卒業研究報告書

題 目

並列アルゴリズムについての調査報告

指 導 教 員

石 水　隆 助教

報 告 者

07–1–037–0251

太 田　尚 吾

近畿大学理工学部情報学科

平成 23 年 1 月 28 日提出

概要

近年では、大容量の記憶デバイスの登場や、ネットワークの高速化により、大量のデータを高速に処理することが求められている。高速な処理を行うためには、複数のプロセッサを持つ並列計算機 (Parallel Computer) が用いられている。しかし、複数のプロセッサを持つ並列計算機は高価であり、容易に用いることができない。そこで、一つの処理にネットワーク上で接続した複数の計算機で対応する、仮想並列計算機 (Parallel Virtual Computing) というシステムが考えられる。仮想並列計算機を用いることで、コストを抑えて処理速度をあげることが出来る。

本研究では、無料で提供されている並列計算が可能になるソフトウェアの一つである MPI(Message Passing Interface)[1] を用いてグラフ問題の一つである最大全域木問題を解き、その有用性を検証する。

目次

1 序論

1.1 本研究の背景

1.1.1 **並列処理** (Parallel Processing)

ある 1 つの処理を、複数のプロセッサを用いて処理を行い、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。並列処理はタンパク質の構造解析や天体の軌道観測、地球の大気の循環など大規模なシミュレーションなど、逐次処理では膨大な時間がかかる場所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 **仮想並列計算** (Parallel Virtual Computing)

前節で述べた通り、近年並列計算の重要性は高まっており、高性能な並列計算機が求められてきている。しかし、複数のプロセッサを持つ並列計算機は高価であり、容易に用いることができない。そこで、一つの処理にネットワーク上で接続した複数の計算機で対応する、仮想並列計算機 (Parallel Virtual Computing) というシステムが注目されている。仮想並列計算機を構築するソフトウェアは様々なものが開発されており、無料で提供されているものもある。このため、仮想並列計算機を個人で使用することも容易になっており、今後並列計算機の重要性はより拡大していくと考えられる。無料で提供されている仮想並列計算機を構築するソフトウェアとしては、MPI(Message Passing Interface)[1] , PVM(Parallel Virtual Machine)[5], OpenMP[6], OpenMosix[8], SCore[9] 等がある。以下に、これらについての説明をする。

1.2 仮想並列計算機を構築するソフトウェア

MPI(Message Passing Interface) は、メモリ型の並列計算をサポートするためのライブラリである。MPI は 1992 年に結成された MPI Forum によって、標準仕様の定義や検討を作り始めたことで具体化してきた。MPI の開発には、アメリカやヨーロッパの 60 人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。MPI は標準を目指して作成されたため、様々な通信関数が実装されている。そのため、MPI を用いて作成したプログラムは移植性が高く、MPI を使用するユーザは通信を考慮せずにプログラムを組むことが出来る。MPI は異機種間での通信は考慮されておらず、並列計算機を構築する際は、オペレーティングシステムも統一しなければならない。

PVM(Parallel Virtual Machine)[5] は、1991 年にアメリカのオークリッジ国立研究所を中心に異機種間の分散処理が目的に開発された、メッセージパッシングによる並列処理を行なうための並列化ライブラリである。PVM はワークステーションクラスタなため、TCP/IP の通信ライブラリで一般的に使用されている LAN 環境があれば並列処理が実行出来るので多くのユーザが利用している。また、異機種間の通信も考慮されているため、対応する計算機は家庭にあるパーソナルコンピュータからスーパーコンピュータなど多くの種類で PVM による並列処理が出来る。PVM の構成は 2 つに大きく分けられる。1 つはデーモン (pdmd3) であり、PVM によって構成された仮想並列計算機上にある全ての計算機にデーモンが存在する。PVM はこのデーモンを使用し通信を行なっている。複数のユーザは互いにオーバーラップさせ仮想並列計算機を構成することが出来る、また、各ユーザは PVM アプリケーションを 1 人で複数実行することが可能となっている。もう 1 つは、PVM インターフェイスルーチンライブラリである、ライブラリには、メッセージパッシング、プロセスの生成、タスクの協調、仮想計算機の構成ルーチンを提供している。

