修　士　論　文

広域計算環境における
コンテンションフリー集合通信

A Contention-Free Collective Communication Algorithm in Wide-area Computing Environments

指導教員　**田浦 健次朗 准教授**

東京大学情報理工学系研究科
電子情報学専攻

氏　名　**48-086436 吉富 翔太**

提　出　日　**平成 22 年 2 月 9 日**

概 要

通信のコンテンションは広く普及している Ethernet で相互に接続された広域計算環境で，集合通信の通信性能を悪化させる大きな要因の一つであり， MPICH のようなの既存の MPI ライブラリでもコンテンションの影響により通信性能が悪化することがある．これを避けるためには，ネットワーク上の一部のリンクに通信が一斉に集中しないように同時に発生する通信の量を抑える必要がある．しかし，コンテンションを起こさないように通信を抑制しつつ，さらに高いスループットを得るための方法は自明ではない．例えば同時刻にただ 1 つの通信のみが 1 本のリンクを通過するようにしてしまうと，通信が逐次化して特に広域環境で広帯域なバンド幅の利用効率が低下し性能が頭打ちになる．したがって，ネットワーク中の全てのリンクに関して「いつ」「どれだけ」の通信が行われれば最適な性能が得られるかを精密に検証し，通信を制御する必要がある．

本論文では，個々の通信を一種のタスクと見なした上で，各々のタスクをいくつかにグループ分けして相互の実行に依存関係をつけることで，コンテンションフリーな通信を行うためのフレームワークを構成した．さらにそのフレームワークを利用して，コンテンションフリーにかつ広帯域なリンクをフルに利用できるような集合通信を実現できるアルゴリズムを提案した．このアルゴリズムはネットワークの遅延やバンド幅などのパラメータに関係なく，あらゆる環境で同一のアルゴリズムを適用することで，コンテンションフリーな多対一・多対多型の集合通信を行うことができる．

次に提案手法を実際に実装し，本アルゴリズムによるスケジューリングを行った結果に基づいて制御した集合通信と，他手法との性能の差異を検証した．その結果，単一クラスタのみを用いたものでは総実行時間を通信データ量が比較的少ないときに既存の MPI のライブラリと比べて alltoall 操作においておよそ 32%程度に，複数のクラスタを利用した場合では同様に特に gather 操作でコンテンションの抑制を考慮しない場合と比べて最大で 30%程度まで実行時間を短縮することができた．また特に広域環境においては，同期に費やす時間を削減するためにデータを転送するアプローチの有効性も示した．

目 次

第 1 章　はじめに **1**
1.1　グリッド環境での高な集合通信 1
1.2　本研究の目的と貢献 2
1.3　本論文の構成 3

第 2 章　関連研究 **4**
2.1　MPI のライブラリでの集合通信アルゴリズムの実装 4
2.1.1　クラスタ内での利用を想定した MPI 4
2.1.2　グリッド環境で使用可能な MPI 5
2.1.3　集合通信のチューニングを行う MPI 5
2.2　スケジューリング問題とみなした集合通信の最適化 6
2.3　ネットワークの構成情報の取得を行うフレームワーク 7

第 3 章　コンテンションの影響とその抑制法 **8**
3.1　コンテンションが発生する状況 8
3.2　集合通信のスループットに関する検証 8
3.3　ルートホストのデータ受信速度 9
3.4　スイッチにおけるパケットロスの検証 10
3.5　集合通信のスループットが低下する要因 11
3.6　コンテンションフリーな通信の実現 12
3.6.1　ホスト間で同期を取りつつ逐次的に送信 13
3.6.2　別の第三ホストへデータ転送 13
3.7　2 手法の通信コストに関する比較 13

第 4 章　コンテンションフリーな通信 **15**
4.1　前提条件 15
4.1.1　ネットワークの構造 15
4.1.2　ネットワークの構成情報の取得 15
4.1.3　耐故障性 16
4.1.4　ホスト間での個々の通信 16
4.2　タスクのグループ分け 17
4.2.1　グループの定義 17

