卒業研究報告書

題 目

MPI を用いたグラフの並列計算処理

指 導 教 員

石 水　隆 助教

報 告 者

07–1–037–0198

川 崎　直 紀

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、様々な分野での情報が計算機で取り扱われている。そのため、CPU の処理性能は著しい増加の傾向にあるが、それに伴い計算機が扱うデータ量が膨大なものになり、計算時間が長くかかる大規模な処理も多くなっている。この問題の解決のために、情報処理の高速化が追求されている。 高速な処理を行うには、1 台の計算機の処理速度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 (Parallel Computer) を用いて並列計算を行う方法の 2 つがある。1 台の計算機の処理速度の向上には物理的な限界があるため、並列処理による高速化が注目されている。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することはできない。そこで、複数の計算機をネットワーク接続して計算機群全体を 1 台の仮想的な並列計算機とする手法が注目されている。

本研究では、仮想並列計算環境を構築するソフトウェアの 1 つである MPI(Message Passing Interface) を用いて仮想並列計算を行い、その実用性を検証する。MPI は無料提供されているソフトウェアであり、MPICH2 のページからダウンロードすることにより容易に使用することが可能である。現在では世界標準とされている分散メモリ型並列処理におけるメッセージ交換のためのライブラリとして使用されている。評価方法は 1 台の計算機を用いて逐次処理した場合と MPI を用いて、複数の計算機で並列処理した場合の所用時間を比べ、処理速度がどの程度向上したかを測定する。

目次

1 **序論** 1

1.1 本研究の背景 1

1.2 本研究の目的 2

1.3 本報告書の構成 3

2 **研究内容** 4

2.1 MPI (Message Passing Interface) 4

2.2 計算機環境 4

2.3 MPICH2 のインストールと環境設定 5

2.4 Visual C++ 2008 Express Edition のインストール 6

2.5 最小全域木 6

2.6 最小全域木問題を解く MPI プログラム 7

3 **結果** 9

3.1 内部計算時間 9

3.2 全体の処理時間 9

3.3 理論値との比較 10

4 **考察** 11

5 **結論・今後の課題** 12

謝辞 13

参考文献 14

付録 A 最小全域木問題を解く MPI プログラム 15

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、処理時間を短縮にすることができ、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。しかし一方短所としては、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。

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、プログラマが細密なチューニングを行えるというメリットがある一方、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ならないといったことが挙げられる。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 MPICH2[4] や LAM[9] といったものがある。

PVM(Parallel Virtual Machine)[8] はネットワークに接続された複数台の計算機を仮想的に 1 台の並列計

算機として使い，大規模な演算ができるようにするためのミドルウエアである。1980 年頃，米テネシー大学のジャック・ドンガラ教授らが，演算性能が低い計算機環境でも大規模な数値計算が実行できるようにすることを目的に開発した。現在はパブリック・ドメイン・ソフトとして無償で配布されているが，米 IBM，米ヒューレット・パッカード，日立製作所など UNIX 機ベンダが自社機用に最適化した PVM を販売している。

OpenMP[3] 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。現在 FORTRAN と C/C++ について標準化が行われている。

OpenMosix[11] は、Linux のプロセスをネットワーク経由で他のクラスタノードに実行させる migration と呼ばれる仕組みである。動的な負荷分散を自動的に行ってくれるので、複数の計算機をあたかも一台の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（エスコアー）[12] 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。

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[4] 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。

1.1.4 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。

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、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、MPICH2 を用いて MPI 環境を構築し、性能検証用問題に対して 1 台の計算機を用いて逐次処理した場合と MPI を用いて、複数の計算機で並列処理した場合の所用時間を比べ、どの程度処理時間が短縮できるかを計測する。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。

1.3 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、本研究の環境および MPI の導入方法と使用機器ついて述べる。3 章では、並列計算による処理時間に関する結果を示し、次の 4 章においてその考察を行う。5 章では結論と今後の課題について述べる