OpenMP[6] は、並列計算環境を利用するために用いられる標準化された基盤である。OpenMP は主に共有メモリ型並列計算機で用いられる。MPI では明示的にメッセージの交換をプログラム中に記述しなければならないが、OpenMP は 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。MPI との比較では、OpenMP は異なるスレッドが同一のデータを同じアドレスで参照できるのに対して、MPI では明示的にメッセージ交換を行わなければならない。そのため SMP 環境においては大きなデータの移動を行なわずにすむので高い効率が期待できる。ただし並列化の効率はコンパイラに依存するのでチューニングによる性能改善が MPI ほど高くならないという問題がある。また、OpenMP は MPI に比べてメモリアクセスのローカリティが低くなる傾向があるので、頻繁なメモリアクセスがあるプログラムでは、MPI の方が高速な場合が多い。

OpenMosix[8] は、Linux のプロセスをネットワーク経由でほかのクラスタノードに実行させる migration と呼ぶ仕組みである。動的な負荷分散を自動的に行うので、複数の計算機を一台の並列計算機として利用し、最大限に性能を引き出すことができる。

SCore[9] は、経済産業省が設立した超並列処理研究推進委員会である新情報処理開発機構で開発された Linux 用クラスター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 仕様に基づいたライブラリ群や補助ツール群を動作させる基盤のことで、当初は UNIX をベースに設計されていた。OpenMP や MPI、MPC++ といった並列プログラミング環境をサポートし、対話型実行環境、ギャングスケジューリング (管理者権限で、他のプログラムの実行を停止し、独占的に特定プログラムを実行させる事ができるスケジューリング機能) を含むマルチスケジューリング、マルチユーザ環境を持つ。

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。

1.3 MPICH

本研究では MPI の実装として幅広く用いられている MPICH2[2] を用いる。MPICH はアメリカのオーゴン国立研究所 [**?**] が開発を行い、無償でソースコードを配布したライブラリであり、rsh/ssh 越しに通信を行うものである。移植のしやすさを重視した作りになっているため、プログラムのソースコードを変更することなく、分散メモリ環境、共有メモリ環境の計算機で動作させることが可能である。本研究では、MPICH の最新版である、MPICH2[2] を用いて検証を行う。

1.4 本研究の目的

1 台の計算機を用いて処理した場合と比べて、MPI を用いて複数の計算機で仮想並列処理した場合にどの程度処理時間を短縮できるかを計測し、MPI による仮想並列計算での処理の有用性を検証するのが本研究の目的である。

1.5 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 の閉路を含まない連結な部分グラフにの中で、辺の重み

の総和が最小のものを求める問題である。m は入力グラフの辺の本数、n は入力グラフの頂点の個数とすると最小全域木問題に対して、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 [3]。また、Sollin のアルゴリズムからは、CREW PRAM 上で、プロセッサ数を p とすると $O(\frac{n^2}{p} + \log^2 n)$ 時間で解く並列アルゴリズムアルゴリズムが得られる [3]。

1.6 本報告書の構成

本報告書の構成を以下に述べる。2 章に本研究での使用機器、実験環境について述べる。3 章に研究の内容、4 章に結果と考察、5 章に結論を述べる。

2 準備

2.1 MPI (Message Passing Interface)

MPI (Message Passing Interface)[1] 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た、分散メモリ型並列計算機における、メッセージパッシングをパラダイムとする並列プログラミングのための標準的な API(Application Programming Interface) を提供するライブラリである。かつては個々の並列計算機が独自のメッセージパッシングライブラリを持っていたが、1990 年代前半に PVM(Parallel Virtual Machine)[5]、そして、その後に MPI へと移行した。現在は MPI がほぼ標準となっており、ほとんどの分散メモリ型並列計算機に実装されている。MPI を用いることで、汎用性の高い並列プログラミングが可能になる。無料で提供されてい MPI の主な実装として、MPICH2[2] や LAM[10]、OpenMPI[11] 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[2] で開発された。2005 年には MPICH の後継として MPICH2 が開発され、現在では主流になっている。

