分木のうち、辺の重みの総和が最小なものである「最小全域木」を求めるものである。「全域」とは、G の全ての頂点が辺を通して繋がっているものを指し、「木」とは、連結でかつ閉路が無いグラフを指す。

最小全域木問題の例を図 2 に示す。入力として、図 2 の左の重み付無向グラフが与えられたとき、その最小全域木は図 2 の右の木となる。この木の辺の重みの合計は 13 であり、与えられたグラフに対する全域木の中では最小となっている。

最小全域木問題を解く主なアルゴリズムとして、Prim のアルゴリズム、Kruskal のアルゴリズム、Sollin のアルゴリズム [1] などがある。頂点数 $|V| = n$、辺数 $|E| = m$ の重み付無向グラフに対し、RAM 上で Prim のアルゴリズムの計算量は $O(m + n \log n)$、Kruskal のアルゴリズムの計算量は $O(m \log m)$、Sollin のアルゴリズムの計算量は $O(n^2)$ で最小全域木問題を解くことができる。また、Sollin のアルゴリズムは多少の変更を加えることで PRAM 上の並列アルゴリズムにすることができ、CREW PRAM 上で p プロセッサを用

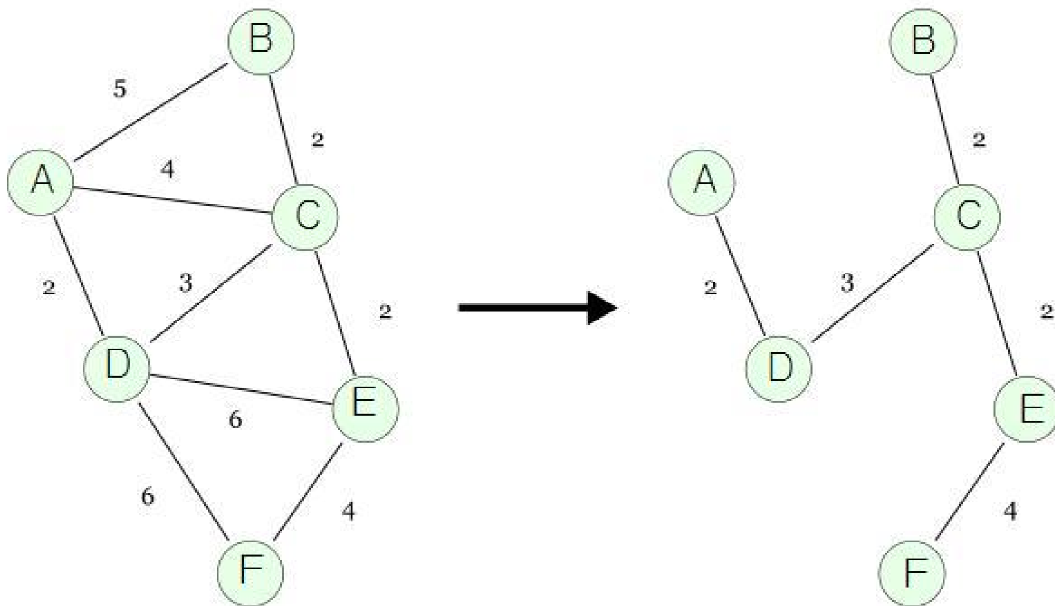

図 2　最小全域木問題の例

いて $O(\frac{n^2}{p} + \log^2 n)$ で最小全域木問題を解くことができる。

2.6.3　Sollin のアルゴリズム

本研究では、最小全域木問題を解く並列アルゴリズムとして、Sollin のアルゴリズムを用い、MPI 上でプログラム化した。以下に Sollin のアルゴリズムを示す。

Sollin のアルゴリズム

入力:　重み付無向グラフ G の隣接行列 W。W の各要素 $W_{x,y}$ $(0 \leq x, y < n)$ はプロセッサ P_x が保持する。

出力:　G の最小全域木 T の隣接行列 B。B の各要素 $B_{x,y}$ $(0 \leq x, y < n)$ はプロセッサ P_x が保持する。

step 1:　入力配列 W を作業用配列 W_0 にコピーし、$k = 0$ とする。

step 2:　隣接行列内に要素がある間、step2-1〜2-3 を繰り返す。

step 2-1:　制御用変数 k に 1 を加える

step 2-2:　各頂点 $v \in V_k$ において、v に隣接する辺 (v, u) $(u \in V_k)$ の中で一番重みの小さい辺 $\{(v, m) | w(v, m) \leq w(v, u)\ (u \in V_k)\}$ を探し、頂点 m を v の親 $p[v]$ として根付有向森を構成する。また、このとき辺 (v, m) および辺 (m, v) を作業用リスト L_k に加える。