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、メッセージ通信の API 仕様である。そのため、PVM[8] と違いソフトウェアがあるというわけではない。MPI は 1992 年に結成された MPI Forum 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、様々な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 PVM と同等の機能を持っており、PVM の問題点の改善も含まれている。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。大きく PVM と違う点は、異機種間の通信が考慮されていないことである、MPI を用いての仮想計算機の構築には使用する計算機のオペレーティングシステム (Operating System) が同じでなければならないという制約が存在する。MPI は PVM と同様に、パーソナルコンピュータからスーパー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[4] や LAM[9] といったものがある。また、MPI のサポートするプログラミング言語は多く、C 言語から C++、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 MPI の主な実装として、MPICH2[4] 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[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。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。完全に新しい MPI-2 に準拠した実装では、Open MPI は、システムおよびソフトウェアのベンダ、アプリケーション開発者、およびコンピュータサイエンスの研究者に利点を提供する。OpenMPI は、簡単に使えて、多種多様な OS やネットワーク接続、バッチ、スケジューリングシステム上でネイディブに動作する。

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。

MPICH2 を使用する理由は、MPI の特徴である移植性の高さ、MPI の新規格である MPI-2 をある程度サポートしている点や、Windows OS にも導入が可能であり現在でも研究開発が盛んに行なわれているためである。

2.2 計算機環境

本研究では、OS として Microsoft 社が提供販売を行なっている OS である Windows 系 OS を使用する。Windows 系 OS を使用する理由としては、Windows 系 OS は現在の一般家庭や教育施設、そして企業などの場所において計算機に取り入れられており、Windows は他社の OS よりも圧倒的に幅広く使用されているためである。

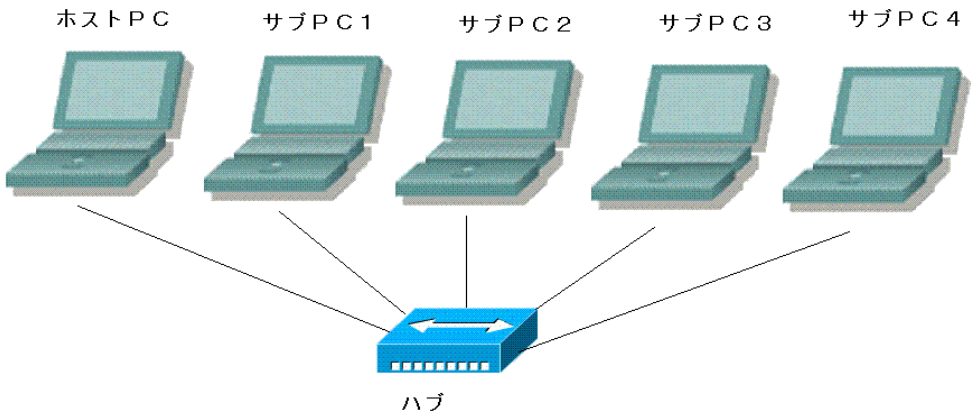

図 1　使用した計算機ネットワークの概念図

表 1　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

また本研究では、同一機種の計算機 5 台を 100Base-TX により LAN 接続し MPI 環境を構築する。本研究では、1 台をホスト PC として、残り 4 台をサブ PC として扱う。図 1 に本研究で使用した計算機ネットワークの構成図を示す。また、表 1 に、本研究で使用する計算機の性能を示す。

2.3　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 を使用するには、MPI を構築する全ての計算機に MPICH2 をイントールする必要がある。Windows 用 MPICH2 の実行形式のインストール用ファイルを公式サイト [4] よりダウンロードし、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:\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、インストーラの実行時に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 フォルダ (本研究では”C:¥Program Files¥MPICH2¥bin”) に環境変数でパス (PATH) 指定をする。また、Windows OS の場合には全ての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管理者権限のあるユーザを設定しておく必要がある。さらに、MPI プログラムの実行には全ての計算機に共通したファイル構成で、MPI プログラムの実行に必要なファイルを置く必要がある。そのため、共有するフォルダを作成することで実行ファイルの受け渡しが迅速に行なえるようになる。ただし、全ての計算機のファイヤーウォールを無効にしなければフォルダの共有ができないので、並列計算実行時はファイヤーウォールを無効にする必要がある。また、使用する OS が Windows Vista の場合は、bin フォルダにある smpd.exe と mpiexec.exe のプロパティを管理者権限で実行するように設定しなければならない。