LAM[10] はネットワーク接続された計算機のための並列処理環境および開発システムである。LAM は、拡張モニタリングおよびデバッグツールによってサポートされた MPI プログラミング標準である。1 つのデーモンとして各々のコンピュータで動き、ナノカーネルおよび人手でスレッド化された仮想プロセス群によって構成される。ナノカーネルは、簡単なメッセージ通信およびローカルプロセスに対する待ち合わせサービスを提供する。いくつかのデーモン中のプロセスが、ネットワーク通信サブシステムを形成し、それは他のマシン上の LAM デーモンとメッセージの送受信を行う。そのネットワーク・サブシステムは、基本同期機構に加えてパケット化およびバッファリングなどの特徴を追加する。ユーザは必要に応じてサービスの追加、削除ができる。

OpenMPI[11] は Open MPI Team が開発している高性能メッセージパッシングライブラリーである。コミュニティー、研究機関、パートナー企業によって開発され、維持されているオープンソース MPI-2 を実装している。特徴としては、完全な MPI-2 規格準拠、スレッドセーフと並行性、ダイナミックなプロセスのスワッピング、ネットワークと耐障害性の処理、異種ネットワークのサポート、シングルライブラリのサポート、ランタイムとしての役割、多数のジョブスケジューラーのサポート、多数の OS のサポート、アクティブなメーリングリスト、BSD ライセンスに基づくオープンソースライセンスなどがある。

本研究では、MPI を実装するソフトウェアとして、現在最も幅広く使用されている MPICH2 を用いる。

2.2 使用機器

本研究では、MPI による仮想並列環境の構築に性能の等しい 4 台の計算機を使用する。1 台をホストコンピュータ、3 台をサブコンピュータとして扱う。本研究で使用した計算機のスペックを表 1 に記す。本研究では表 1 に示す計算機を 100Base-TX による LAN を用いて接続し MPI 環境の構築を行った。また、本研究で使用する計算機は、OS として Windows 系 OS を使用する。Windows 系 OS を使用するのは C++ の環境で使い易いからである。

本研究では表 1 のスペックをもつ計算機計 4 台での検証を行なう。また、仮想並列計算機を構築する際に LAN やルータ、ハブを使用し計算機をネットワークでつないでいる。この構成図を図 1 に示す。

表 1　使用した計算機のスペック

	OS	メモリ	CPU
ホストコンピュータ	Windows Vista	1.00GB	Intel Core 2 Duo 1.40GHz
サブコンピュータ 1	Windows Vista	1.00GB	Intel Core 2 Duo 1.40GHz
サブコンピュータ 2	Windows Vista	1.00GB	Intel Core 2 Duo 1.40GHz
サブコンピュータ 3	Windows Vista	1.00GB	Intel Core 2 Duo 1.40GHz

