卒業研究報告書

題 目

MPI による並列計算

指 導 教 員

石 水　研 助教

報 告 者

07–1–037–0138

穂 積　剛 文

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、PC の高性能化やネットワークシステムの発達により、日常のネットワークにおいても取り扱われる情報の量は日々増大しており、その情報処理時間を短縮することは計算機を使用する上での重大な課題である。高速な処理を行う方法の一つに、一つの処理に対して複数のプロセッサを用いて協調して処理を行う並列計算を使うことが挙げられている。しかしながら並列計算機は高価でなかなか利用することが難しい、そこでネットワーク上接続した複数の計算機郡全体を一台の仮想的な並列計算として用いる仮想並列計算 (Parallel Virtual Computing) のシステムの採用が考えられる。本研究では、無料提供されている仮想並列計算環境を構築するソフトウェアのひとつである MPI(Message Passing Interface)[2] を用いて最小全域木問題を解き、逐次処理した場合並列計算した場合どの程度処理時間が短縮できるかを計算し評価する。

目次

1 序論

1.1 **並列処理** (Parallel Processing)

ある 1 つの処理を、複数のプロセッサが協調してデータを処理することにより、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い複雑な計算処理を可能とする処理を並列処理 (Parallel Processing) という。計算処理においては計算素子が高度に集積化され、極めて高速に処理を行うことができる。しかし、素子の集積化による高速化はいずれ限界に達すると考えられる。一方、並列処理にはそのような限界は本質的に存在しない。このため、並列処理に対して大きな期待を持つことが出来る。並列計算機の歴史は、VLSI 技術の発達によって実現されている要素プロセッサ個数の増大過程でもある。数大規模のミニコンなどをチャネル結合した時代を源として、マイクロプロセッサの普及とともに、並列計算機として集積できる要素プロセッサ個数も飛躍的に増加している。現在では、104 から 105 個要素プロセッサ規模の超並列計算機も出現している。複数のプロセッサが協調してデータを処理することにより、問題を短時間で解け、またより複雑な問題を解くことができるようになる。現在の並列計算機は地球規模の気象シミュレーションや天体の軌道計算など、計算量の大きな問題を短時間で解く必要のある分野は多岐に渡っている。

1.2 **仮想並列計算機** (Parallel Virtual Computer)

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もとめられている。しかしながら高性能な並列計算機は、非常に高価で個人の使用は難しい。そこで本研究では、複数の計算機をネットワーク接続して接続された計算機全体を仮想計算機 (Virtual Machine) として用いてクラスタ処理による仮想並列計算機を構築する。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[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

1.3 MPI(Message Passing Interface)

MPI(Message Passing Interface)[2] は、メモリ分散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 は新しいプログラミング言語ではなく、 C または Fortran から呼び出すサブプログラムのライブラリである。 MPI は MPI Forum という国際フォーラムで標準化されており、現在 MPI 2.0 が公開されている。またソフトウェアでなく MPI 標準というインターフェイスの規格であり、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を用いることで、MPI を使用するユーザーは、通信を考慮せずプログラムを組むことが出来る。MPI2 では動的なプロセス管理の機能が取り入れられた。動的なプロセス管理とは、アプリケーションの中から動的にプロセスを生成したり、停止したりすることである。この機能は、処理中に必要に応じてプロセス数を調整できるため、効率の良い並列計算には欠かせないものである。

1.4 PVM(Parallel Virtual Machine)

PVM(Parallel Virtual Machine)[3] は 1991 年にアメリカのオークリッジ国立研究所 (Oak Ride National Laboratory) のメンバーが中心となって開発された、並列計算を行うためのソフトウェアである。動作するマシンの種類が多いこと（LinuxBSDWindows のどの OS でも動作可能）や 入手方法が容易である為研究機関などで広く利用されている PVM ソフトウェアの構成は大きく二つにわけられる。一つは計算機上にデーモンと呼ばれるもので、デーモンを存在させ、このデーモンを用いて各計算機間の通信を行うことによりデーモンで接続された複数の計算機を一つの仮想並列計算機として構成させる。そしてユーザは PVM アプリケーションを一人で複数実行することも可能である。もう一つがスルーチンライブラリで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。かつては分散メモリ環境は PVM が主流であったが、現在は MPI に押されているようで、web 上で情報を検索しても MPI に比べて圧倒的に情報は少ない。もともとお金を持ち合わせていない人があまったマシンをかき集めてきて大量の計算をさせようとしたのが発祥と言われており、様々なアーキテクチャを混在させて使うことも可能である。

