卒業研究報告書

題 目

MPIによる並列計算

指 導 教 員

石 水 隆 助教

報 告 者

07–1–037–0142

廣 岡 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

近年、IT は急激に進歩を続けて、もはや IT 無しには個人生活もビジネスも成り立たない状況になってきている。それに伴いそれらを処理する計算機には、膨大な高速処理が求められるようになった。高速処理の手段として、1 台の計算機の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 台の計算機処理速度の向上には限界があり、性能を向上させるには膨大な時間や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。だが高速処理が可能となる並列計算機はとても高価なものである。そのため、ネットワークで複数の計算機を繋げることにより、計算機群から 1 台の安価で処理能力の高い仮想的な並列計算機を作り出すことが注目されている。

目次

1 序論

1.1 本研究の背景

1.2 並列処理

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、また、処理能力も向上すると期待されている。並列処理は地球規模の環境変動の解明・予測に用いられる地球シミュレータなど、極めて大きな計算処理を必要とする問題の解決に用いられる。

1.3 並列計算機

複数のプロセッサを用いることで並列処理を行うことが可能な計算機を並列計算機 (Parallel Computer) という。並列計算機には、全てのプロセッサが共有メモリ (Shared Memory) に対して読み書きを行い、プロセッサ間の通信はメモリを通して行う共有メモリ型並列計算機 (Shared Memory Parallel Computer) と、それぞれのプロセッサは個々に局所メモリ (Local Memory) を持ち、通信にはネットワーク間でのメッセージの送受信を使用する分散メモリ型並列計算機 (Distributed Memory Parallel Computer) の大きく 2 つに分けることができる。共有メモリ型並列計算処理は計算機間の同期の問題やデータの送受信といった問題に対してはメモリを共有しているため対処しやすいが、プロセッサの増減などの変化があった場合には、メモリに全てのプロセッサを接続させることが困難になってしまう。そのために、現在では局所メモリが使用できる分散メモリ型並列処理が主流となっている。一方、分散メモリ型並列処理にはクラスタシステム上で動くアプリケーションはまだ少ないという短所がある。

1.4 仮想並列計算機

膨大なデータを高速処理するには並列処理が有効だが、一般的に、並列計算機自体は非常に高価なものであり、維持コストも非常に高くなる。そのため、並列計算機を持つのはごく一部の大学や研究所そして企業しかなく、注目はされていたが利用しにくい状況であった。そこで、複数の計算機をネットワークで繋ぐことにより、計算機全体を 1 台の仮想的な並列計算機とする仮想並列計算機 (Parallel Virtual Computing) が注目されている。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。代表的な仮想並列計算機を構築するソフトウェアとしては、PVM(Parallel Virtual Machine)[3] や MPI(Message Passing Interface)[2]、OpenMP[5] などがある。

1.4.1 PVM(Parallel Virtual Machine)

PVM(Parallel Virtual Machine)[3] は並列計算を行うためのソフトウェアである。アメリカのオークリッジ国立研究所 [13] のメンバーが中心となって開発されたソフトで、Linux,Windows,BSD など様々の OS で動作し、入手方法が容易である。PVM をインストールすると、ネットワークに接続された複数台の計算機を単一の並列計算機として利用することが出来るようになる。

PVM に適した処理は以下の通りである。

- 内部計算付加に比べ、通信負荷が高くない処理
- 非常に負荷の高い問題を異機種共同で処理する場合
- 地理的に離れた計算機を使って処理する場合
- 分散処理を行う場合

1.4.2 MPI(Message Passing Interface)

MPI(Message Passing Interface)[2] は並列・分散プロセス間のメッセージ機能を提供する標準規格である。1995 年に MPI フォーラムによって標準化されて以来、多くの実装が存在しており、コードの移植性に優れている。自由に使用できる実装としては MPICH などがあり、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、プログラマが細かいチューニングが行えるのが利点である。

MPI に適した処理は以下の通りである。

- 内部計算負荷に比べ、通信負荷が高い処理をする場合
- 均一なプロセッサによる高速処理を行う場合
- リアルタイム処理を行う場合

1.4.3 OpenMP

OpenMP[4] 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[11] 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[12] などがある。

