卒業研究報告書

題 目

MPI を用いたグラフの並列計算

指 導 教 員

石 水　隆 助教

報 告 者

07–1–037–0213

藤 本　涼 一

近畿大学理工学部情報学科

平成 23 年 1 月 28 日提出

概要

計算機の高速処理化についての研究・開発は、近年ますます発展している分野である。大容量の記憶デバイスの登場や、ネットワークの高速化などにより、大量のデータを高速に処理することが求められており、現在様々な角度から計算機の向上が目指されている。高速処理化を実現させる手段として、並列計算機 (Parallel Computer) による並列処理 (Parallel Processing) が注目されている。だが、並列計算機は一般的には高価なものであり、個人で使用するにはあまりに困難である。だが、ネットワーク接続された複数の計算機を用いて仮想的に 1 台の並列計算機を構築する仮想並列計算機 (Parallel Virtual Computing) を用いればコストをかけずに並列計算をできる環境を整えることができる。本研究では、仮想並列計算機を可能とさせるソフトウェアである MPI(Message Passing Interface)[2] を用いてその有用性を評価する。MPI とは、分散メモリ型の並列計算機において、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。本研究では、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う。検証を行うための問題としては、最小全域木問題を用いる。本研究では、頂点数 5,10,20,40,80,160 のグラフに対して 1 から 5 台の計算機を用いて MPI 上で最小全域木問題に対する処理速度の比較を行い、その結果により MPI による仮想並列計算機の有用性を実験的に評価する。

目次

1 序論

1.1 本研究の背景

1.1.1 並列処理 (Parallel Processing)

並列処理 (Parallel Processing) とは、計算機において複数のプロセッサで 1 つのタスクを動作させることである。問題を解く過程はより小さなタスクに分割できることが多い、という事実を利用して分割された小さなタスクを複数のプロセッサに並列処理を行わせることにより、処理時間を短縮することができ、処理効率の向上につながるといった手法である。

1.1.2 並列計算機 (Parallel Computer)

並列処理を行うために用いられるのが並列計算機 (Parallel Computer) である。並列計算機は複数のプロセッサを持ち、各プロセッサが協調して動作することにより高い処理能力を得ることができる。並列計算機による処理の方法は、共有メモリ (shared memory) 型と分散メモリ (distributed memory) 型の 2 つに大きく分類できる。共有メモリ型の並列計算機は、それぞれが同じメモリを通して計算するため同期やデータの送受信が対処しやすいという長所があるが、プロセッサの増加によりメモリにプロセッサを接続させることが困難になるといった短所がある。また一方、分散メモリ型による並列計算機はプロセッサが個々のメモリを持つといった特徴から、複数のプロセッサを多数接続させる並列計算機を構築しやすくなるといった長所をもっており、そのことから現在では分散メモリ型並列計算機が主流となっている。一方分散メモリ型並列計算機の短所としては、他のプロセッサの持つデータをすぐに参照できないといったことが挙げられる。

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 (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。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)[3], OpenMP[5] 等がある。以下に、これらについての説明をする。

1.1.3 仮想並列計算機を構築するソフトウェア

MPI(Message Passing Interface)[2] とは並列計算機を実装するための標準化された規格であり、実装自体を指すこともある。複数の CPU 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えるようにする。ライブラリレベルでの並列化であるため、言語を問わず利用でき、プログラマが緻密なチューニングを行える一方、利用にあたっては明示的に手続きを記述する必要があり、デッドロックの対処などもプログラマ側が大きな責任を持たなければらないことなどが挙げられる。業界団体や研究者らのメンバからなる MPI Forum によって規格が明確に定義されているため、ある環境で作成したプログラムが他の環境でも動作することが期待できる。MPI は専用の並列計算機からワークステーション、パーソナルコンピュータに至るまで幅広くサポートしている、無料で提供されている主な実装は MPICH[6] や LAM[9] といったものがある。

