卒業研究報告書

題 目

MPI を用いた並列計算処理

指 導 教 員

石 水 隆 助教

報 告 者

07–1–037–0066

八 木 佑 介

近畿大学理工学部情報学科

平成 23 年 1 月 28 日提出

概要

現在、様々な分野で計算処理の高速化が求められている。高速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) が用いられる。しかし、一般に並列計算機は非常に高価であり、容易に用いることはできない。そこで、複数の PC をネットワーク接続して 1 台の仮想的な並列計算機とする仮想並列計算 (Parallel Virtual Computing) が現在重視されている。本研究は、仮想計算機を構築するソフトウェアの一つである MPI(MessagePassing Interface)[2] を用いてその性能を実験的に評価する。評価方法として、グラフ問題の一つである最小全域木問題を MPI を用いて並列処理で解き、1 台で解いた場合と比べてどれだけの処理時間の短縮が出来ているのか検証を行なう。MPI とは分散メモリ型並列計算機 (Distributed Memory Parallel Computer) において、複数のプロセッサ間で、データのやりとりをするために用いる、メッセージ通信操作の仕様標準である。

目次

1 **序論** 1

1.1 並列処理 1
1.2 仮想並列計算機 1
1.3 仮想並列計算機を構築するソフトウェア 1
1.4 最小全域木問題 2
1.5 本研究の目的 2
1.6 本報告書の構成 2

2 **準備** 3

2.1 使用 PC 3
2.2 MPI (Message Passing Interface) 3
2.3 MPICH2 の導入 4

3 **実行内容と実行結果** 5

3.1 実験内容 5
3.2 最小全域木問題 5
3.3 Sollin のアルゴリズム 5
3.4 最小全域木問題を解く MPI プログラム 6
3.5 実行結果 7
3.6 考察 7

4 **結論・今後の課題** 9

付録 A ソースプログラム 12

1 序論

1.1 並列処理

現在、様々な情報が PC で取り扱われている。取り扱われる情報の量は日々増大しており、その処理時間を短縮することは PC を使用する上での重大な課題である。ある 1 つの処理を、複数のプロセッサを用いて協調して行う事で単一のプロセッサでの処理よりも高速に計算処理を行うことを並列処理（Parallel Processing）という。並列処理を用いることにより、処理時間の大幅な短縮が得られ、処理能力も向上すると期待されている。データの高速処理には、複数のプロセッサを持つ並列計算機は非常に有用である。しかし一般に並列計算機は非常に高価であるため容易に利用できない。このため、近年、複数の PC 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 (Parallel Virtual Computer) として用いるクラスタ (Cluster) 処理が注目されている。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。

1.2 仮想並列計算機

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が用いられる。しかし、一般的に並列計算機 (Parallel Virtual Computing) は高価であるために、容易に用いることはできない。そこで、複数の PC をネットワーク接続することにより仮想的な並列計算機として利用する仮想並列計算 (Parallel Virtual Computing) が現在注目されている。仮想並列計算機を構成するソフトウェアには無償で提供されているものもあるため、安価で並列計算機を構築できる。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)[2], PVM(Parallel Virtual Machine)[5], OpenMP[6] 等がある。以下に、これらについて説明する。

1.3 仮想並列計算機を構築するソフトウェア

MPI(Message Passing Interface) は、メッセージ通信のプログラムを記述するために広く使われる標準を目指して開発された、メッセージ通信の API 仕様である。MPI は 1992 年に結成された MPI Forum(MPIF) により標準使用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、ヨーロッパの 40 の組織から 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に様々な通信関数が実装されている、MPI 規約を用いて作成したプログラムは移植性が高いため、MPI を使用するユーザは、通信を考慮せずプログラムを組むことが出来る。MPI のサポートするプログラミング言語は多く、C 言語や Fortran そして最近では Java などに対応している。無料で提供されている MPI の主な実装は MPICH[3] や LAM[8]、OpenMPI[7] といったものがある。

PVM(Parallel Virtual Machine)[5] は米国のオークリッジ国立研究所を中心に開発された、メッセージパッシングによる並列計算を行うためのソフトウエアであり、動作するマシンの種類が多く、比較的容易に入手できる。PVM は、アプリケーション、マシン及びネットワークレベルでの異機種間利用をサポートする。言い替えると、PVM の下では、アプリケーションを構成するタスクは、 問題に最も適したアーキテクチャを利用することができる。PVM は、異なる PC 間の整数あるいは浮動小数点数の表現の違いを吸収するための、