1.4.4 各並列計算の相違点

各並列計算の相違点を表 1 に示す。

表 1　各並列計算の相違点

	PVM	MPI	OpenMP
異機種間による仮想並列計算	可能	不可能	不可能
耐故障適合性	高い	低い	低い
メッセージ通信能力	低速	高速	低速

異機種間による仮想並列計算機はそれぞれの目的から性能が違ってくる。PVM は異種計算機ネットワークを目的として作成されているのに対し、MPI や OpenMP はメッセージ変換システムとして作成されている。次に、耐故障性は、PVM は仮想並列計算機にプロセッサを加えたり外したりすることができるが、MPI や OpenMP は基本的には出来ないため、PVM の方が高い能力を持っている。耐故障性とは障害発生時の被害を最小限度に抑える能力のことである。メッセージの通信能力は PVM は異機種間通信などサポートするため

のオーバヘッドがあるが、MPI は通信の高速化に重点が置かれているために、PVM に対して MPI の方が高速となっている。

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。

1.5 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 の最小全域木を求める問題である。頂点数 $|V| = n$、辺数 $|E| = m$ の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が得られる [1]。

1.6 本研究の目的

本研究では、近年重要度を増している仮想並列計算機の性能を検証する。仮想並列計算機を構築するソフトウェアには MPICH2 を使用する。本研究における検証方法としては、MPICH2 を用いて仮想並列環境を構築し、MPI 上で問題を解いたときに 1 台で処理した場合と複数台で処理した場合で比較して、どの程度、処理時間が短縮されたかを計測する。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。

1.7 本報告書の構成

本報告書の構成は以下の通りである。2 章では、計算機、MPICH2 など本研究を行うにあたって使用した環境状況を述べ、3 章では、本研究で扱った問題やその解法アルゴリズムについて述べる。4 章では、計測結果を述べ、5 章では、計測結果に基づいて考察を述べる。6 章では、本研究の結論、および今後の課題を述べる。

2 研究環境

2.1 MPI(Messe 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計算を利用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。MPI は複数の 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。MPI の自由に使用できる実装としては MPICH が有名である。他にも商用ベンダなどによる独自の実装が存在する。また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、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。

無料で提供されている MPI の主な実装として、MPICH2[6] や LAM[9]、OpenMPI[10] 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[6] で開発された。2005 年には MPICH の後継として MPICH2 が開発された。LAM はノートルダム大学の科学コンピュータ

表 2　本研究で使用した計算機一覧

	ホスト 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

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 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。コミュニティー，研究機関，パートナー企業によって開発，維持されているオープンソース MPI-2 を実装している。

2.2　使用計算機

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。本研究で使用した計算機の性能を表 2 に示し、図 1 に本研究で使用した仮想並列計算機の構成図を示す。また、本研究で使用する計算機は、OS として Windows 系 OS を用いる。Windows 系の OS するのは他社の OS より普及率が高いからである。

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。MPICH2 を使用するためには、MPI を構築する全ての計算機に MPICH2 をインストール (Install) する必要がある。MPICH2 はインターネット上の 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、、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。本研究では 2010 年 12 月現在の Windows 用 MPICH2 の最新の版である mpich2-1.0.6p1-win32-ia32.msi を各計算機にダウンロードし、C:￥Program Files￥backslash MPICH2 の下にインストールを行った。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に対して環境変数の PATH(”C:￥Program File￥MPICH2￥bin”) 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新規でコマンドプロンプト上で”mpiexec” とコマンド実行させたときに、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。また、Windows 上で MPI を使用する場合、MPI を構築する全ての計算機に管理者権限を持つ同名のアカウントを予め設定し、各計算機のアカウントには同一のパスワードを設定する必要がある。また、各計算機上に共有フォルダを作成しておくことで、実行ファイルの受け渡しを円滑に行うことができる。

2.4　Visual C++2008Express Edition の導入

本研究で作成する並列処理を行う MPI プログラムは、C++ 言語を用いる。C++ 言語のコンパイルには Microsoft 社製の Visual C++2008Express Edition [8] 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