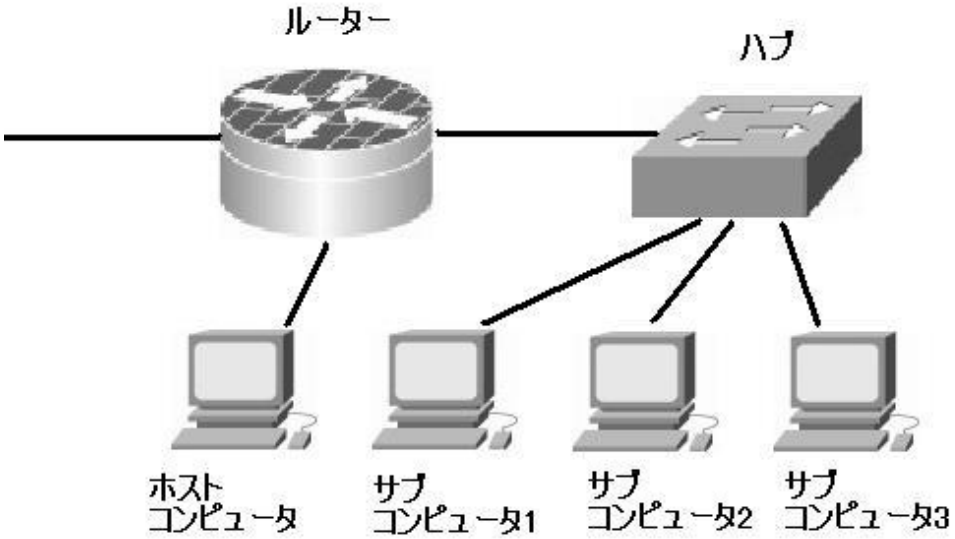

図 1　並列仮想計算機の構成

2.3　MPICH2 のインストール

MPICH2 を使用するために、各計算機に MPICH2[2] のインストールを行う。MPICH2 の公式ページから本研究では、2010 年 12 月現在の最新のヴァージョンである、mpich2-1.3.1-win-ia32.msi をダウンロードした。このファイルを解凍すると、実行形式のインストーラファイルとなる。インストーラを起動し、インストール先を C:Program Files \MPICH2 \ に指定する。そして MPICH2 の実行ファイルのあるフォルダに対して、環境変数 PATH を指定する。環境変数の設定は、マイコンピュータを右クリックメニューから [プロパティ] を選択し、[詳細設定] のタブで [環境変数] のボタンを左クリックシステム環境変数のリストから、変数名の path を選択し、編集ボタンをクリックすることにより行える。変数値の最後に;C:Program Files \MPICH2 \bin \ を追加する。コマンドプロンプトを立ち上げ、mpiexec 命令を実行したとき、引数の入力

を促す Usage メッセージが表示されるのを確認することで、MPICH2 をインストール出来たかどうかを確認することが出来る。OS が Windows の場合には、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設定しておく必要がある。並列環境の作成のために、それぞれの計算機に mpi アカウントを作り、パスワードを ipm に設定しておく。また、プロセスの実行の際には、全ての計算機に実行ファイルを置く必要があるため、mpi 共有フォルダをそれぞれの計算機に準備する。mpi というフォルダを作り、右クリックし、共有をクリックする。

2.4 Visual C++ のインストール

本研究で作成する MPI プログラムは C++ 言語を用いる。本研究では、C++ 言語をコンパイルできる環境を作るために、Visual C++ のインストールを行う。VisualC++2008 Express Eddition をマイクロソフト [4] の公式ページからダウンロードし、インストールを行う。インストーラーを起動し、指示にしたがっていく。これを用いて MPICH2 による並列プログラミングを行うためには、VisualC++2008 のツールオプションから、MPICH2 のインクルードファイルとライブラリファイルのあるフォルダを指定して追加する必要がある。C:Program Files\MPICH2\include、C:Program Files\MPICH2\lib のフォルダを選択する。

3 研究内容

本研究では計算機 4 台を用いて MPI 環境を構築する。MPI の有用性を検証するための問題として、最小全域木問題を用いる。

3.1 最小全域木問題

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 の閉路を含まない連結な部分グラフにの中で、辺の重みの総和が最小のものを求める問題である。