PVM(Parallel Virtual Machine)[3] は、1991 年に米国のオークリッジ国立研究所 (Oak Ride National Laboratory)[4] 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 TCP/IP の通信ライブラリで一般的に使用されている LAN(Local Area Network) があれば並列処理が実行可能であり、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まで多くの種類の計算機で並列処理環境を構築することが出来る。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、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。PVM を実行中は、PVM によって構成された仮想並列計算機上にある各々の計算機にデーモンが存在し、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ラップさせ仮想並列計算機を構成することが出来る。ライブラリは、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。また、各ユーザは PVM アプリケーションを 1 台の計算機上で複数実行することが可能となっている。PVM の大きな特徴として、耐故障性 (Fault Tolerant) が挙げられる。通常、仮想並列計算機は計算中にある 1 台が停止してしまうと、計算処理が出来なくなってしまうが、PVM では、任意で計算機の追加や削除が行なえると共に、故障した計算機を仮想並列計算機内から迅速に削除され、計算処理自体が停止してしまうことなく続行できる。また、PVM の問題点として PVM は多くの並列計算機に移植されるようになったために、各並列計算機ベンダが独自にチューニングを行なった PVM を開発してしまっており、PVM で作成をしたプログラムの移植性が乏しくなってしまったことが挙げられる。

OpenMP[5] は、並列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。複数の CPU が一つのメモリを共有するアーキテクチャでの並列性を記述するための API（Application Program Interface）仕様でであり、この API 仕様をサポートするベンダが作ったコンパイラを使えば、並列的に動作するソフトウェアを作ることができる。

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[6] を用いる。MPICH2 は高性能かつ広い携帯性を実現した MPI 規格である。MPI は自由に利用可能なライセンスとして配布されており、Linux(IA32 と x86-64)、Mac OS/X(パワー PC とインテル)、Solaris(32 と 64 ビット)、および Windows のプラットフォームで利用可能となっている。

1.1.4 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、G の全ての頂点を含み連結かつ閉路を持たない部分グラフのうち、辺の重みの和が最小になるグラフ T を求める問題である。頂点数 $|V| = n$, 辺数 $|E| = m$ の重み付無向グラフに対す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]。

1.2 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複数の計算機をネットワーク接続して MPI 環境を構成し、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、MPICH2 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っている。検証を行うための問題としては、最小全域木問題を用いる。

1.3 本報告書の構成

本報告書の構成を以下に述べる、2 章には本研究における検証対象である MPI および MPI を実装するために使用した機器とソフトの設定方法、検証用の問題である最小全域木問題ついて述べる。3 章に結果、考察、4 章に結論と今後の課題について述べる。

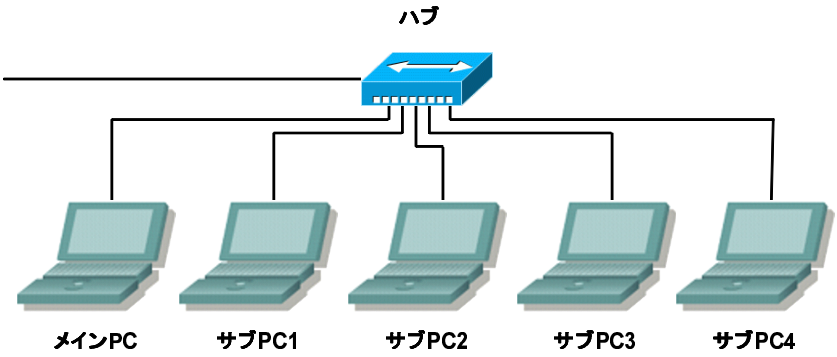

図 1　本研究で使用した計算機ネットワークの概念図

2　研究内容

2.1　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た。MPI の標準化への取り組みは 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

2.2　MPICH2

MPICH[6] は MPI を実装するためのソフトウェアとして Argonne 国立研究所 [7] で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。MPICH は移植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。2005 年には MPICH の後継として MPICH2 が開発された。

2.3　使用機器

本研究では、MPI による仮想並列環境を構築するために 5 台の計算機を使用する。図 1 に本研究で使用した計算機ネットワークの概念図を示す。それぞれの計算機は、100Base-TX のイーサーネットケーブルとハブにより、ネットワークが構成されている。本研究では、1 台の計算機をメイン PC、他の 4 台の計算機をサブ PC として扱う。表 1 に、本研究で用いた計算機の性能を示す。

2.4　MPICH2 のインストールと環境設定

本節では、MPICH2 のインストールと環境設定について述べる。MPICH2 を使用するためにまず、MPI を構築する全ての計算機に MPICH2 をインストール (Install) が必要である。MPICH2 はインターネッ