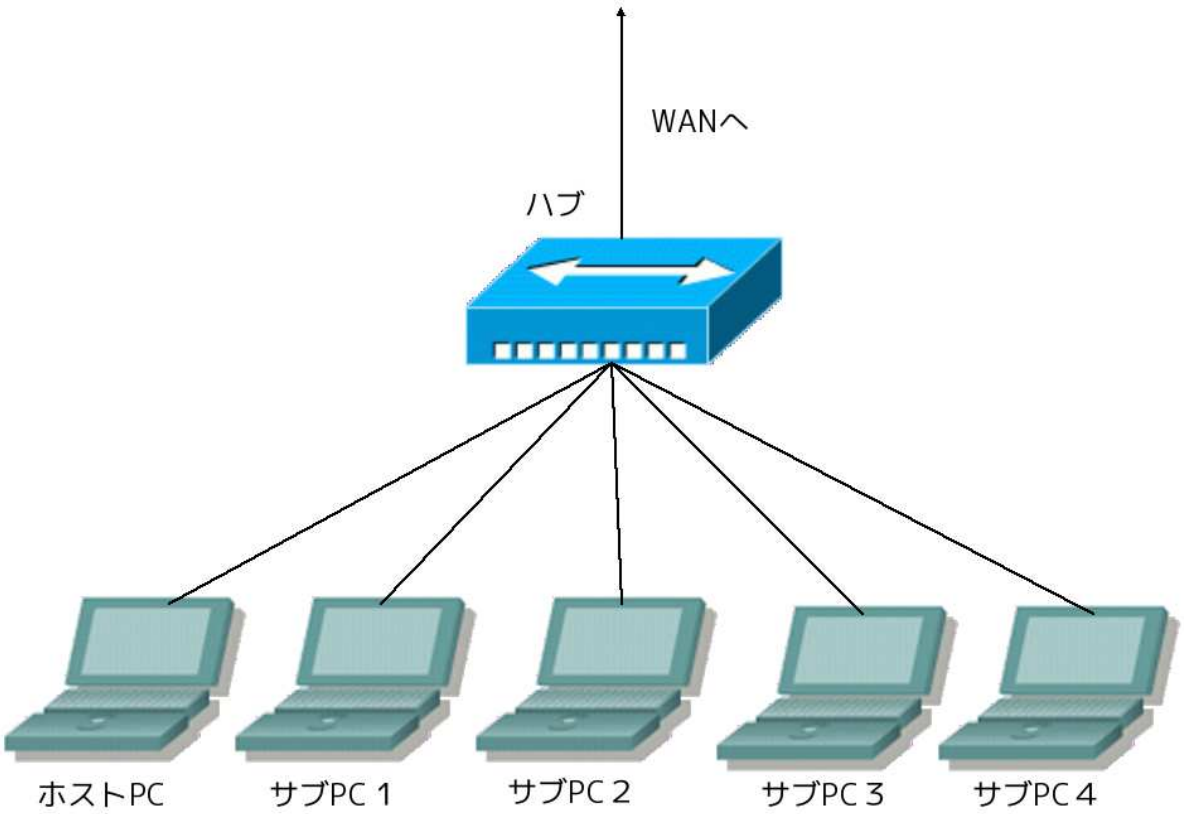

図 1　仮想並列計算機の構成図

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を追加する、追加する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。

3　最小全域木問題と解法アルゴリズム

3.1　重み付無向グラフ

あるグラフ $G = (V, E)$ において、G の全ての辺 $(u, v) \in E$ $(u, v \in V)$ に対して辺 (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、無向グラフとは、各枝の始点と終点がどちらであるかを気にしないグラフもある。 このようなとき、平面上の幾何学的表現では各枝を表現する矢線から矢印を取って、そのグラフを表現する。また、グラフ G において、頂点や辺に実数等が割り当てられる場合、重み付きグラフ (weighted graph) またはラベル付きグラフ (labeled graph) と呼ぶ。重みは、実数の場合もあり、また文字や文字列であることもある。上記の無向グラフと重み付グラフの両方の性質を持ったものが重み付無向グラフである。グラフ G が重み付無向グラフのとき、G の全ての辺 $(u, v) \in E$ $(u, v \in V)$ に対して $w(u, v) = w(v, u)$ となる。ただし、$w(u, v)$ は辺 (u, v) の重みである。