1.5 OpenMP

OpenMP[6] は複数の CPU（複数コアを含む）を持った計算機環境を利用するために用いられる標準化された基板で、主に共有メモリ型並列計算機で用いられる。OpenMP を使う最大の利点は、OpenMP に対応したコンパイラであれば、非常に簡単に並列化できる点である。現在、gcc、Visual C++、および Intel コンパイラなど主要なコンパイラは OpenMP に対応している。習得も他の並列か技法に比べて比較的容易である。なお、速度を最優先にする場合、単一コンピュータ上で動かした場合でも、OpenMP より MPI の方が効率のよいことが多い。

1.6 openMosix

openMosix は、負荷を均等に分散させるために、クラスター内の別のマシン間でプロセスを透過的に移動させるよう、カーネルを拡張したものである。openMosix ソフトウェアは、カーネルパッチとサポートツールを含んでいる。カーネルパッチはマシン間でプロセスを透過的に移動させるためのものである。プロセスの移動はユーザにとって完全に透過的である。クラスター内のマシンで数十個のファイルを圧縮する場合。圧縮を始めて数秒後 2、openMosix は、負荷が高いマシンからクラスター内の別マシンに一部のプロセスを移動させる。クラスター内のマシンの数によって、最も効率のよい負荷分散を行なうのである。

1.7 Score

SCore（エスコアー）とは、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 (Real World Computing Partnership, RWCP) 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

1.8 MPICH

MPICH は、アメリカのオ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プログラムのソースコードを変更することがなく、分散メモリ環境、共有メモリ環境のマシンで動作させることが可能である。この MPICH は UNIX や Linux に限らず、Windows 系へのサポートもしており、OS への対応が充実している。本研究では、MPICH の最新のヴァージョンである MPICH2[4] を使用し並列計算による計算速度向上効果を検証する。

1.9 本研究の目的

本研究では、MPI を用いた並列計算を行いその有効性を検証する。検証方法としては、MPI を用いて並列処理した場合と逐次処理した場合ではどの程度処理時間が短縮できるかを検証する。

1.10 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、与えられた重み付き無向グラフ（ネットワーク）の部分グラフとなる全域木の中で、辺の重みの和が最小のになるような木を見付ける問題である。最小全域木問題に対して、n は入力グラフの頂点の個数とし m が入力グラフの辺の本数をした時 Prim は O(m+nlogn)、Kruskal は O(mlogn)、Sollin は $O(n^2)$ の逐次アルゴリズムを提案した。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

1.11 本報告書の構成

本報告書の構成を以下に述べる。2 章では研究の環境を揃える準備、3 章では研究方法、4 章では研究の考察を述べ、5 章で理論値と比較した後 6 章で結論を述べる。

2 準備

2.1 MPI

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、多くの実装が存在している、また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。

無料で提供されてい MPI の主な実装として、MPICH2[4] や LAM[7]、OpenMPI[6] 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[4] で開発された。2005 年には MPICH の後継として MPICH2 が開発された。本研究で 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。

LAM は（Local Area Machine）はシステムが様々なプログラム開発やデバックを支援するツールとともに、簡単な実行制御機能があり、ＬAMを用いることによって分散型並列計算機が１つの問題を解く１つの並列マシンのように振る舞うことができる。また、ＭＰＩ－２における１方向通信、動的プロセス管理に関する機能もカバーされている。