ここで重み付グラフとは、各辺 $k = (u, v)$ に重み $w(k) = w(u, v)$ が与えられているグラフである。また、無向グラフとは辺が相異なる頂点の非順序対 (v, w) で与えられるグラフである。閉路とは、始点と終点が同じ路のことであり、連結とは、任意の 2 頂点の間に辺が存在することである。この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 [3] 等がある。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 のアルゴリズムは $O(m + n \log n)$、Kruskal のアルゴリズムは $O(m \log m)$、Sollin のアルゴリズムは $O(n^2)$ 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。

本研究では、頂点数が 10,20,30,40,80,160 の重み付無向グラフに対し、その最小全域木をそれぞれ 1,2,3,4 台の計算機で計算し、その実行時間を測定する。本研究で作成した最小全域木問題の計算するプログラムは Sollin[3] のアルゴリズムを元にしている。

3.2 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。Sollin のアルゴリズムを以下に示す。

[Sollin のアルゴリズム]

入力: 重み付無向グラフ G の隣接行列 A。A の各要素 $A_{x,y}$ $(0 \leq x, y < n)$ 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: 行列の要素が空になるまで 2-1 から 2-4 を繰り返す。

step 2-1: k に 1 を加える。

step 2-2: 頂点 $v \in V_k$ において、v に隣接する辺の中最も小さい辺 (v, m) を探し、頂点 m を v の親 $p[v]$ として木を構成する。また、このとき辺 (v, m),(m, v) を作業用配列 L_k に加える。

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

step 2-4: 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。各木の根に集められたデータをから、各木を 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。

[Sollin のアルゴリズムの計算量]

step 1: コピーするだけなので定数の計算量である。

step 2-1: 定数個の加算なので定数の計算量である。

step 2-2: 大小比較していき、比較する辺を半分にしていくので $\log n$ 回繰り返すことになる。よって、計算量は $\frac{n^2}{\log^2 n}$ プロセッサで $O(logn)$ となる。

step 2-3: ポインタジャンプで処理が半減していくので、計算量は $\frac{n^2}{\log^2 n}$ プロセッサで $O(logn)$ となる。

step 2-4: データを集めるためには 2 つのデータを比較するので、計算量は $\frac{n^2}{\log^2 n}$ プロセッサで $O(logn)$ となる。

step 3: 定数回の書き換えを行うので、計算量は 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。

3.3 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生成される。

出力: G の最小全域木 T の隣接行列 $answer$。$answer$ はホスト PC に保持される。

step 0-1: ホスト 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 $hairetu$ として保持する。

step 0-2: $hairetu$ と頂点に関するデータをホスト PC からサブ PC に送信する。

step 1: ホスト PC から送られたデータから自分が処理する頂点を決め、重みが最も小さい辺を選びホスト PC に辺のデータを送信する。

step 2: $hairetu$ を更新し、サブ PC に送信する。辺に隣接する頂点番号の小さい方を親にし、ホスト PC にデータを送信する。

step 3: $hairetu$ を更新し、重みが最も小さい辺を繋ぐ。

step 4: 全ての頂点が処理されるまで繰り返し、グラフと実行時間を表示させる。

本研究で作成した MPI プログラムの実行は VisualC++ でコンパイルし、コマンドプロンプトから mpiexec sollin.cpp と打ち込むことで実行することができる。

表 2　内部計算時間と計算機数の関係

頂点数 \ 台数	1	2	3	4
10	0.000010	0.000008	0.000006	0.000005
20	0.000017	0.000012	0.000009	0.000009
30	0.000027	0.000022	0.000015	0.000014
40	0.000038	0.000031	0.000024	0.000021
80	0.00095	0.0007	0.00034	0.00025
160	0.00258	0.0018	0.0009	0.00025

(m 秒)

表 3　全体の処理時間と計算機数の関係

頂点数 \ 台数	1	2	3	4
10	0.0045	0.012	0.016	1.3
20	0.006	0.013	0.023	2.0
30	0.012	0.018	0.0055	3.0
40	0.023	0.039	0.064	3.2
80	0.26	0.31	0.4	4.1
160	5.6	6.2	6.2	12

(m 秒)