データ変換を扱うことができる。そして、PVM を使用することで多様なネットワークで接続された仮想計算機を実現することができる。

OpenMP[6] は主に共有メモリ型並列計算機で用いられ、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。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。国内で実務に使っている例は非常に少ないが 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 OpenMosix[9] や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にて開発された Linux 用クラスター計算機用超並列プログラム実行環境 Score[10] などがある。

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。

本研究では MPI の実装として幅広く用いられている MPICH2[3] を用いる。MPICH2 はアメリカのアーゴン国立研究所が模範実装として開発し、無償でソースコードを配布したライブラリである。移植しやすさを重視した作りになっているため、盛んに移植が行われ、世界中のほとんどのベンダの並列マシン上で利用することができる。

1.4 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、全ての頂点を含む G の部分木のうち、重みの和が最小となるものを求める問題である。最小全域木問題に対して、Prim は $O(m + n \log n)$、Kruskal は $O(m \log n)$、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]。

1.5 本研究の目的

本研究では、MPI を用いた並列計算の有用性を検証する。その検証方法として、複数の PC を LAN 接続して MPI 環境を構築し、性能検証用の問題に対して MPI を用いて並列処理した場合に、逐次処理した場合と比べてどの程度処理時間が短縮できるかを計測する。本研究では、性能検証用の問題として最小全域木問題を用いる。

1.6 本報告書の構成

本報告書の構成を以下に述べる。2 章では本研究を始めるにあたっての準備。3 章では実行内容を説明し、実行結果を示す。4 章では結論、今後を述べる。

表 1　本研究で使用した PC 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

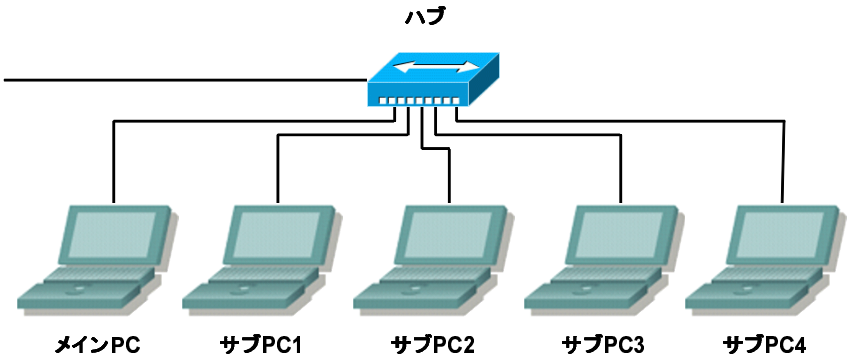

図 1　ネットワーク構成図

2　準備

2.1　使用 PC

本研究では、MPI による仮想並列環境を構築するために 100Base-TX による LAN 接続された 5 台の PC を使用する。本研究で使用した PC の性能を表 1 に示す。また、本研究で使用した LAN の構成図を図 1 に示す。本研究では、OS として Windows 系 OS を用いた。Windows 系 OS を使うのは 2010 年現在では一般向けの PC の大半で使用されており、日頃から使い慣れているからである。

2.2　MPI (Message 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 Supercomputing ’92 会議において、後に MPI フォーラムとして知られることになる委員会が結成され、メッセージ・パッシングの標準を作り始めたことで具体化した。これには主としてアメリカ、ヨーロッパの 40 の組織から 60 人の人間が関わっており、産官学の研究者、主要な並列計算機ベンダのほとんどが参加した。そして Supercomputing ’ 93 会議において草案 MPI 標準が示され，1994 年に初めてリリースされた。

無料で提供されてい MPI の主な実装として、MPICH2[3] や LAM[8]、OpenMPI[7] 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[3] で開発された。2005 年には MPICH の後継として MPICH2 が開発された。MPICH2 では、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

環境の確立を可能にし、問題発生時にも問題の原因究明が容易に行える。

LAM はノートルダム大学の科学コンピュータ研究室が作成したフリーの MPI ライブラリである。MPICH と違い、デーモンを介して通信を行うので、MPICH に比べて通信が高速になる。OpenMPI は、他のいくつかの FT-MPI、LA-MPI、LAM/MPI、PACX-MPI といった技術や資源を融合し、利用できる最善の MPI ライブラリを構築するために始められたプロジェクトである。

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 MPICH よりも故障に強く、効果的で、使いやすいからである。