step 2-3:　各頂点 $v \in V_k$ において、それぞれの根となる頂点 $r[v]$ を探す。

step 2-4:　各頂点 $v \in V_k$ において、v の根 $r[v]$ に v に接続する全ての辺 (v, u) $(u \in V_k)$ の重みおよび u の根 $r[u]$ データを集める。このとき各有向森の根に集められたデータを元に各有向森を 1 つの頂点とするグラフ G_{k+1} を構成する。

step 3:　作業用リスト L_i $(0 \leq i < k)$ から解行列 B を作成する。

以下に Sollin のアルゴリズムの計算量について述べる。

step 1: 定数個の代入のための定数時間である。

step 2-1: 定数個の足し算を行う定数時間である。

step 2-2: 大小比較を行ない、比較する辺を半分にするので $\log n$ 回繰り返すことで求められる。よって時間は $\frac{n^2}{\log^2 n}$ プロセッサで $O(\log n)$ となる。

step 2-3: ポインタジャンプすることにより根までの距離が半分になるので $\log n$ 回で求められる。よって時間は $\frac{n^2}{\log^2 n}$ プロセッサで $O \log n$ となる。

step 2-4: データを集めるには 2 つのデータを比較することによって行われるので、時間は $\frac{n^2}{\log^2 n}$ プロセッサで $O \log n$ となる。

step 3: 各辺において定数回の名前の書き換えを行うので、n^2 プロセッサを用いて $O(1)$ 時間となる。

step 2 の繰り返し回数は $O(\log n)$ 回であるので、Sollin のアルゴリズムは、p プロセッサ CREW PRAM 上で $O(\frac{n^2}{p} + \log^2 n)$ 時間で最小全域木問題を解くことができる。

2.7 最小全域木問題を解く MPI プログラム

本研究では、MPI の性能を評価するため、Sollin のアルゴリズムを元に MPI 上で最小全域木問題を解く並列プログラムを C++ 言語を用いて作成し、1 台の逐次計算による処理と、複数台による並列計算による処理とで、処理時間にどれほどの差が生まれるかを検証を行う。付録 A に本研究で作成した MPI プログラムを示す。

以下に本研究で作成した MPI プログラムについて述べる。

[最小全域木問題を解く MPI プログラム (計算機 k 台)]

入力: 無し。入力となる重み付無向グラフ G は、プログラム実行開始に生成される。

出力: G の最小全域木 T の隣接行列 $answer$。$answer$ はメイン PC に保持される。

step 0-1: メイン PC 上で入力となる重み付無向グラフ G を生成し、隣接行列 top として保持する。

step 0-2: top をメイン PC からサブ PC に送信する。

step 1: それぞれの頂点にプロセッサを割り振り、最小値を求めるメソッドである。メイン PC のデータをサブ PC へと送信し、各 PC が担当する頂点を (size[頂点数] × myrank[各々のプロセッサ番号]) / numprocs[PC 台数] 以上 (size[頂点] × (myrank[各々のプロセッサ番号]+1)) / numprocs[PC 台数] 未満より決める。各 PC は自身の担当する頂点の辺のうち最小のものを判定する。そして、メイン PC 以外の PC がメイン PC に求まった重みの最小の辺のデータを送信する。メイン PC はサブ PC から送信されたデータを受信し、各頂点の最小の辺のみを繋ぐ。

step 2: それぞれの頂点にプロセッサを割り振りポインタジャンプするメソッドである。更新された top をメイン PC からサブ PC へ送信し、2 つの頂点間で、互いの親 oya[SIZE] が互いであった場合頂点番号が小さい方の親を、自分自身に付け替える。サブ PC はデータをメイン PC に送信する。

step 3: 根に全てのデータを集めるメソッドである。親のデータが更新された場合、元の頂点を保存する。

本研究では、頂点数が 5、10、20、40、80、160 の 6 種類の重み付無向グラフに対し、MPI 上で 1〜5 台計算機を用いた場合の計算時間の測定を行なう。