OpenMPI は並列コンピューティング環境を利用するために用いられる標準化された基盤。OpenMP は主に共有メモリ型並列計算機で用いられる。

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

使用する PC のスペック 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計算機を 4 台使用する。それぞれのコンピュータは 100Base-TX、イーサネットケーブルとハブにより計算機環境が構築されている。本研究では、1 台をホストコンピュータとして、残り 3 台をサブコンピュータとして扱う。表 1 に、そのコンピュータの性能を示す。本研究では表??に示す計算機に対し MPI 環境の構築を行った。図 1 に本研究で構築した仮想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 の中でな最も利用率の高い WindowsOS を利用する。

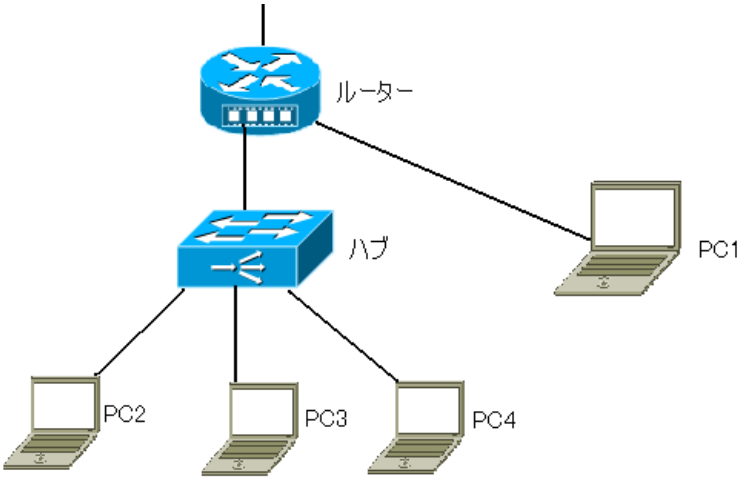

図 1　仮想並列計算機の構成

使用する PC のスペック

	1 台目	2 台目	3 台目	4 台目
OS	Intel core 2DuO	Intel core 2DuO	Intel core 2Duo	Intel core 2Duo
CPU	1.40GHz	1.40GHz	1.4GHz	1.4GHz
Memory	1.00GB	1.00GB	1.00GB	1.00GB

2.3 MPICH のインストールと環境設定

仮想並列計算環境を作るために、MPICH のホームページ [4] からの Windows 用の MPICH2 がダウンロードする。2010 年 12 月時点では、mpich2-1.3.1p1-win32-ia32.msi が最新でありこれをインストールする。方法は、Windows Vista の場合は, スタート¿すべてのプログラム¿アクセサリからコマンドプロンプト を右クリックし，管理者として実行 をクリックする．ユーザ アカウント制御 のウインドウが表示されるので，はい をクリックする．管理者： コマンド プロンプト が開くので，cd コマンドで，MPICH2 をダウンロードしたフォルダへ移動し，MPICH2 のインストーラ パッケージの名前 mpich2-1.3.1p1-win32-ia32.msi を入力し Enter キーを押し実行をクリックする。画面が表示され Next を続けてクリック認証画面では I agree をチェックし、Next をクリックする。。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、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に、環境変数を用いて MPICH プログラムのあるフォルダへの PATH を通す必要がある。この方法は、Windows 系 OS の場合はマイコンピュータのプロパティから行うことができる。プロパティで表示されるシステム環境変数のリストから、変数名 path を選択し、変数値の最後に C:\ Program Files\ MPICH2\ bin\ を追加すればよい。PATH の指定が正しくできているかの確認は、新規にコマンドプロンプトを立ち上げ、mpiexec と命令を実行したときに、引数の入力を促す Usage メッセージが表示されるかどうかで判別できる。Visual Studio2008 がすでに起動している場合は,Visual Studio2008 を再び立ち上げないと設定が反映されないので注意が必要である。また、Windows の計算機を使用する場合に、使用する計算機全てに管理者権限を持つ同一のアカウント、パスワード共通のアカウント名とパスワードを設定しておき共通したファイルを構成することで実行ファイルの受け渡しをスムーズに行うことが出来る。