4.2.2 タスクをグループに分類する意義 18
4.3 送信元が同じホストであるタスクの分類 19
4.4 リンクのバンド幅を考慮したタスクの分類 19
4.4.1 定義 19
4.4.2 手順 20
4.5 通信データ量を考慮したグループ分け 20

第 5 章 集合通信のスケジューリング 26
5.1 スケジューリングの概要 26
5.2 スケジューリングのステップ 27
5.3 グループの選択 27
5.4 タスクの選択 28
5.4.1 確実にタスク間の順序関係を満たした通信を行う方法 28
5.4.2 同期に要するコスト 29
5.4.3 タスク選択のポリシー 29
5.4.4 タスクの選択手法が総通信時間に与える影響 29
5.5 ホスト間でのデータ転送 31
5.5.1 データ転送を行うことの意義 31
5.5.2 タスク分割が可能な条件 32
5.5.3 再帰的なタスクの分割と分割回数の制限 34
5.5.4 転送の有無での通信コストの見積り 35
5.5.5 転送による総通信時間の短縮効果 36

第 6 章 実環境での実装と性能評価 38
6.1 アルゴリズムの実装 38
6.2 スケジューリングに関する性能評価 39
6.3 単一クラスタ環境における性能評価 40
6.3.1 gather 及び alltoall のスループット 40
6.3.2 データ転送の有無に起因するスループットの差異 42
6.3.3 遅延・バンド幅の変動が通信時間へ与える影響 43
6.4 複数クラスタ環境での性能評価 44
6.4.1 2 クラスタでの gather および alltoall のスループット 44
6.4.2 クラスタ数の増加とそれに伴う通信性能の変化 45

第 7 章 おわりに 48
7.1 まとめ 48
7.2 今後の課題 49

図 目 次

3.1 コンテンションの有無が gather のスループットに与える影響 10
3.2 1 クラスタ 32 ホストでのデータの受信ペース 11
3.3 同期を取りながら通信を行う場合 13
3.4 データを転送させながら通信を行う場合 13

4.1 リンクを共有するタスクのグループ化 18
4.2 入力バンド幅を考慮したタスクのコンテンションフリーなグループ分け 21
4.3 ネットワーク内のリンク，グループ及びタスク間の関係 22
4.4 一つのリンクに対するタスクのコンテンションフリーなグループ分け 23
4.5 4 ホスト，2 スイッチの環境における全対全通信のタスクのグループ分け 24
4.6 全対全通信での全てのタスク間の依存関係 25

5.1 スケジューリング 1 ステージのイメージ 27
5.2 データ受信後に複数ホスト間での同期を取る機構 29
5.3 シミュレーションに利用するネットワーク構造 30
5.4 タスクの選び方が総実行時間に与える影響 31
5.5 タスク分割が行われたときのホストの位置関係 34
5.6 データ転送の有無が集合通信の総通信時間に与える影響 37

6.1 単一クラスタ (istbs クラスタ) のネットワーク構成 39
6.2 生成されるグループの個数 40
6.3 1 回のスケジューリングあたりの所要時間 40
6.4 gather の実行時間 (96 ホスト) 41
6.5 alltoall の実行時間 (64 ホスト) 41
6.6 転送の有無による gather の実行時間の差異 42
6.7 遅延・バンド幅の変動が通信時間に与える影響 43
6.8 使用したネットワークの構成 44
6.9 2 クラスタでの gather の性能 46
6.10 2 クラスタでの alltoall の性能 46
6.11 クラスタ数の増加と通信性能 47

表 目 次

3.1 gather のスループット調査実験の実験環境 9
3.2 iperf によるスイッチでのパケットロスの調査 11

6.1 各クラスタの仕様 45
6.2 alltoall の通信に参加させたクラスタ 46

第1章　はじめに

1.1　グリッド環境での高な集合通信

昨今の計算機の性能向上やネットワークの高速化，クラウド技術の普及に伴い，複数の計算機を利用して科学技術計算に代表される様々な処理を並列分散して実行させることが一般的になっている．このような処理において，計算機間の通信，特に本論文で取り上げる多数の計算機間で一斉に複数の通信が行われる集合通信は，実に多種多様な場面で発生する．いくつか例を挙げれば，