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
Memory	1GB	1GB	1GB	512MB	1GB

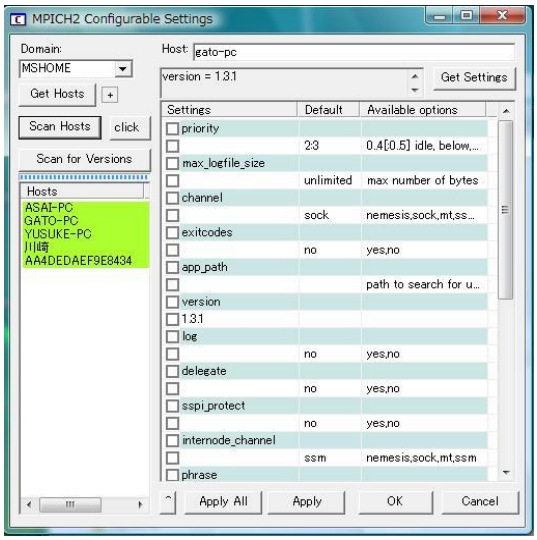

図 2　wmpiconfig のウィンドウ

ト上の MPICH2 のウェブページ [6] において無料で配布されているので使用する OS に合うものをダウンロードし、インストールする。本研究では 2010 年 12 月現在の Windows 用 MPICH2 の最新版である mpich2-1.0.6p1-win32-ia32.msi を各計算機にダウンロードし、C:¥Program Files¥MPICH2 の下にインストールを行った。Windows で計算機を使用するにあたって、全ての使用する計算機に管理者権限を持つ同名のアカウント mpi を作成し、各計算機で同一のパスワードを設定した。また、各計算機上に”C:¥Stree”という、ネットワークを通じて共有することができるフォルダを作成した。インストールが完了すると、各計算機の MPICH2 のバイナリのあるフォルダ (本研究では”C:¥Program Files¥MPICH2¥bin”) に対して環境変数の PATH の指定をしておく必要がある。PATH が通っているかどうかを確かめるにはコマンドプロンプト上で”mpiexec”命令を実行させたときに、引数の入力を促す Usage メッセージが表示されているかどうかで判断できる。

2.5　MPICH2 の実行方法

本節では、MPICH2 の実行方法について述べる。まずは”C:¥Program Files¥MPICH2¥bin”内にある wmpiconfig.exe を起動して参加する計算機の接続を行う。図 2 に wmpiconfig.exe を起動したときに表示されるウィンドウを示す。ウィンドウの左上に Domain という項目があり、そこから使用する Domain を指定する。次に Get Hosts というボタンを押すと、現在その Domain に参加している計算機が表示される。その

後 Scan Hosts というボタンを押すと、Domein に参加している計算機と継っていれば図 2 のようにその計算機が緑色で表示される。継っている事を確認できれば、Apply All を押すことにより、この wmpiconfig.exe 内での作業は終了となる。

wmpiconfig.exe の設定後、コマンドプロンプト上で mpiexec 命令を実行することにより mpich2 を起動できる。MPICH2 を 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、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。

2.6 Visual C++ のインストール

本研究では、MPI 上のプログラム言語として C++ 言語を用いる。本研究では C++ を実行できる環境を作るために、Visual C++ のインストールを行う。Visual C++ 2008 Express Edition は、マイクロソフトの公式ページ [8] より配布されているので、その仮想 CD をダウンロードすることにより、インストールすることができる。また、Visual C++ のツールオプションより、MPICH2 のインクルードファイルおよびライブラリファイルのあるフォルダ ”C:¥Program Files¥MPICH2¥include” および ”C:¥Program FilesY =MPICH2¥lib” を追加し、リンカ入力の依存ファイル”mpi.lib”を追加することにより、MPICH2 を用いて並列プログラムを作成する環境を作る。

2.7 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 の全ての頂点を含み連結かつ閉路を持たない部分グラフのうち、辺の重みの和が最小になるグラフ T を求める問題である。重み付無向グラフとは、G の全ての辺 $(u, v) \in E$ $u, v \in V$ に対して、辺 (v, u) が存在し、かつ辺 (u, v) の重みと辺 (v, u) の重みが等しいグラフである。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 [1] が有る。それぞれの計算量は辺の数 $|E| = m$、頂点の数 $|V| = n$ の重み付無向グラフ $G = (V, E)$ 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容易に並列アルゴリズムを得ることが可能である。並列化した場合 Sollin のアルゴリズムの計算時間は p プロセッサ CREW PRAM 上で $O(\frac{n^2}{p} + \log^2 n)$ で最小全域木問題を解くことができる。