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 Edition が、マイクロソフトの公式ページ [5] より配布されているので、そのデータをダウンロードしインストールを行うことができる。これを用いて MPICH2 による並列プログラミングを行うには以下の設定をする必要がある。まず Visual C++2008 を起動し、起動画面の上部にあるツールバーから [ツール]、[オプション] を選択し、オプションダイアログを開く。オプションダイアログで、ツリービューの [プロジェクトおよびソリューション] を開く。その後、[VC++ ディレクトリを表示するプロジェクト] からインクルードファイルに MPICH2 のインクルードファイルを追加する。同様に、ライブラリファイルも追加する。初期設定だと、追加するフォルダはそれぞれ C:Program Files\MPICH2\include C:Program Files\MPICH2\lib となっているので環境に合わせて適切

なフォルダを選択する。また、追加する依存ファイルとして、デバッグの場合 ”mpi.lib cxxd.lib ”と入力しリリースの場合 ” mpi.lib cxx.lib ” と入力する。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。また Windows 上で MPI のプログラムを実行する場合、管理者権限が必要なので注意が必要である。

2.5 ワークグループの設定

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、それぞれのコンピュータに mpi アカウントを作りパスワードを共通のものとし、ワークグループを「ISHIMIZU」として統一する。これにより、コンピュータ同士でのネットワークが構築される。また、プロセスの実行の際には、すべての使用コンピュータ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コンピュータに準備する。

3 方法

3.1 目的

本研究では,4 台の家庭用の計算機をネットワーク接続し MPI 環境を作る。また、MPI の有効性を検証するための問題としては最小全域木問題を用いる。

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、G の部分グラフとなる全域木の中で、辺の重みの和が最小になるような木 T を見付ける問題である。また、グラフ G の頂点や辺に実数等が割り当てられるとき、G を重み付きグラフ (weighted graph) またはラベル付きグラフ (labeled graph) と呼ぶ。付加された重みにより頂点をに名称を定義したり、道に長さやコストなどを与えることができる様になる。また、グラフ G=(V.E) において、全ての辺 $(u, v) \in E$ $(u, v \in V)$ について辺 (v,u) が存在するとき、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である. 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. 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) 時間 Kruskal のアルゴリズムは O(mlogn) 時間、Sollin のアルゴリズムは (On^2) 時間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 $O(\frac{n^2}{p} + \log^2 n)$ 時間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 10,20,30,40,80,160 の重み付無向グラフが与えられたとき、MPI 上でその最小全域木を求める。本研究で作成した最小全域木問題を求める MPI プログラムは Sollin's Algorithm[1] を元にしている。

3.3 Sollin's algorithm

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

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。計算量は定数個の代入のため定数時間となる。
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。計算量は定数個の加算なので定数時間となる。
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ので $\log n$ 回繰り返すこので $\frac{n^2}{\log^2 n}$ プロセッサで $O(logn)$ 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。根となる頂点の探索はポインタジャンプを繰り返すことにより行える。一回のポインタジャンプで根までの距離が半分になるので $\log n$ 回繰り返すこので根に達成できる。したがって計算量は $\frac{n^2}{\log^2 n}$ プロセッサで $O(logn)$ となる。
step 2-4 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 $O(logn)$ 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。step2 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、最小全域問題を解く並列アルゴリズムとして、Sollin's Algorithm を用い、MPI 上でプログラム化した。以下に Sollin's Algorithm を示す。[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 $ansewr$。$answer$ はメイン PC に保持される。
step 0-1: ホスト PC 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: 隣接行列 $hairetu$ をメイン PC からサブ PC に送信する。
step 1: 自分が処理するデータをホスト PC から送られたデータから決め、送られたデータの中で最も辺の重みが小さいものを選びホスト 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を親としホスト PC にデータを送信する。