4　結果・考察

表 2 に頂点数 10,20,30,40,80,160 の重み付無向グラフに対して、その全域木をそれぞれ 1,2,3,4 台の計算機を用いて求めた場合のプログラムの内部計算にかかった時間を示し、表 3 に各計算機間の通信時間を含めた全体の処理時間を示す。

表 2 より、計算機の台数を増やすことにより、内部計算の時間は短縮されることが示されたが、表 3 により、プログラムの全体の処理時間は計算機の台数が増える程、処理にかかる時間は長くなっていることが示される。これは、各計算機間でのデータの通信に時間がかかっているためだと考えられる。

次に、内部計算時間の計測結果と理論値を比較する。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、$T_{comp}(n,p) = \frac{an^2+bn+c}{p} + d\log^2 n + e\log n + f$ と置き、表 2 の値から連立方程式を立てる。

$$T_{comp}(n,p) = \frac{2.83*10^{-7}n^2 + 11*10^{-4}n + 1.2*10^{-4}}{p} + 2.12*10^{-4}\log^2 n + 7.41*10^{-3}\log n + 7*10^{-4} \quad (1)$$

を求めることが出来る。

5　結論・今後の課題

本研究では、並列計算アルゴリズムの有用性を検証するために MPI を用いて最小全域木問題を解く時間を計測した。本研究の計測結果より、仮想並列計算環境を用いて複数の計算機で処理することで、内部の計算時間は短縮できることが示されたが、全体の処理時間は台数が増えるごとに時間がかかるようになった。これは、各計算機間での通信に時間がかかったためであり、MPI の有用性を示すためには、よりよい通信の環境を構築することが考えられる。また、分散ファイルシステム内においてファイルブロックのキャッシュを有効に活用するための協調型キャッシュアルゴリズムのような、クラスタ環境に対応したアルゴリズムを使用することで、よりよい結果を出すことができる可能性がある。新しい通信環境の構築や、クラスタ環境に対応したアルゴリズムを使った設計が今後の課題である。

謝辞

石水隆先生お呼び情報論理工学研究室の皆様には様々な助言をいただき、この研究を完成させることが出来ました。深く感謝申し上げます。

参考文献

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

[2] MPICH2, http://www.mcs.anl.gov/research/projects/ mpich2, Argonne national laboratory.

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

[4] Microsoft,http://www.microsoft.com/ja/jp/default.aspx

[5] PVM,http://www.csm.ornl.gov/pvm/

[6] OpenMP,http://www.openmp.org/

[7] Argonne National Laboratory,http://www.anl.gov/

[8] OPenMosix,http://openmosix.sourceforge.net/

[9] SCore,http://www.pccluster.org/

[10] LAM,http://www.lam-mpi.org/

[11] OpenMPI,http://www.open-mpi.org/

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。

sollin.cpp

```
#include "mpi.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 10 //


int alive[SIZE];
int hairetu[SIZE][SIZE];//配列
int changex[SIZE][SIZE];
int changey[SIZE][SIZE];
int answer[SIZE][SIZE];//答え
int parent[SIZE];//親
int min[SIZE/2];
int box = 0;
int arank;
int numberprocess;//
double begin;
double last;
MPI_Status status;
int size = sizeof parent/sizeof parent[0];

//logn を求めるメソッド
int logn(int n){
     int i = 0;
     while(n>1){
          n=n/2;
          i++;
     }
     return i;
}
```

//頂点から出ている辺の最小値を求めるメソッド

```
void S1(){
     for(int i=0; i<size; i++){
          bool check=false;
          int min=998;
          if(alive[i]==1){
               for(int j=0; j<size;j++){
                    if(alive[j]==1 && min > hairetu[i][j]){
                         min=hairetu[i][j];
                         box=j;
                         check=true;
                    }
               }
               if(check){
                    answer[changex[i][box]][changey[i][box]]++;
                    parent[i]= box;
               }
          }
     }
}
```