MPI プログラムを実行するにはまず、Visual C++ を開き、MPI プログラムのファイルを選択して開き、ビルドする。すると、フォルダの中に拡張子.exe の実行形式ファイルが作成されるので、これを各計算機の同じ位置のフォルダに入れておく。そして、計算を実行するにはホスト PC のコマンドプロンプトを開き、mpiexec コマンドで使用するサブ PC と exe ファイルを入力することで、MPI プログラムを実行できる。

2.4 Visual C++ 2008 Express Edition のインストール

本研究で作成する MPI プログラム言語は C++ 言語を用いる。C++ 言語のコンパイルツールは Microsoft 社製 [5] の Visual C++2008Express Edition [6] を使用する。Visual C++ 上で MPI プログラミングをするには 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++ を起動しその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。最後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は mpi.lib である。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。これらの設定を行なうことで Visual C++ を用いて MPICH2 による並列プログラミングが可能となる。

2.5 最小全域木

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 (MST: Minimum Spanning Tree) とは、重み付無向グラフ $G = (V, E)$ に対して、G の部分グラフとなる全域木のうち、辺の重みの総和が最小となる全域木である。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。また、全域グラフとは、元のグラフ G に対し、G の全ての頂点を含むグラフを指す。木とは連結 (connected) でかつ閉路 (loop) が無いグラフである。つまり G の最小全域木とは、G から連結であり、かつ閉路を作るような辺をを取り除いた G の部分グラフの中で辺の重みの総和が最小となるような木であるグラフである。

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 の最小全域き T を求める問題である。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] 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 $O(m \log n)$、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。

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 のアルゴリズムを用いる。以下に Sollin の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 B。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、v の根となる頂点を探す。

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。

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: 定数個の代入のための定数時間である。

step 2-1: 定数個の足し算を行う定数時間である。

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る。よって時間は $\frac{n^2}{\log^2 n}$ プロセッサで $O(\log n)$ となる。

step 2-3: ポインタジャンプすることにより根までの距離が半分になるので $\log n$ 回で求められる。よって時間は $\frac{n^2}{\log^2 n}$ プロセッサで $O(\log n)$ 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 $O(\log n)$ 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。

2.6 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後生成される。

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 の部分行列をホスト 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　ホスト PC の内部計算時間と計算機数の関係

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	0.002700
2 台	0.000004	0.000008	0.000011	0.000030	0.000600	0.001600
3 台	0.000004	0.000006	0.000010	0.000025	0.000350	0.000950
4 台	0.000004	0.000006	0.000010	0.000020	0.000260	0.000850
5 台	0.000004	0.000006	0.000010	0.000018	0.000220	0.000680

(m 秒)

表 3　計算全体の処理時間と計算機数の関係

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	0.3	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	13.0
5 台	1.2	2.0	2.6	2.4	3.6	10.0

(m 秒)

3　結果

本研究では、頂点数が 5,10,20,40,80,160 の重み付無向グラフに対して、それぞれ計算機 1〜5 台を用いて MPI 上で最小全域木問題を解き、計算機数を増加させることでどれだけの処理時間が短縮されたかの計測を行なった。

以下の節では、今回の検証で得られた結果を MPI 上で最小全域木を求めたときのホスト PC の MPI プログラムの実行にかかる時間を内部計算時間と全体の処理時間それぞれについて示す。

3.1　内部計算時間

本節では、本研究で作成した MPI プログラムを用いて最小全域木問題を解いたとき、ホスト PC での通信時間を含まない内部計算時間を表 2 に、頂点数の変化に伴う処理時間の変化を図 2 に示す。表 2 および図 2 より、内部計算時間については、並列計算を使用した場合、計算機 1 台の場合と比べて処理時間の短縮が得られることが示される。また、処理時間の短縮効果は、頂点数が多いほどより顕著に得られることが示される。