3.2　最小全域木問題

最小全域木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 G の部分木のうち、辺の重みの総和が最小なグラフである。また、最小全域木問題とは重み付無向グラフ G が与えられたとき、G の最小全域木を求める問題である。

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] 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 $O(m + nlogn)$ 、Kruskal のアルゴリズムは $O(mlogn)$、Solin 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 [1] を用い、MPI 上でプログラム化した。

以下に Sollin のアルゴリズムとその計算量について述べる。

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。この計算量は定数個の代入のため定数時間である。

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

step 2-1:　配列変数 k に 1 を加える。この計算量は定数個の加算なので定数時間である。

step 2-2:　各頂点 $v \in V_k$ において、v に隣接する辺 (v, u) $(u \in V_k)$ の中最も小さい辺 $\{(v, m) | w(v, m) \leq w(v, u)$ $(u \in V_k)\}$ 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。計算量は大小比較を行ない、比較する辺を半分にするという作業を $\log n$ 回繰り返すので $\frac{n^2}{\log^2 n}$ プロセッサで $O(\log n)$ となる。

step 2-3: 各頂点 $v \in V_k$ において、$r[v]$ の根となる頂点 $r[v]$ を探す。この処理はポインタジャンピングを繰り返すこととでき、1 回のポインタジャンプにより根までの距離が半分になる。したがってこの処理を $\log n$ 回繰り返すのでこの計算量は $\frac{n^2}{\log^2 n}$ プロセッサで $O(\log n)$ となる。

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

step 3: 作業用リスト L_i $(0 \leq i < k)$ から解行列 B を作成する。計算量は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の検証を行う。付録 A に本研究で作成した MPI プログラムを示す。

本研究で作成した MPI プログラムは、計算機のうち 1 台をホスト PC として用いる。ホスト PC は入力となる重み付無向グラフ 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。サブ PC は送信されたデータに対して Sollin のアルゴリズムに基づき処理を行い、その処理結果をホスト PC に返送する。ホスト PC は返送された処理内容からグラフを再構成する。

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 top として保持する。

step 0-2: top をホスト PC からサブ PC に送信する。このとき、サブ PC 全部に top と頂点に関するデータを送信する。を送信する。

step 1-1: ホスト PC から送られたデータから自分が担当する頂点を決める。これは for ループの for(int tantou = (size*myrank)/numprocs ; tantou ¡ (size*(myrank+1))/numprocs ;tantou++) から決まる。(tantou = 自身が担当する頂点 ,size = 頂点数, myrank = 自身のプロセッサ番号, numprocs = プロセッサ数,) そこから最も辺の重みが小さいものを選び保存する。

step 1-2: 保存した頂点最小の重みデータをホスト PC にデータを返送する。

step 1-3: 返送されたデータから隣接行列 $answer$ の論理和をとる。

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

step 2-2: 担当している頂点の親の親に付け変える。

step 2-3: 選ばれた辺は次回から選ばれないようにする。

step 2-4: 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 2-5: 親が自身でなくなったとき、その頂点を殺し、以後頂点は使用しない。

step 2-6 それぞれのデータを集め、もう一度ホスト PC に送りなおす。

step 3-1: 全てのデータを集める。

step1.2 を logn 回繰り返す。

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 台計算機を用いて最小全域木を求めた場合の計算時間の測定を行う。