2.3 MPICH2 の導入

本節では、MPICH2 のインストールと環境設定について述べる。本研究を始めるにあたり、まず MPI を構築する全ての PC に MPI による仮想並列計算環境を構築する必要がある。本研究では、その実装には MPICH2[3] を用いた。これは、MPI 規格を実装したフリーのライブラリ群である。MPICH2 のインストールの仕方について説明する。MPICH2 を使用するにはまず全ての PC に MPICH2 をインストールする。公式サイト [3] より Windows 用 MPICH2 の実行形式のインストーラファイルをダウンロードし、そのファイルを実行することにより、インストーラが起動し自動的にインストールされる。デフォルトでは $C:¥ProgramFiles¥MPICH2$ フォルダにインストールされる設定になっているので、適当なフォルダを指定する。インストール後の環境設定として、インストール先のフォルダにある bin ファイルに環境変数でパス (”C:¥Program Files¥MPICH2¥bin”) 指定をする。

Windows で MPICH2 を使用するにあたって、全ての使用する PC に管理者権限を持つ同じ名前、パスワードを持ったアカウントを作成する。プログラムの作成には Microsoft 社製の Visual C++2008Express Edition[4] を使用する。

3 実行内容と実行結果

3.1 実験内容

本研究では、同一機種の PC5 台を LAN 接続し MPI 環境を構築する。また、MPI の性能検証用の問題として、本研究では最小全域木問題を用いた。

本研究では、グラフ頂点数が 5,10,20,40,80,160 の重み付無向グラフが与えられたとき PC1〜5 台を用いて MPI 上で最小全域木問題を解く時間を計測した。

3.2 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、「辺の重みの総和」が最小となる G の全域木を求める問題である。

重み付無向グラフとは、グラフ $G = (V, E)$ において、G の全ての辺 $(u, v) \in E$ $(u, v \in V)$ に重み $w(u, v)$ が付加されており、また、全ての辺 (u, v) に対して辺 (v, u) が存在し、$w(u, v) = w(v, u)$ を満たすグラフである。また、全域木とは連結グラフ G に対して、頂点集合が G と一致し閉路を含まない連結部分グラフを、G の全域木と呼ぶ。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 [1] 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 $O(m \log n)$、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。

3.3 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 C。C の各要素 $C_{x,y}$ $(0 \leq x, y < n)$ 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: 配列用変数 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、それぞれの根 $r[v]$ 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。この時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 C 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: 定数個の代入に必要な定数時間である。

step 2-1: 定数個の足し算をする定数時間である。

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、時間は $\frac{n^2}{\log^2 n}$ プロセッサで $O(logn)$ となる

step 2-3: ポインタジャンプすることにより距離が半分になり $\log n$ 回で求められ、時間は $\frac{n^2}{\log^2 n}$ 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 $O(logn)$ 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。

3.4 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。**??**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 の部分行列をメイン PC からサブ PC に送信する。このとき、サブ PCi $(0 \leq i < k)$ には top の要素 $A_{x,y}$ $(0 \leq x < n, 0 \leq y < \frac{ni}{k})$ を送信する。(送信する要素は作成したプログラムに合わせてください。)

step 1: ループ範囲を指定しつつ、頂点ごとの最小の辺を選択し、保存する

step 2-1: ポインタがお互いに送信し合っている場合、小さい数字に付け変える

step 2-2: 自分の親を親に付け変える

step 3-1: 選ばれた辺は選ばれないようにする

step 3-2: 親が自分自身の頂点の場合、子のデータを親に付け変える

step 3-3: 親が自分自身でなかった場合は頂点をないものとし、以後の処理を行わない

step 0-2 **から** step3: $\log n$ 回繰り返す

表 2　全体の処理時間と PC 数の関係

台数	頂点 5	頂点 10	頂点 20	頂点 40	頂点 80	頂点 160
1 台	0.003	0.004	0.0056	0.02	0.30	5.7
2 台	0.009	0.011	0.01	0.04	0.35	6.0
3 台	0.013	0.017	0.025	0.07	0.45	6.0
4 台	1.4	2.0	2.6	3.2	4.0	13
5 台	1.2	2.0	2.6	2.4	3.6	10

(m 秒)

表 3　内部計算時間と PC 数の関係

台数	頂点 5	頂点 10	頂点 20	頂点 40	頂点 80	頂点 160
1 台	0.000004	0.000009	0.000018	0.000040	0.00100	0.0027
2 台	0.000004	0.000008	0.000011	0.000030	0.00060	0.0016
3 台	0.000004	0.000006	0.000010	0.000025	0.00035	0.00095
4 台	0.000004	0.000006	0.000010	0.000020	0.00026	0.00085
5 台	0.000004	0.000006	0.000010	0.000018	0.00022	0.00068