//頂点にプロセッサを割り振り最小値を求めるメソッド

```
void PS1(int i){
     bool check=false;
     int min=998;
     MPI_Bcast(parent,size,MPI_INT,0,MPI_COMM_WORLD);
     MPI_Bcast(alive,size,MPI_INT,0,MPI_COMM_WORLD);
     MPI_Bcast(hairetu,size*size,MPI_INT,0,MPI_COMM_WORLD);
     MPI_Bcast(changex,size*size,MPI_INT,0,MPI_COMM_WORLD);
     MPI_Bcast(changey,size*size,MPI_INT,0,MPI_COMM_WORLD);
     MPI_Bcast(answer,size*size,MPI_INT,0,MPI_COMM_WORLD);
     if(arank==0){
          if(alive[arank]==1){
               for(int j=0;j<size;j++){
                    if(alive[j]==1 && min>hairetu[arank][j]){
                         min=hairetu[arank][j];
                         box=j;
                         check=true;
                    }
```

```
                }
                if(check){
                    answer[changex[arank][box]][changey[arank][box]]++;
                    parent[arank]= box;
                }
            }
            for(int i=1;i<numberprocess;i++){
                MPI_Recv(&parent[i],1,MPI_INT,i,i,MPI_COMM_WORLD,&status);
                MPI_Recv(&alive[i],1,MPI_INT,i,i+numberprocess,MPI_COMM_WORLD,&status);
                MPI_Recv(hairetu[i],size,MPI_INT,i,i+(numberprocess*2),MPI_COMM_WORLD,&status);
                MPI_Recv(changex[i],size,MPI_INT,i,i+(numberprocess*3),MPI_COMM_WORLD,&status);
                MPI_Recv(changey[i],size,MPI_INT,i,i+(numberprocess*4),MPI_COMM_WORLD,&status);
                MPI_Recv(answer[i],size,MPI_INT,i,i+(numberprocess*5),MPI_COMM_WORLD,&status);
            }
        }else{
            if(alive[arank]==1){
                for(int j=0;j<size;j++){
                    if(alive[j]==1 && min>hairetu[arank][j]){
                        min=hairetu[arank][j];
                        box=j;
                        check=true;
                    }
                }
                if(check){
                    answer[changex[arank][box]][changey[arank][box]]++;
                    parent[arank]= box;
                }
            }
            MPI_Send(&parent[arank],1,MPI_INT,0,arank,MPI_COMM_WORLD);
            MPI_Send(&alive[arank],1,MPI_INT,0,arank+numberprocess,MPI_COMM_WORLD);
            MPI_Send(hairetu[arank],size,MPI_INT,0,arank+(numberprocess*2),MPI_COMM_WORLD);
            MPI_Send(changex[arank],size,MPI_INT,0,arank+(numberprocess*3),MPI_COMM_WORLD);
            MPI_Send(changey[arank],size,MPI_INT,0,arank+(numberprocess*4),MPI_COMM_WORLD);
            MPI_Send(answer[arank],size,MPI_INT,0,arank+(numberprocess*5),MPI_COMM_WORLD);
        }
    }
```