- データインテンシブな処理における処理ファイルの分配や，処理結果の集約
- MPI に代表されるアプリケーションや，分散共有メモリ上でのデータの転送
- ネットワークファイルシステム上で複数のファイルの読み書き

など枚挙に暇が無い．従って，一般的に多くのアプリケーションの実行時間を短縮し性能を向上させることを目指すにあたり，その中で発生する集合通信の最適化を図ることは非常に重要な役割を持っている．

実際に集合的な通信の最適化を図る際に考慮しなければならないことは非常に多い．一つには，同じ操作をする集合通信であっても通信するメッセージの大きさや通信に参加するプロセス数によって最適な通信アルゴリズムが異なっていることが理由となる [9]．例えば広く利用されている MPI の実装である MPICH や MPICH2[4] では，MPI_Broadcast や MPI_Alltoall などの各種集合通信 API について，それぞれ通信されるデータサイズやプロセス数に応じて内部的に数個のアルゴリズムを切り替えて使用するように実装されている．

また，ネットワークの性能がヘテロな環境で特に重要になるのが，ネットワークの構造に応じて最適なアルゴリズムを使い分けたり通信のタイミングを変化させたりすることである．これは，ボトルネックとなりうる通信路に一度に大量の通信が集中しないようにするためである．例えば broadcast では，帯域の狭い WAN よりも広い LAN で通信を多く行うようにすることで，スループットを向上させる方法 [17] などが提案されている．

他にも集合通信で大きな問題となる要素として挙げられるのが，特に本研究で焦点を置いている通信のコンテンションが及ぼす影響である．コンテンションが発生する主要因は Ethernet で接続されたネットワークや中継機器のバッファの容量を超えるデータが流入しようとすることであり，通信のスループットを非常に悪化させる原因となる．例えば，1 つのスイッチにわずか 3 台のノードが

接続されたような環境において，2 ノードが残りの 1 ノードに対してデータを一斉に送りつけると，コンテンションにより激しく通信性能が悪くなることが分かっている．

集合通信に関する最適化に関する研究は数多く存在するが，多くは計算機間の End-to-End の遅延やバンド幅，送受信のメッセージ長といったパラメータのみを用いたり，あるいは LAN と WAN といった大雑把な区分でトポロジーの違いを考慮した上で集合通信をモデル化するものである [10, 27, 30]．そしてこのモデルに基づき，全体の実行時間が最小となるようにノード間の通信パターン決定してゆく．これらは厳密なネットワークトポロジーを考慮せず通信パターンが作られるので，ネットワーク中の一部分のリンクに通信が同時に集中してしまうことで，コンテンションが発生しうるという問題点がある．

そのため，ネットワーク上のあらゆるリンクにおいて，通信のコンテンションを発生させないようにホスト同士の間で通信を制御しあうことが集合通信のスループットを悪化させないためには要不可欠である．このような通信を実現する方法としては，複数の通信が意図せず重複するのを防ぐために，ホスト同士で逐一通信を同期させながら排他的に通信を行う方法 [11] が挙げられる．ところが，ホストの間で通信を同期させるのに必要な時間は一般にホスト間のネットワークの遅延に比例するため，特に広域・高遅延なネットワークが存在するグリッド環境では，同期に要する時間が無視できないほど大きくなり集合通信のスループットがかえって低下してしまうという問題が起きる．さらに，近年の WAN のバンド幅の広帯域化に伴い，できるだけ多くのデータを同時に流すほど帯域を大きく占有することができ結果的にデータを高速に転送することができるが，ホスト間で排他的に同期をしながら通信を行う場合には，同時刻に任意のリンクを高々1 組の通信しか利用しないためその恩恵を受けることができず，WAN での通信性能が向上しないという問題点もある．