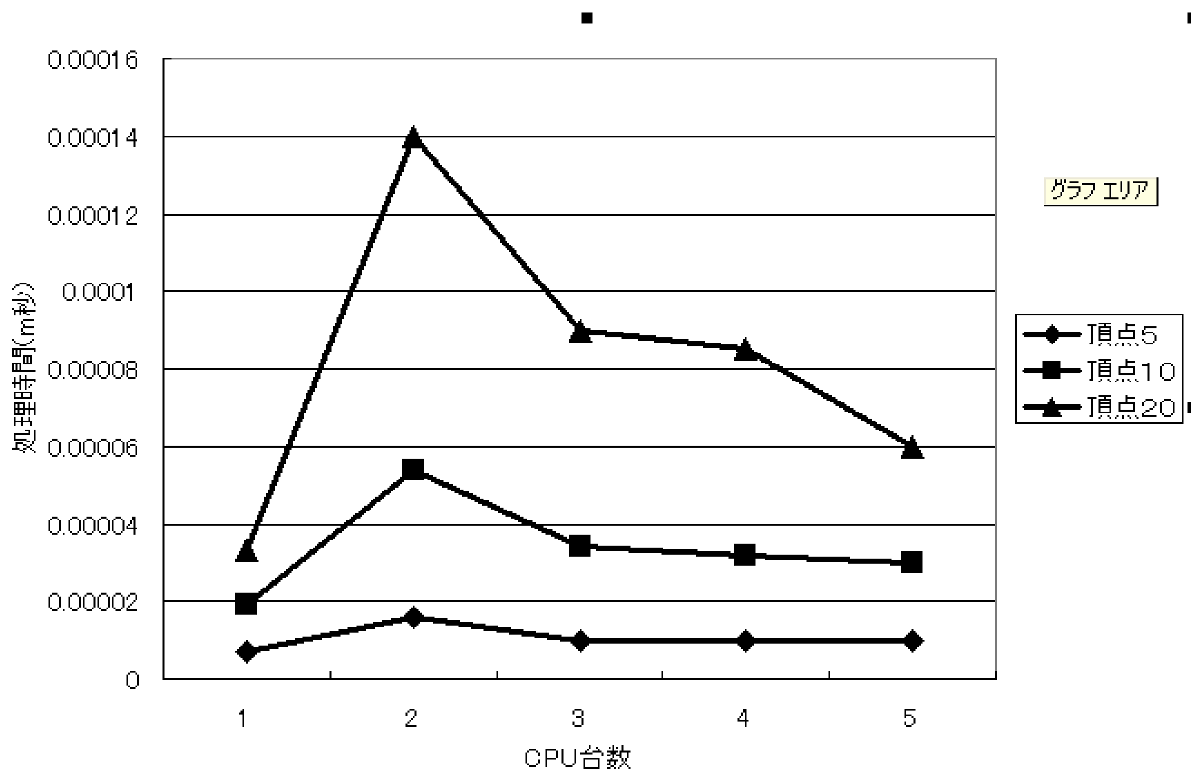

図 3　内部処理時間と計算機数の関係 (n=5,10,20)

3　計測結果

本研究では、MPI による並列化の有用性を検証するために MPI 上で最小全域木問題を解き、その時間を計測した。以下に本研究における計測結果を示す。

3.1　内部処理時間

頂点数がそれぞれ 5、10、20、40、80、160 の重み付無向グラフに対して、1〜5 台の計算機を用いて MPI 上で最小全域木問題を解いたときのメイン計算機の通信を含まない内部処理のみの時間を図 3、図 4、および表 2 に示す。

3.2　全体の処理時間

次に頂点数が 5、10、20、40、80、160 の重み付無向グラフに対して、1〜5 台の計算機を用いて MPI 上で最小全域木問題を解いたときの全体の処理時間を図 5 および表 3 に示す。