//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

```
void PPJ(){
    MPI_Bcast(parent,size,MPI_INT,0,MPI_COMM_WORLD);
```

```
     if(arank==parent[parent[arank]] && arank<parent[arank]){
          parent[arank] = arank;
     }
     if(arank!=0){
          MPI_Send(&parent[arank],1,MPI_INT,0,arank,MPI_COMM_WORLD);
     }
     if(arank == 0){
          for(int i=1;i < size;i++){
               MPI_Recv(&parent[i],1,MPI_INT,i,i,MPI_COMM_WORLD,&status);
          }
     }
     MPI_Bcast(parent,size,MPI_INT,0,MPI_COMM_WORLD);
     for(int j=0;j <logn(size);j++){
          if(arank == 0){
               printf("元データ%d %d %d %d %d\n",parent[0],parent[1],parent[2],parent[3],parent[4]);
               parent[arank]=parent[parent[arank]];
               for(int i=1;i < size;i++){
                    MPI_Recv(&parent[i],1,MPI_INT,i,i,MPI_COMM_WORLD,&status);
               }
               printf("Recv後%d %d %d %d %d\n\n",parent[0],parent[1],parent[2],parent[3],parent[4]);
          }else {
               parent[arank]=parent[parent[arank]];
               MPI_Send(&parent[arank],1,MPI_INT,0,arank,MPI_COMM_WORLD);
          }
     }
}

//ポインタジャンプするメソッド
void PJ(){
     for(int j=0;j <logn(size);j++){
          for(int i = 0;i < size;i++){
               parent[i]=parent[parent[i]];
          }
     }
}


void S3(){
     for(int i = 0;i < size;i++){
          if(i == parent[parent[i]]){
```

```
                for(int j= 0;j <size;j++){
                     if(i == parent[j] && i != j){
                          hairetu[i][j] = 999;
                          hairetu[j][i] = 999;
                          for(int count = 0;count < size;count++){
                               if(hairetu[i][count] > hairetu[j][count] && hairetu[i][count] != 999){
                                    hairetu[i][count] = hairetu[j][count];
                                    hairetu[count][i] = hairetu[count][j];
                                    changex[i][count] = j;
                                    changey[count][i] = j;
                               }
                          }
                          for(int count =0;count < size;count++){
                               if(hairetu[i][parent[count]] > hairetu[i][count] && hairetu[i][parent[
                                    hairetu[i][parent[count]] = hairetu[i][count];
                                    hairetu[parent[count]][i] = hairetu[count][i];
                                    changey[i][parent[count]] = count;
                                    changex[parent[count]][i] = count;
                               }
                          }
                     }
                }
           }else {
                alive[i] = 0;
           }
     }
}

bool check(int x){
     for(int i = 0; i < size ;i++){
          for(int j = i ; j < size ;j++){
               if(hairetu[i][j] == x){
                    return true;
               }
          }
     }
     return false;
}
```

//グラフを作成するメソッド

```
void Graph(){
     for(int i=0;i<SIZE;i++){
          alive[i]=1;
          parent[i]=(SIZE+i+1);
          for(int j=0;j<SIZE;j++){
               answer[i][j]=0;
               hairetu[i][j]=0;
               changex[i][j]=i;
               changey[i][j]=j;
          }
     }
     srand(time(NULL));
     for(int i=0;i<size;i++){
          for(int j=i;j<size;j++){
               if(i==j){
                    hairetu[i][j]=999;
               }else{
                    int x = (rand() % (size*2))+1;
                    while(check(x)){
                         x = (rand() % (size*2))+1;
                    }
                    hairetu[i][j] = x;
                    hairetu[j][i] = x;
               }
          }
     }
     for(int i = 0; i < size ;i++){
          for(int j = 0 ; j < size ;j++){
               printf("%3d ",hairetu[i][j]);
          }
          printf("\n");
     }
}
```

//メインメソッド

```
int main(int argc,char **argv){
     MPI::Init(argc,argv);
     MPI_Comm_size(MPI_COMM_WORLD,&numberprocess);
     MPI_Comm_rank(MPI_COMM_WORLD,&arank);
     begin = MPI_Wtime();
```

```
    if(arank == 0) Graph();
    for(int i = 0;logn(size) > i ;i++){
        PS1(arank);
        PPJ();
        S3();
    }
    if(arank ==0){
        for(int i = 0; i < size ;i++){
            for(int j =0;j<size;j++){
                printf("%2d ",answer[i][j]);
            }
            printf("\n");
        }
        last = MPI_Wtime();
        printf("かかった時間：%10.8f \n",last-begin);
    }
    MPI::Finalize();
}
```