本研究では最小全域木問題を解くアルゴリズムとして Sollin の並列アルゴリズムを用いる。

以下に Sollin の並列アルゴリズムを示す。また、以下では、グラフ $G_k = (V_k, E_k)$ $(0 \leq k < \log n)$ の隣接行列を W_k と表記する。

[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ループする
　　step 2-1: 配列用変数 k を 1 増やす

step 2-2: 各頂点 $v \in V_k$ において、v から出てる辺 (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、根付有向森で v 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 $r[u]$ のデータを集める。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 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。

step 1: 定数個の代入だけなので定数時間で可能である。

step 2-1: 定数個の足し算を行うだけなので定数時間で可能である。

step 2-2: 2 つの辺の大小比較を行っていくことで、1 度の操作で比較する辺を半分にすることができるので、$\log n$ 回繰り返すことによって最小値を求めることができる、よって step 2-2 の計算量は $\frac{n^2}{\log^2 n}$ プロセッサで $O(\log n)$ である。

step 2-3: 1 度のポインタジャンプを行うことで根から葉までの距離を半分にすることができるので、$\log n$ 回繰り返すことによって根 $r[v]$ を求めることができる、よって step 2-3 の計算量は $\frac{n^2}{\log^2 n}$ プロセッサで $O(\log n)$ である。

step 2-4: データを集める際に 2 つのデータを比較して、小さい方を更新していくことで、一回の操作で更新する頂点の数が半分になるので、$\log n$ 回繰り返すことによってデータを根に集めることができる。よって step 2-4 の計算量は $\frac{n^2}{\log^2 n}$ プロセッサで $O(\log n)$ である。

step 3: 各辺にプロセッサを割り当てることで定数時間で実行可能だが、step2 は $\log^2 n$ 時間より短くならないので、実際には $\frac{n^2}{\log^2 n}$ 台のプロセッサを使用して $\log^2 n$ 時間で実行することが現実的である。

step1、step3 は定数時間で実行可能であり 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。

2.8 検証用プログラム

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後生成される。

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 と頂点に関するデータをメイン PC からサブ PC に送信する。

step 1: ループ範囲を ループ変数 tantou = (頂点数 size*自分の PC 番号 myrank)/総 PC 数 numprocs;

tantou < (size*(myrank+1))/numprocs ;tantou++ と指定することで、自分の担当する頂点だけを処理できるようにする。ループで担当した頂点から継っている辺かつ、その辺の先にある頂点が生きている辺の中で最小の辺を選択する。選んだ辺が step 3 で変更が加えられなかったかどうかを判断し、答用の配列に今回選択した辺を保存する

step 2-1: ポインタがお互いを見合っている頂点が有る場合、頂点番号の少ないほうのポインタを自分自身につけかえる。

step 2-2: 担当している頂点の oya 配列の値を $oya[oya[$担当している頂点$]]$ の値に書き換える

step 3-1: 今回選択された辺を、次回から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、子のデータを親につけかえるという処理を行う。この操作が行われるのは以下の場合である。

1: 親から対象の頂点に行くより、子からその頂点に行く時の方が重みが少ない場合、親から対象の頂点に行く辺の重みを、子から対象の頂点に行く辺の重みにつけかえる

2: 親から対象の頂点に行くより、対象の頂点の子に行く時の方が重みが少ない場合、親から対象の頂点に行く辺の重みを、親から対象の頂点の子に行く辺の重みにつけかえる

以上の操作が行われた場合、変更が行われた場所を保存する配列に変更された箇所を保存する。

step 3-3 親が自分自身でなかった頂点は殺し、以後処理は行われない。

step 0-2 **から** step 3: $\log n$ 回繰り返す

本研究では、入力となる重み付無向グラフの頂点数が 5,10,20,40,80,160 のそれぞれ場合に対して、MPI 上で 1 から 5 台計算機を用いた場合の計算時間の測定を行う。