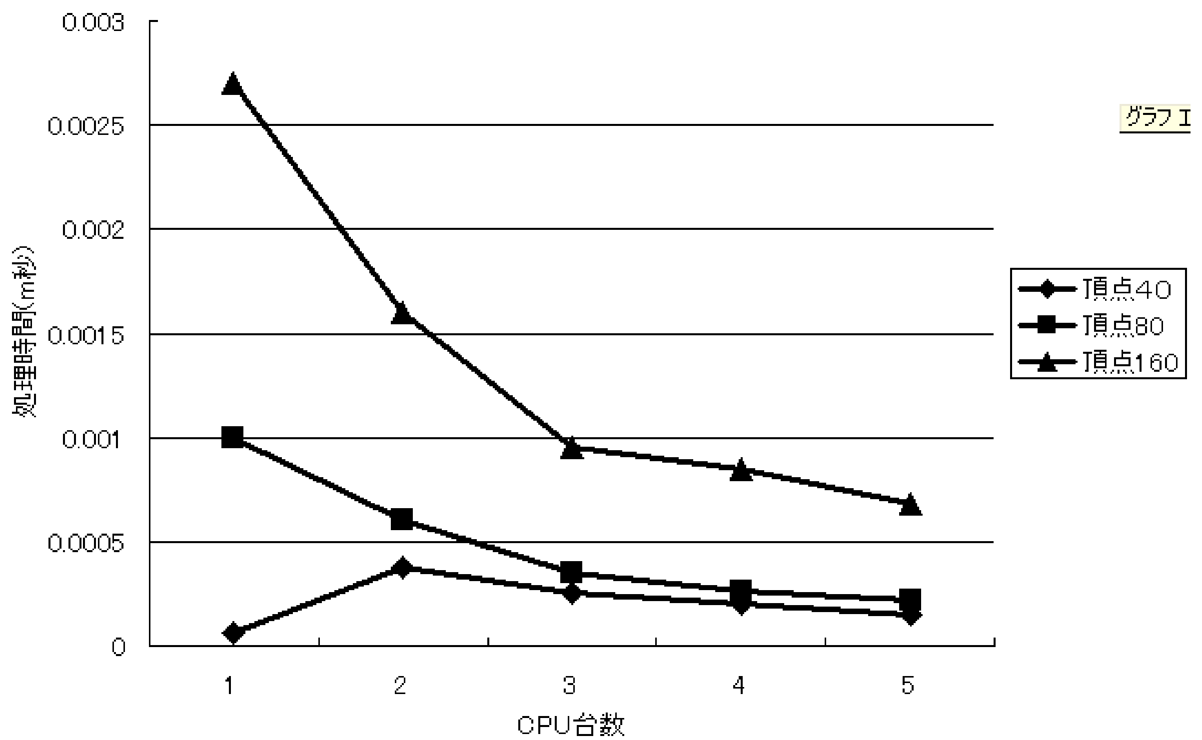

図 4　内部処理時間と計算機数の関係 (n=40,80,160)

表 2　内部処理時間 (m 秒)

	1 台	2 台	3 台	4 台	5 台
頂点 5	0.000007	0.000016	0.00001	0.00001	0.00001
頂点 10	0.000019	0.000054	0.000034	0.000032	0.00003
頂点 20	0.000033	0.00014	0.00009	0.000085	0.00006
頂点 40	0.000065	0.000038	0.00025	0.0002	0.00015
頂点 80	0.0001	0.0006	0.00035	0.00026	0.00022
頂点 160	0.0027	0.0016	0.00095	0.00085	0.00068

3.3　理論値との比較

本節では、MPI プログラムの処理時間の計測結果と理論値との比較を行う。Sollin のアルゴリズムの計算量は $O(\frac{n^2}{p} + \log^2 n)$ であるので、