3.2　全体の処理時間

本節では、計算機数の増加による全体の処理時間についての結果を表 3 および図 3 に示す。理論上は計算機数が増加するに従い処理速度は向上するはずであるが、表 3 および図 3 から示されるように計算機数の増加により逆に処理時間が悪化してしまっている。

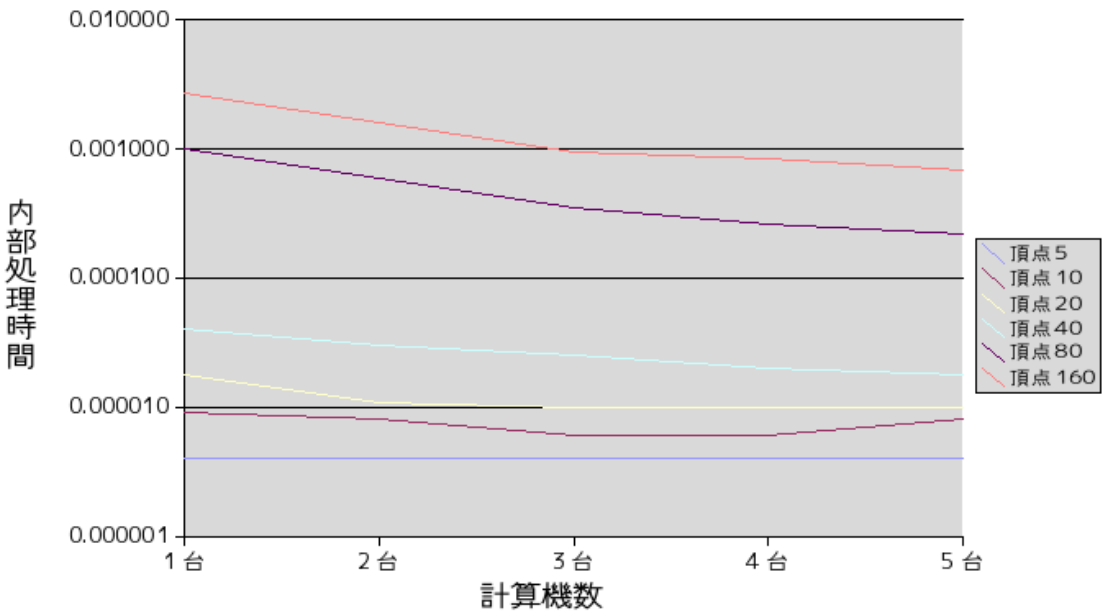

図 2　内部計算時間と計算機数の関係

3.3　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、

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

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、

$$T_{comp}(n,p) = 6.5n^2{*}10^{-4} + \frac{1.5n^2*10^{-4} - 45.0n*10^{-4}}{p} + 1.22\log^2 n{*}10^{-4} - 2.45\log n{*}10^{-4} + 1.16{*}10^{-4} \tag{2}$$