表 2　内部計算時間と計算機台数の関係

台数	頂点 5	頂点 10	頂点 20	頂点 40	頂点 80	頂点 160
1 台	0.000022	0.000059	0.000168	0.000440	0.001000	0.002700
2 台	0.000016	0.000052	0.000141	0.000385	0.000600	0.001600
3 台	0.000010	0.000034	0.000090	0.000255	0.000350	0.000950
4 台	0.000010	0.000028	0.000085	0.000200	0.000260	0.000850
5 台	0.000010	0.000028	0.000060	0.000148	0.000220	0.000680

(m 秒)

表 3　全体の計算時間と計算機台数の関係

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)

3　結果・考察

表 2 に頂点数 5,10,20,40,80,160 のそれぞれのグラフに対して 1 から 5 台の計算機を用いて MPI 上で最小全域木を求めた場合のメイン PC での通信を除いた内部計算時間の合計を示し、表 3 に各頂点数での通信を含んだプログラム開始から終了までの計算時間を示す。表 2 より、各頂点数のグラフに対して計算機数の増加に伴い内部計算時間が減っていることが示される。計算機数増加による内部計算時間の減少の割合は、頂点数が多いグラフほどより顕著であり、頂点数が少ない場合においては計算機数の増加しても内部計算時間が減少しない場合もある。従って、MPI による時間短縮効果は、頂点数の大きいグラフに対する処理、すなわち計算量が大きい処理に対してより効果的に得られ、計算量が小さい処理に対しては並列計算があまり有効でないことがが示される。一方、表 3 に示されているように、計算機台数の増加に伴い内部計算時間が減少しているにも関わらず全体の処理時間は大きくなる。この事から計算時間増大の原因は通信時間であると考えられる。また、表 3 より、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サブ PC3 とそれ以外の PC とのメモリ容量の違いによるものだと考えられる。またこの検証として 4 台目と 5 台目を入れ換えて実行したところ、今度は 5 台目を増やした時に計算時間が増大ことから、処理時間の急激な増加は計算機の性能の違いによるものだと裏付けられる。この事より他の計算機に比べて処理能力の大きく劣る計算機を用いての並列計算はあまり有効でないと考えられる。

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。この連立方程式より、

$$T_{comp}(n,p) = \frac{0.86 * 10^{-6} * n^2 + (-109) * 10^{-6} * n + 4.6 * 10^{-3}}{p} + 1.94 * 10^{-6} log^2 n + 21.4 * 10^{-6} * n - 134 * 10^{-6} \quad (1)$$

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 を用いて最小全域木を解く時間を計測した。しかし本研究からは内部計算自体を早くすることはできたが、通信に時間がかかってしまったため、MPI の有用性が十分示せたとは言えない。今後の課題としては、MPI 上で通信も含めた処理時間の短縮を得ることが挙げられる。短縮のためには、通信環境を整備すること、通信も考慮し並列アルゴリズムを改良することが必要であると考えられる。並列アルゴリズムの改良には、例えば今回のプログラムでは変数毎に通信を行ったが、複数の変数を同時に送信できるようにしたり、各サブ PC と個別に通信を行った場所があったのを無くし、全ての PC が同時に通信できるようにすることが考えられる。また今回のプログラムではメイン PC での処理が多くなってしまっているので、データの書き換えをそれぞれの PC で行う事ができればメイン PC の負担が軽くなり処理時間の短縮できると考えられる。また通信の同期を取るタイミングについても考慮する必要が有る。

謝辞

本研究を進めるにあたり、石水先生には遅くまで御指導頂きありがとうございました。また同じ研究室の皆様には多くの助言や励ましをもらい大変感謝しております。この 1 年間皆様本当にありがとうございました。

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] http://www.csm.ornl.gov/pvm/pvm_home.html : PVM 公式ページ

[4] http://www.ornl.gov/ : オークリッジ国立研究所

[5] http://openmp.org/wp/ : OpenMP 公式ページ

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

[7] Argonne ホームページ,http://www.mcs.anl.gov/index.php

[8] Visual Studio 2008 Express Editions,
http://www.microsoft.com/japan/msdn/vstudio/2008/product/express/

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

付録 A　最小全域木問題を解く MPI プログラム

以下に、本研究で使用した最小全域木計測用 C++ プログラムを示す。

```
//Stree.cpp
#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[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のが見合っている所を、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();



}
```