$$T_{comp}(n,p) = \frac{an^2 + bn + c}{p} + d\log^2 n + e\log n + f \quad (1)$$

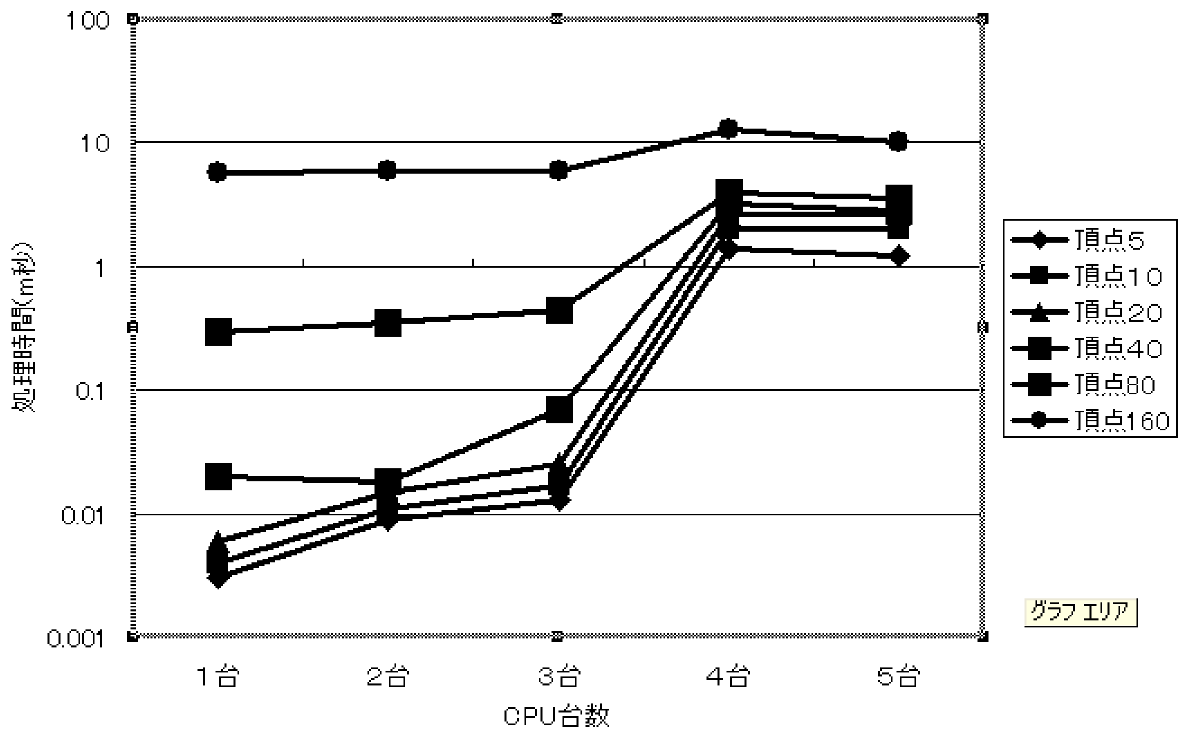

図 5　全体の処理時間と計算機数の関係

表 3　全体の処理時間 (m 秒)

	1 台	2 台	3 台	4 台	5 台
頂点 5	0.003	0.009	0.013	1.4	1.2
頂点 10	0.004	0.011	0.017	2	2
頂点 20	0.006	0.015	0.025	2.6	2.6
頂点 40	0.02	0.018	0.07	3.2	2.8
頂点 80	0.3	0.35	0.45	4	3.6
頂点 160	5.7	6	6	13	10

と置き、表 2 の値から連立方程式を立てる。この連立方程式より、

$$T_{comp}(n,p) = \frac{0.66n^2 \times 10^{-6} - 86n \times 10^{-6} + 2.2 \times 10^{-3}}{p} + 1.74 \log^2 n \times 10^{-6} + 2.11 \log n \times 10^{-6} - 80 \times 10^{-6} \qquad (2)$$

が得られる。

4　考察

図 3、図 4、および表 2 より、メイン PC の内部処理時間自体は短縮できていることが示される。これは計算機数を増加するごとに、各計算機の処理内容が減少するからだと推測される。また、グラフのサイズに着目した場合、頂点数の大きいグラフほど計算機数の増加による内部計算時間の短縮効果が得られていることが示される。つまり、容量の大きいデータほど、並列計算が有効だと示される。

次に、最小全域木問題を解く処理時間のうち、全体の処理時間は、図 5 および表 3 より、計算機数の増加に伴い全体の処理時間が増加することが示される。処理時間が増加したのは、計算機間の通信や同期にかかる時間が全体の処理時間の大きな割合を占めていたからだと考えられる。また、図 5 および 3 において、計算機数を 3 台から 4 台に増やしたときに急に全体の処理時間が増加しているのは図 1 で示されているように、4 台目の計算機の RAM の数値が他の計算機よりも小さいことや、計算機自体の空き容量が少ないことによるものだと考えられる。上記の仮説を検証するために計算機を繋ぐ順序を変更し処理時間を計測したところ、計算機の台数に関係なく、遅延の原因だと思われる計算機を含めたときに処理時間の大きな増加が見られた。よって上記の仮説は妥当であることが示される。

5 結論・今後の課題

本研究では、MPI による並列化の有用性を検証するために MPI 上で最小全域木問題を解き、その時間を計測した。頂点数の多いグラフに対しては MPI を用いて並列化を行なうことにより、内部処理時間の短縮が得られる。一方全体の処理時間は、計算機台数を増やすごとに増加することが示され、MPI による並列化が必ずしも効率的だとは言えないことが示される。これは求められる処理や接続されるネットワークの速度により、使用されるメインメモリのバイト数も変化することが考えられるので、様々な処理においての検証が必要であり、今回の検証結果だけでは不十分である。