表 1　内部計算時間と計算機数の関係

台数 \ 頂点数	頂点 10	頂点 20	頂点 30	頂点 40	頂点 80	頂点 160
1	0.000010	0.000017	0.000027	0.000038	0.00095	0.00258
2	0.000008	0.000012	0.000022	0.000031	0.0007	0.0018
3	0.000006	0.000009	0.000015	0.000024	0.00034	0.0004
4	0.000005	0.000009	0.000014	0.000021	0.00025	0.00074

(m 秒)

step 3:　$hairetu$ を更新し、重みが最も小さい辺を繋げる。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する。

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、メイン PC が各サブ PC にデータを送信し、サブ PC が送られてきたデータを元に計算した結果をメイン PC へ戻す。

また以下には本研究で作成した MPI プログラムを載せる。

4　結果・考察

頂点数 10,20,30,40,80,160 のグラフに対して 1〜4 台の計算機を用いて MPI 上で最小全域木を求めたときに、その内部計算にかかった時間を表 1 に示す。また、通信時間を含む全体の処理時間を表 2 に示す。表 1 より計算機の数が大きいほど内部計算時間が減っていることが示されている。しかしながら表 2 より通信時間を含むプログラムの全体処理の時間は、計算機台数増加に伴い増えていることが示されている。計算機数の増加により内部計算時間が減少しているにもかかわらず、全体処理時間が増加していることは、全体の計算処理において通信時間の割合が大きいということが示されていると言える。

4.1　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、

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

と置き、表 1 の値から連立方程式を立てる。この連立方程式より、

$$T_{comp}(n,p) = 0.66\frac{2.83 * 10^{-7} * n^2 + 11 * 10^{-4} + 1.2 * 10^{-4}}{p} + 2.12{*}10^{-4}\log^2 n + 7.41{*}10^{-4}\log n + 7{*}10^{-4} \tag{2}$$

が得られる。

表 2　全体の処理時間と計算機数の関係

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

(m 秒)

5　結論

本研究では MPI による並列計算の実用性を求めるために MPI を用いて最小全域問題を解く時間を検証した。しかしながら内部計算の計算速度を早めることができたのだが、重要な全体処理時間短縮を各 PC 間での通信時間の増加によって、十分にすることができず、MPI の実用性を充分に示せたことにはならない。この事より、実用性のある MPI による並列計算をする為には通信時間短縮を考えなければならないことがわかる。その為には通信環境を良くしたりサーバ間の通信速度に負荷の掛からないプログラムを作ることが必要であり今後の課題と考えられる。

謝辞

本研究におきまして多忙の中様々な助言を頂いた石水隆先生に感謝の意を表します。ご迷惑も沢山お掛けしましたが一年間本当にありがとうございました。また、一緒に研究をした太田君にも感謝いたします。

参考文献

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

[2] P パチェコ 著、秋葉博 訳:MPI 並列プログラムミング、培風館 (2001)

[3] PVM 公式ホームページ,http://www.csm.ornl.gov/pvm/

[4] http://www.mcs.anl.gov/research/projects/mpich2/：MPICH2 公式ホームページ

[5] http://www.microsoft.com/japan/msdn/vstudio/2008/product/express/ :microsoft の公式ホームページ

[6] http://openmp.org/wp/ :OMP 公式ホームページ

[7] http://www.lam-mpi.org/ LAM のホームページ

[8] http://openmosix.sourceforge.net/ OpenMP の HP

[9] http://www.pccluster.org/ score の HP

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 4 //頂点の数を表す。


int alive[SIZE];    //頂点が生きているかを判定する。
int hairetu[SIZE][SIZE]; //辺の重みの配列
int changex[SIZE][SIZE];
int changey[SIZE][SIZE];//配列
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);
        }
    }
}

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;
}

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");
      }
}

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();
}



\addcontentsline{toc}{section}{\bibname}
\bibliography{reference-sjis}

\bibliographystyle{jplain}

\newpage
\appendix
```