1.2 本研究の目的と貢献

本論文では，ネットワーク上で通信のコンテンションの発生を抑制させることで特に gather 操作に代表されるような多対一型の集合通信や all_to_all 操作に代表される多対多型の集合通信で高いスループットを得られるようなアルゴリズムを提案する．さらに，クラスタ内で完結するようなネットワークが低遅延な環境から，複数のクラスタをまたいだようなネットワークが高遅延な環境まで，あらゆる環境で同一のアルゴリズムを用いて効率の良い集合通信を実現する．

前述のようにホスト同士の間で同期を取りながら通信を逐次的に行うだけでは，ネットワークの遅延が大きかったり，通信の規模が大きくなると良い性能は得られなくなってしまう．しかし，通信のコンテンションを発生させないようにするには，ネットワークの一部の領域に同時に通信が集中してしまうのを防止するような機構を設けることが必須である．そこで本研究では，通信のコンテンションを抑制させるために大別して次の 2 通りの通信に関するフレームワークを採用する．

一つ目は既存のいくつかの手法と同じようにホスト同士が同期を取りながら逐次的に通信を行う

ものである．不必要な同期はなるべく行うことのないよう，同期回数と同期時間の削減により集合通信全体のスループットを向上させることを目指すものである．

二つめは各ホスト同士でデータを転送させていくというものである．データを転送させることによって，通信の宛先を分散させることができる．そのため結果としてネットワークの一部に通信が集中してしまうことを防止することができるとともに，ホスト間で通信を同期させる必要もなくなるために，同期に要する時間を減らすことにもつながる．

一概に集合通信のスループットを向上させると言っても，問題は全く容易ではない．まず上記の 2 つのフレームワークに基づいて通信を制御したとき，その全体の総通信時間はネットワークの遅延やバンド幅依存となるためにどちらの手法が優れているかは自明ではない．さらに両手法を厳密に適用し，リンクのバンド幅に関わらず完全に一つのリンクを排他的に通信が行われるようにしてしまうと，通信が逐次化してスループットが頭打ちになる．したがって，ネットワーク中の全てのリンクに関して「いつ」「どれだけ」の通信が行われれば最適な性能が得られるかを精密に検証し，通信を制御する必要がある．本研究ではこのフレームワークを踏まえて個々のホスト間の通信を上手く順序づけて排他的に行わせることでコンテンションフリーかつスループットの高い集合通信を実現する．

1.3 本論文の構成

以降の本論文の構成は次の通りである．

第 2 章 関連研究 集合通信の最適化手法に関する既存の研究を紹介し，それらに対する本研究の位置づけを述べる．

第 3 章 コンテンションの影響とその抑制 本研究で問題としている通信のコンテンションが，集合通信の実行時間にどれぐらいの影響を与えるのか検証した後，その抑制方法について述べる．

第 4 章 コンテンションフリーな通信 ネットワーク上でコンテンションを発生させないような，コンテンションフリーな通信を実現するためのフレームワークについて説明する．

第 5 章 集合通信のスケジューリング コンテンションフリーな集合通信を行うための通信のスケジューリングの手法について述べる．

第 6 章 実環境での性能評価 本手法を実環境で実験した結果を述べる．

第2章　関連研究

2.1　MPI のライブラリでの集合通信アルゴリズムの実装

並列分散環境で利用される MPI ライブラリは数多く存在している．それらのライブラリでは，多くの場合標準規格として集合通信を行う API が実装されている．例えば Broadcast であれば MPI_Bcast，Alltoall であれば MPI_Alltoall などの API がそれに相当する．これらの API の内部での実装はライブラリごとに様々で，使用されているアルゴリズムの良し悪しによって集合通信の性能が大きく左右される．