また、本研究で使用したプログラムでは接続される計算機の性能を自動的に出力し、割りだした性能によりデータ分割割合を変動させることが出来ていない。本来なら不特定多数の計算機と接続して仮想並列計算機とすることもあるので、全てをプログラムにより処理できなければ実用性に欠ける。そして、プログラムの問題が解決したとしても根本的な部分で重大な問題が残る。それは処理工程において、接続される全ての計算機と同期し、計算機の性能を出力する工程と、その出力結果に依存してデータを分割する工程が、通常の方法よりも余分にオーバヘッドとして存在することである。これにより、この方法は膨大な処理かつ計算機の性能差が大きい場合は有効だと考えられるが、それ以外の場合ではたとえ最適な分割割合が求められても結果として処理時間が増加してしまうことが考えられる。したがって、今後の課題としてはオーバヘッドを無くすため、仮想並列計算機に使用される計算機の性能を処理実行時に求めるのではなく、接続した段階で計算機の性能を把握できるシステムを構築することが考えられる。よって MPI を使用する際には、機能性の高い計算機や低い計算機、ネットワークの性能やその他の計算機環境、BSP[3] や CGM[5] などの上で通信を考慮し、また、クラスタ環境に対応したプログラミングをする必要がある。

謝辞

本研究を完成するにあたって、夜遅くまでご指導していただいた石水助教、また同研究室の皆さんには様々な助言をしていただき感謝の言葉を述べたいと思います。

参考文献

[1] J.JáJá 著, An Introduction to Parallel Algorithms-, Addison-Wesley Professional (1992).

[2] P.Pacheco 著, 秋葉博訳,MPI 並列プログラム, 培風館 (2001).

[3] L.G.Valiant, A Bridging Model for Parallel Computation, Comm. of the ACM, Vol.33, No.8, pp.103–111, (1990).

[4] A.Baumker,W.Dittrich,F.M.Heide and I.Rieping,”Realistic Parallel Algorithms:Priorit Queue Operations and selection for BSP Model,”Proc.EuroPar’96,VOl.II.pp.369-376,1996.

[5] F.Dehne, A.Fabri and A.Rau-Chaplin, Scalable Parallel Computational Geometry for Corse Grained Multicomputers, Proceeding of ACM Symposium on Computational Geometry, pp.298–307 (1993).

[6] 牛島省 (著):OpenMP による並列プログラミングと数値計算法、丸善株式会社 (2006)

[7] 畑崎隆雄 (訳)、ラジープ・タークル、ユーイング・ラスク、ウイリアム・グロップ (著)：実践 MPI-2. ピアソン・エデュケーソン (2002)

[8] MPICH2(URL): http://www.mcs.anl.gov/research/projects/mpich2/downloads/index.php?s=downloads

[9] MPICHonWindowsLocal(URL): http://ums.futene.net/wiki/Paralell/4D5049434832206F6E646F7773204C6F63616C.

[10] Visual Studio 2005.(URL): http://www.microsoft.com/japan/msdn/vstudio/

[11] 最小全域化 (URL): http://www.deqnotes.net/acmicpc/prim/

[12] MPI ライブラリの調査と性能の計測 (URL): http://mikilab.doshisha.ac.jp/dia/research/report/2004/0613/001/report

[13] MPI 覚え書き [YASUAKI ITO’s HomePagel](URL): http://www.ornl.gov/

[14] Al Geist et al.,”PVM:Parallel Virtual Machine,”MIT Press1994

[15] OpenMosix(URL): http://theochem.chem.nagoya-u.ac.jp/wiki/wiki.cgi/ClusterBuild?page=OpenMosix