(m 秒)

また、本研究では、MPI 上で上記のアルゴリズムを処理するために PC のうちの 1 台をメイン PC として処理させる。メイン PC はランダムに作ったグラフを他の PC に送る。またメイン PC はアルゴリズム終了後に他の PC から送られた結果からもう一度処理を繰り返すか判断する。

3.5　実行結果

表 2 に頂点数が 5,10,20,40,80,160 のそれぞれの場合で PC1〜5 台を用いて MPI 上で最小全域木を求めたときの全体の処理時間を示し、表 3 にメイン PC での通信時間を含まない内部処理時間を示す。ただし、メイン PC とは、入力となる重み付グラフ G を作成し、G の部分グラフを各サブ PC にネットワークを通して送信する PC である。表 2 より、全体の処理時間はサブ PC が増えるごとに遅くなっていることが示される。表 3 より、メイン PC の内部計算時間についてはサブ PC の台数が増えるごとに速くなった。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 CPU 台数の増加による内部計算時間の短縮効果が得られた。MPI を用いて並列処理することにより処理の高速化が期待されたが、結果は表 1 に示す通り予想に反したものとなった。これは内部計算時間は速くなったが、各 PC 間での通信時間が長くなったからであると考えられる。

3.6　考察

表 3 に示されるように、内部計算時間については MPI を用いて並列処理することにより処理時間の短縮が得られた。しかし、全体の処理時間については、MPI を用いて並列処理することにより処理の高速化が期待

されたが、結果は表 1 に示す通り予想に反したものとなった。これは内部計算時間は速くなったが、各 PC 間での通信時間が長くなったからであると考えられる。また、表 2 において、PC 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表 1 に示すようにサブ PC3 のメモリが他の PC の半分であり他の PC よりも処理性能が低い。そのためサブ PC3 の処理時間がボトルネックとなり全体の処理時間が増加したと思われる。この仮説を検証するため、MPI ネットワークに PC を加える順番を変更して実行したところ、台数に関係無く常にサブ PC3 を MPI ネットワークに加えたときに通信計算量の増加が起こった。よって原因がサブ PC3 であることが裏付けられる。

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、

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

と置き、表 3 の値から連立方程式を立てる。この連立方程式より、

$$T_{comp}(n,p) = \frac{1.5n^2 * 10^{-4} - 45.0n * 10^{-4}}{p} + 1.22\log^2 n * 10^{-4} - 2.45\log n * 10^{-4} + 1.16 * 10^{-4} \tag{2}$$

が得られる。

4　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。しかし本研究で得られた計測結果では PC 数の増加に反して全体の処理時間は長くなっている。よって本研究からは MPI の有効性を検証できたとは言えない。並列化によって内部計算時間は時間が短くなったが PC 数の増加により各 PC 間での通信回数が増えてしまったため通信時間が長くなり、全体の処理時間が延びたと思われる。この原因としては PC の性能など実行環境に問題があり、並列処理による処理の高速化を得るためには、高スペックな PC 同士のネットワーク構成など実行環境を整え、プログラムのアルゴリズムを改善することが必要であると思われる。PC の性能については今回１台のみメモリが半分であったため全ての PC が高スペックな物にすれば結果は良くなったと思われる。アルゴリズムについては通信を考慮した BSP[11] や CGM[12] のアルゴリズムを使うことで改善されると思われる。

謝辞

研究を進めていくにあたり、石水先生に基礎知識から研究内容まで指導や助言をいただきましたことを心より感謝を申し上げます。

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] MPICH2, http://www.mcs.anl.gov/research/projects/mpich2/.

[4] Visual C++2008, http://www.microsoft.com/japan/msdn/vstudio/express/.

[5] PVM Parallel Virtual Machine, http://www.csm.ornl.gov/pvm/pvm_home.html.

[6] OpenMP, http://rest-term.com/contents/misc/technote/index.php/OpenMP

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

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

[9] OpenMosix, http://theochem.chem.nagoya-u.ac.jp/wiki/wiki.cgi/ClusterBuild?page=OpenMosix

[10] Score 操作マニュアル, 日本電気株式会社 HPC エンジニアリングセンター, http://www.gsic.titech.ac.jp/TITechGrid/SCore-manual.htm

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

[12] 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).

付録 A　ソースプログラム

以下に、本研究に使用した最小全域木問題を解く MPI プログラムを示す。

Stree.mpi

```
#include "mpi.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 80 //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[tantou][box]][
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のが見合っている所を、プロセッサ番号が小さい方を
    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();



}
```