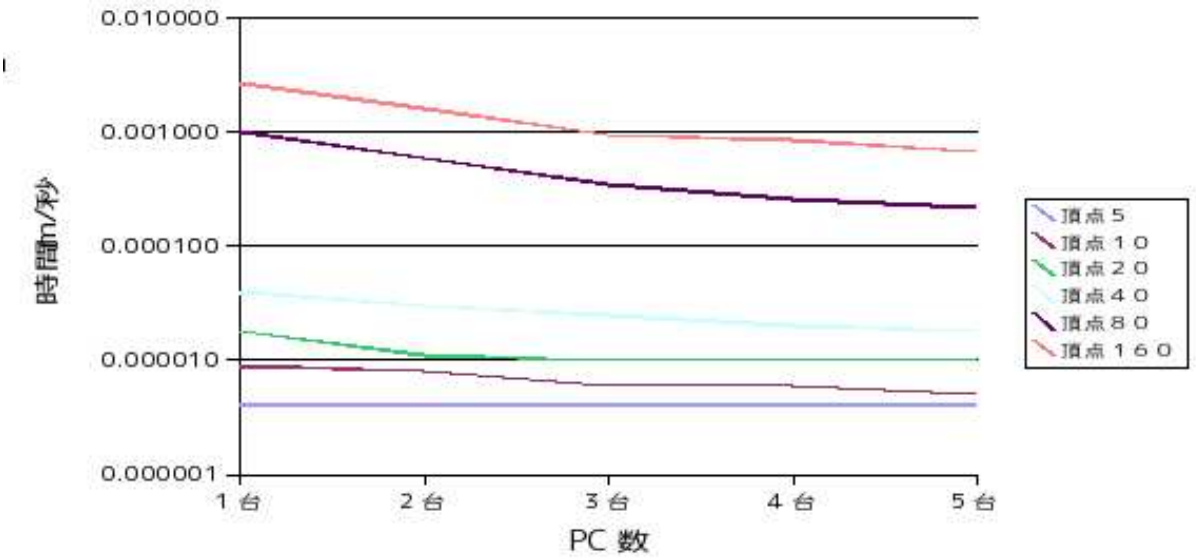

図 2　内部処理時間と計算機数の関係

4　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。

4.1　内部処理時間

MPI 上で計算機 1〜5 台を用いて最小全域木問題を解いた場合の通信を含まない内部処理時間を図 2 に示す。図 2 より、処理における内部処理時間は連結するサブ PC の増加とともに速くなっていくことが示される。

4.2　全体の処理時間

MPI 上で計算機 1〜5 台を用いて最小全域木問題を解いた場合の全体の処理時間を図 3 に示す。3 の結果より、内部処理時間とは逆に連結するサブ PC が増えると全体の処理時間は増加していくことが示される。