[16] SCORE(URL): http://ja.losts.net/SCORE

[17] http://www.cs.hiroshima-u.ac.jp/ yasuaki/dokuwiki/doku.php?id=mpi:openmpi

付録 A　最小全域木問題を解く MPI プログラム

以下に、本研究で作成した最小全域木問題を解く MPI プログラムを示す。

mpich2.cpp

```
#include "mpi.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 160 //頂点数
int oya[SIZE]; //それぞれの頂点の親
int alive[SIZE];//頂点が生きているかの判定
int trank[SIZE];//それぞれの頂点にどの PC が担当しているかの配列
int answer[SIZE][SIZE];//答えようの配列
int change_x[SIZE][SIZE];// 子を殺した際のデータを保存
int change_y[SIZE][SIZE];
int  top[SIZE][SIZE];//  辺の重みの配列
int myrank,numprocs,zerop;
double st1=0,st2=0;
double St1start,St1finish,St2start,St2finish,start,finish; //時間計測用
MPI_Status status;

int size = sizeof oya/sizeof oya[0];

int log(int n){//ループ回数計測用
    int i = 0;
    while(n > 1){
        n /= 2;
        i++;
    }
    return i;
}


int Hen(int x){ //グラフの辺の数計測
    int i = 0;
```

```
    x--;
    while(x > 0){
        i += x;
        x--;
    }
    return i;
}


void PStep1(){//それぞれの頂点にプロセッサを割り振り、最小値を求める
    int ans[SIZE][SIZE];
    MPI_Bcast(oya,size,MPI_INT,0,MPI_COMM_WORLD);// メイン PC のデータをサブ PC へと送る
    MPI_Bcast(alive,size,MPI_INT,0,MPI_COMM_WORLD);
    MPI_Bcast(top,size*size,MPI_INT,0,MPI_COMM_WORLD);
    MPI_Bcast(change_x,size*size,MPI_INT,0,MPI_COMM_WORLD);
    MPI_Bcast(change_y,size*size,MPI_INT,0,MPI_COMM_WORLD);
    MPI_Bcast(answer,size*size,MPI_INT,0,MPI_COMM_WORLD);
    St1start = MPI_Wtime();
    for(int tantou = (size*myrank)/numprocs ; tantou < (size*(myrank+1))/numprocs ;tantou++){
        int min = 999998;
        bool hantei = false; //最小値が変わったかどうかを判定する変数
        int box = 0;//最小値の場所を保存
        if(alive[tantou] == 1 && tantou < size){
            for(int j = 0;j < size;j++){//最小値を求めるループ
                if(alive[j] == 1 && min > top[tantou][j]){
                    min = top[tantou][j];
                    box = j;
                    hantei = true;
                }
            }
            if(hantei){
                answer[change_x[change_x[tantou][box]][change_y[tantou][box]]][change_y[change_x[tan
付け替え前の頂点を探し、その場所を１増やす
                oya[tantou]= box;
            }
        }
        if(myrank != 0)MPI_Send(&oya[tantou],1,MPI_INT,0,tantou,MPI_COMM_WORLD);//メ イ
ン PC 以外の PC がメイン PC にデータを送る

    }
```

```
    St1finish = MPI_Wtime();
    st1 += (St1finish - St1start);
    if(myrank == 0){ //サブ PC のデータをメイン PC が受け取る
        for(int i = zerop;i < size;i++){
                MPI_Recv(&oya[i],1,MPI_INT,trank[i],i,MPI_COMM_WORLD,&status);
        }
    }
    MPI_Reduce(answer, ans, size*size, MPI_INT, MPI_LOR, 0,MPI_COMM_WORLD);//全 体 の
answer の論理和を取る
    if(myrank==0){
        for(int i = 0;i < size; i++){
            for(int j = 0;j < size ; j++){
                answer[i][j] = ans[i][j];
            }
        }
    }

}

void PPointJump(){//それぞれの頂点にプロセッサを割り振りポインタジャンプする
    /*
    まず親の親が自分自身と見合っている所を、頂点番号が小さい方を
    親を自分自身に付け替える
    */
    MPI_Bcast(oya,size,MPI_INT,0,MPI_COMM_WORLD);
    for(int tantou = (size*myrank)/numprocs ; tantou < (size*(myrank+1))/numprocs ;tantou++){
        trank[tantou] = myrank;
        if(tantou == oya[oya[tantou]] && tantou < oya[tantou]) oya[tantou] = tantou;
        if(myrank!=0)MPI_Send(&oya[tantou],1,MPI_INT,0,tantou,MPI_COMM_WORLD);
    }

    if(myrank == 0){
        for(int i=zerop;i < size;i++){
            MPI_Recv(&oya[i],1,MPI_INT,trank[i],i,MPI_COMM_WORLD,&status);
        }
    }
    /*それぞれのデータを集め、もう一度送りなおす*/
    MPI_Bcast(oya,size,MPI_INT,0,MPI_COMM_WORLD);
    for(int tantou = (size*myrank)/numprocs ; tantou < (size*(myrank+1))/numprocs ;tantou++){
```

```
        St2start = MPI_Wtime();
        for(int j=0;j <log(size);j++){//ポインタジャンプ
            oya[tantou]=oya[oya[tantou]];
        }
        St2finish = MPI_Wtime();
        st2 += (St2finish - St2start);
        if(myrank!=0)MPI_Send(&oya[tantou],1,MPI_INT,0,tantou,MPI_COMM_WORLD);