が得られる。

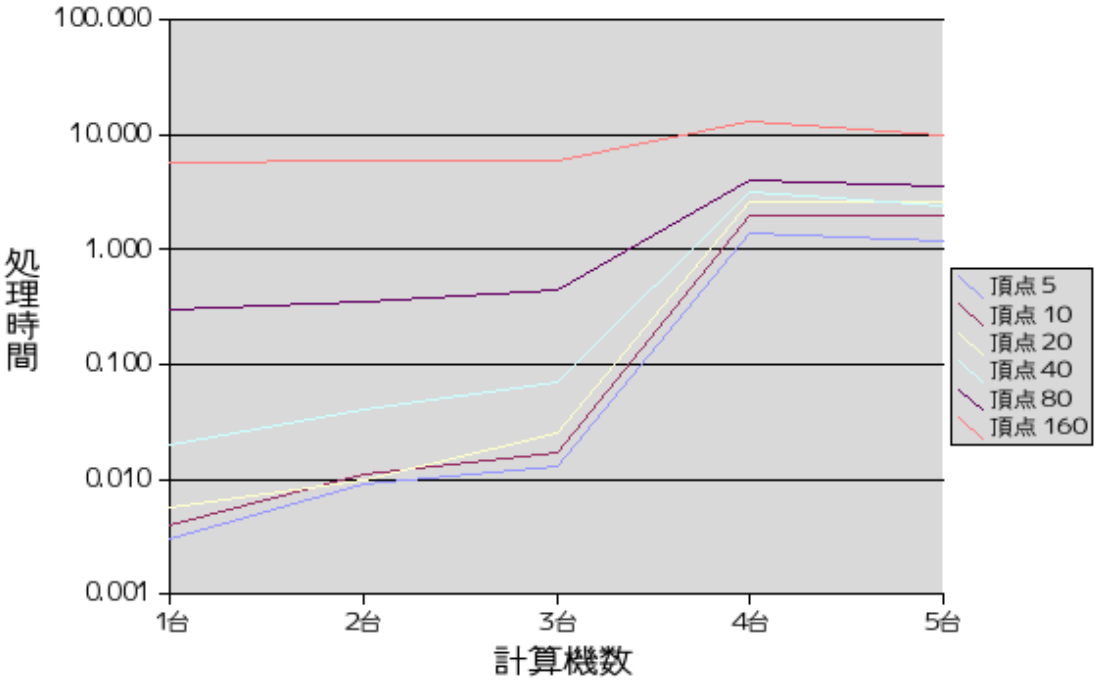

図 3　全体の処理時間と計算機数の関係

4　考察

表 2 よりホスト PC の内部計算時間はサブ PC の台数の増加に応じて処理時間の短縮が得られていることが示されるが、表 3 より全体の処理時間はサブ PC 増えるごとに遅くなっていることが示される。すなわち、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とから計算時間増大の原因は通信にかかる時間であると考えられる。本研究で使用した LAN は 100Bast-TX であり、充分に高速とは言えない。よって、通信時間を減らす為には、高速なネットワーク環境を構築することが必要だと考えられる。また、処理するデータ量をスペックの高い計算機と低い計算機で差をつけることにより、各計算機のエンコード処理時間の差を吸収し、効率よく通信ができ、全体の処理速度の向上が期待できる。

また、表 3 において、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 4 台めに増やした計算機サブ PC3 のメモリの容量が少なく、低スペックであったためと思われる。この検証のために、MPI ネットワークに計算機を加える順番を変更して実行した結果、計算機台数に関係無く、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ことから、上記の仮説は裏付けられた。すなわち、計算機の中で、他の計算機と比べてスペックの大きく低い計算機があると、それを MPI ネットワークに加えると通信時間は大きく増大してしまう可能性が高いことが示された。

5　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の計測を行った。しかし本研究の結果からは MPI の有効性を検証できたとは言えない。本研究の結果では、サブ PC の台数の増加に応じて内部時間は時間が短くなったが、通信に時間がかかるため全体の処理時間は計算機数の増加に反して長くなった。Sollin のアルゴリズムは、PRAM のアルゴリズムであるため、通信は一切考慮されていない。よって、MPI による並列化の真価を発揮するためには、通信を考慮した BSP や CGM のアルゴリズムを使うことが必要である。したがって通信環境を整備し、CPU の性能差を考慮して、データを負荷分散すること BSP[13] や CGM[14] 上で通信を考慮しアルゴリズムを開発することが今後の課題である。

謝辞

本研究におきまして並列処理について様々な助言を頂いた石水隆先生に感謝の意を表します。また、同じ研究を行うにあたり常に励ましてくれた研究室のメンバーには心から感謝しています。

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] 牛島省 著,OpenMP による並列プログラミングと数値計算法, 丸善株式会社 (2006).

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

[5] Microsoft Visual Studio Express,http://www.microsoft.com/japan/msdn/vstudio/

[6] Microsoft Visual C++ 2008 Express Edition,http://www.microsoft.com/japan/msdn/vstudio/express/

[7] yet-unnamed weblog, http://raeyoan.blog120.fc2.com/blog-entry-65.html

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

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

[10] Open MPI:Open Source High Performance Computing, http://www.open-mpi.org/

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

[12] 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

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

[14] 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 mpich2.cpp を示す。

mpich2.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);//  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[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);//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、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();



}
```