4.3　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、

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

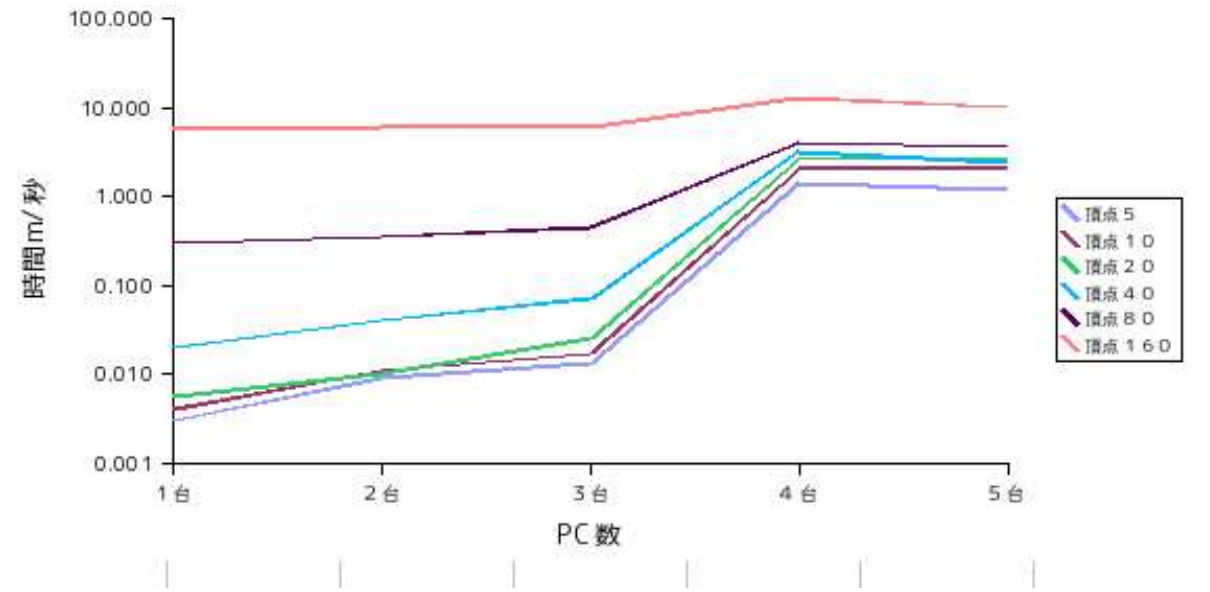

図 3　全体の処理時間と計算機数の関係

と置き、図 2 の値から連立方程式を立てる。この連立方程式より、

$$T_{comp}(n,p) = \frac{1.2n^2 * 10^{-6} - 7.6n * 10^{-6} + 2.3 * 10^{-6}}{p} + 1.86 * 10^{-6}\log^2 + 2.06\log n * 10^{-6} + 3.89 * 10^{-6} \quad (2)$$

が得られる。

5　考察

本研究で得られた計測結果では、図 2 に示される通りサブ PC を増やした場合の処理速度の向上が確認できた。これにより、内部処理時間については、MPI を用いて並列処理を行えば処理時間の向上可能なことが示される。しかし全体で見ると図 3 に示される通りサブ PC の増加に伴い処理時間の低下が起こった。これは各サブ PC からのデータ通信時間が多きく影響していると思われる。また図 3 は、でサブ PC が 3 台に増えたときに処理時間の急激な増加が起こった。これは複数の PC を繋げたことにより LAN に負荷がかかったため、もしくは 3 台目に MPI ネットワークに加えた計算機であるサブ PC3、1 台だけ OS の種類が異なることから来た何らかの弊害からによるものだと推測される。その原因を確認するために、MPI ネットワークに計算機を加える順序を入れ換えて計測を行った。計測の結果、サブ PC の台数に関係無く、サブ PC3 を MPI ネットワークに加えたときに処理時間の増加が起こったことから、処理時間の増加は後者の原因の要素が大きいと思われる。

6　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。しかし本研究では、MPI による並列化によって内部処理時間は向上には成功した。しかし、計算機の増加に伴う各 PC 間での通信時間により、全体の処理時間は計算機数を増やした場合、予想に反して時間がかかっており、MPI の有用性が充分示せたとは言えない。また今回の研究では使用した計算機はどれも類似機種による調査なので、仮想並列計算機の構成の中でスペックの低い計算機がある場合などは考慮していない。

これらの点から、ネットワーク構築時の接続方法やプログラムのアルゴリズムを通信を考慮した BSP[14] や CGM[15] のアルゴリズムを利用し改善することで全体の処理時間の向上を得ることや、異なるスペックの計算機で MPI 構築することによりおこる処理速度の違いやそれに伴う弊害を調査することが今後の課題である。

謝辞

本研究をするにあたり、石水隆先生には多忙のなか御指導頂本当にありがとうございました。また本研究を共に行ってくれた情報論理工学研究室の皆様にもとても感謝いたします。

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) PVM(URL),http://www.csm.ornl.gov/pvm/
4) OpenMP(URL),http://www.openmx-square.org/
5) OpenMX(URL),http://www.openmx-square.org/workshop/workshop10/index.html
6) MPICH2(URL),http://www.mcs.anl.gov/research/projects/mpich2/.
7) MPICH2 on Windows
   Local(URL),http://ums.futene.net/wiki/Paralell/4D5049434832206F6E2057696E646F7773204C6F63616C.html
   .
8) Microsoft Visual Studio Express(URL),http://www.microsoft.com/japan/msdn/vstudio/express/
9) LAM/MPI Parallel Computing(URL),http://www.lam-mpi.org/
10) Open MPI(URL),http://www.open-mpi.org/
11) OpenMosix, http://theochem.chem.nagoya-u.ac.jp/wiki/wiki.cgi/ClusterBuild?page=OpenMosix
12) Score, http://ja.wikipedia.org/wiki/SCore
13) オークリッジ国立研究所 (URL),http://www.ornl.gov/
14) L.G.Valiant, A Bridging Model for Parallel Computation, Comm. of the ACM, Vol.33, No.8, pp.103–
    111, (1990).
15) 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　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。

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);//  ホスト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]
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);//ホ スト
PC以外のPCがホストPCにデータを送る


    }
```

```
    St1finish = MPI_Wtime();
    st1 += (St1finish - St1start);
    if(myrank == 0){ //サブPCのデータをホスト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();


}
```