    }
    if(myrank == 0){
        for(int i=zerop;i < size;i++){
            MPI_Recv(&oya[i],1,MPI_INT,trank[i],i,MPI_COMM_WORLD,&status);
        }
    }

}


void Step3(){//根にすべてのデータを集めるメソッド
    for(int i = 0;i < size;i++){
            if(i == oya[oya[i]]){ //親の親が自分自身の場合
                for(int j= 0;j <size;j++){
                    if(i == oya[j] && i != j){
                        top[i][j] = 999999;//すでに使った辺を消す
                        top[j][i] = 999999;
                        for(int count = 0;count < size;count++){//親のデータが更新された
場合に元の頂点を保存
                            if(top[i][count] > top[j][count] && top[i][count] != 999999){
                                top[i][count] = top[j][count];
                                top[count][i] = top[count][j];
                                change_x[i][count] = j;
                                change_y[count][i] = j;
                            }

                        }
                        for(int count =0;count < size;count++){ //親のデータが更新された
場合に元の頂点を保存
                            if(top[i][oya[count]] > top[i][count] && top[i][oya[count]] != 999999){
                                top[i][oya[count]] = top[i][count];
                                top[oya[count]][i] = top[count][i];
                                change_y[i][oya[count]] = count;
```

```
                                change_x[oya[count]][i] = count;
                            }
                        }
                    }

                }
            }else {
                alive[i] = 0; //ルートでない頂点を殺す
            }
        }
}



bool hantei(int x){ //同じ重みの辺が無いかを判定
    for(int i = 0; i < size ;i++){
        for(int j = i ; j < size ;j++){
            if(top[i][j] == x){
                return true;
            }
        }
    }
    return false;
}

void makeGraph(){ //グラフ作成部分
    for(int i = 0;i < SIZE;i++){
        alive[i] = 1;
        oya[i] = (SIZE+i+1);
        for(int j = 0;j < SIZE ;j++){ //初期化
            answer[i][j] = 0;
            top[i][j] = 0;
            change_x[i][j] = i;
            change_y[i][j] = j;
        }
    }
    int y;
    for(int x = 0 ; x < numprocs;x++){//どの PC がどの頂点を担当するのかを保存
        for(y = (size*x)/numprocs ; y < (size*(x+1))/numprocs ;y++){
            trank[y] = x;
```

```
        }
        if(x == 0)zerop = y;
    }
    srand(time(NULL));//乱数の種を作成
    for(int i = 0; i < size ;i++){
        for(int j = i ; j < size ;j++){
            if(i == j){
                top[i][j] = 999999;
            }else {
                int x = rand() % (Hen(size)+1) ;
                while(hantei(x)){
                    x = rand() % (Hen(size)+1);
                }
                top[i][j] = x;
                top[j][i] = x;
            }
        }
    }

    /*for(int i = 0; i < size ;i++){
        for(int j = 0 ; j < size ;j++){
            printf("%3d ",top[i][j]);
        }
        printf("\n");
    }*/

}


int main(int argc,char **argv)
{
    MPI::Init(argc,argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);//参加しているプロセス数を計測
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);//自身のプロセス番号を所得
    start = MPI_Wtime();

    if(myrank == 0) makeGraph();

    for(int i = 0;log(size) > i ;i++){ //頂点数 n の場合 logN 回ループする
        PStep1();
```

```
        PPointJump();
        if(myrank == 0)Step3();

    }
    printf("rank%d Step1 処理時間：%10.6f seconds\n",myrank,st1);
    printf("rank%d Step2 処理時間：%10.6f seconds\n",myrank,st2);
    if(myrank ==0){
    /*  for(int i = 0; i < size ;i++){
            for(int j = 0 ; j < size ;j++){
                printf("%2d ",answer[i][j]);
            }
            printf("\n");
        }*/
    finish = MPI_Wtime();
    printf("処理時間：%10.6f seconds\n",finish - start);
    }


    MPI::Finalize();



}
```