2.1.1　クラスタ内での利用を想定した MPI

均質な環境での利用を想定している MPI のライブラリの代表例としては MPICH[4] や OpenMPI[5] などが挙げられる．これらのライブラリでは，それぞれの集合通信について一つ以上のアルゴリズムが実装されており，利用される環境での計算機 (プロセス) の数や計算機間のネットワークの遅延やバンド幅さらには通信されるデータの大きさなどのパラメータをもとにして，実行時に最適なアルゴリズムが選択されるようになっている．

例えば MPICH2 では，broadcast が 2 つ以上のアルゴリズムの切り替えによって実行される [7, 29]．一つはルートホストを根とする binomial tree を作成し，このツリーに沿ってデータを通信していくというものである．もう一つは実行時のホスト数が 2^n 台となる場合に限り，最初にデータを n 個に分割して scatter しておき，その後全ホスト間で allgather するとことで broadcast された状態を実現するアルゴリズムである．後者のアルゴリズムはホストの数が 2^n となる場合には前者のアルゴリズムに比べて通信に要する全コストが小さくなるという特徴がある．

またコンテンションに対する対処法の面では，例えば gather については OpenMPI ではルートホスト以外のホストがデータを一斉に送信するという最も単純な実装が行われている．一方 MPICH2 では binomial tree を用いて階段状にルートホストに向かって通信を行わせる実装になっている．ところがいずれの手法においても，ネットワークのトポロジーや同時に発生する通信間の協調を考慮しているものではないので，コンテンションにより通信性能が悪化する可能性がある．

2.1.2 グリッド環境で使用可能な MPI

一方グリッド環境に適応した MPI ライブラリ中での集合通信の最適化に関する研究も数多く行われている．

MagPIe[17] は，MPICH2 のメッセージサイズやネットワークの遅延・バンド幅を考慮して最適な通信を行うという手法を，複数のクラスタにまたがった環境で使用できるように拡張した MPI ライブラリである．MagPIe ではクラスタ内とクラスタ間での通信のアルゴリズムを使い分けることにより，WAN のように遅延の大きなホスト間の通信を極力減らしつつ，クラスタ間のバンド幅を無駄遣いしないような集合通信の実装が行われている．具体的には，各クラスタに対し一つだけ coordinator node と呼ばれるホストを割り当てる．この coordinator node のみがクラスタ間通信を代表して行い，クラスタ内の通信は MPICH と同様のアルゴリズムを用いて行う．

MPICH-G2[15] も MagPIe 同様複数クラスタにまたがった環境での使用を想定している MPI ライブラリである．MPICH-G2 が MagPIe と異なる点は MagPIe がクラスタ間とクラスタ内のように，明らかに遅延が異なる 2 種類のネットワーク構造だけを考慮しているのに対し，MPICH-G2 はより一般的に通信の最適化を目指している点である．

GridMPI[23] は，クラスタ間の遅延はクラスタ内の遅延よりも非常に大きいとみなす一方で，バンド幅が大きいと仮定し複数ホストがクラスタ間の通信に寄与することで高バンド幅なネットワークを有効利用することにより，通信のスループットを高めるものである．

以上の手法では，ネットワークのトポロジーを詳細に考慮したものではなく，またコンテンションを抑制することを目的としているわけではないので，多対一あるいは多対多型の集合通信で性能が悪化してしまう可能性がやはり存在する．また，遅延やバンド幅の値を正確に利用し個々の通信の制御をさせるものでもないため，実行環境に最適な通信のパターンが構築されるわけでもない．

2.1.3 集合通信のチューニングを行う MPI

集合通信によってどのアルゴリズムが実行されるかがアプリケーションの実行時に一意に定める手法ではアプリケーション実行中のネットワークトポロジーや利用可能なバンド幅の変動に対応することが難しい．さらに，アーキテクチャの違いやネットワークの遅延・バンド幅が一つ一つ異なる場合などの非常に不均一な環境では最適な性能が得られない．

そこでアプリケーション実行前に集合通信のアルゴリズムをシステムに応じてチューニングする手法が提案されている [10, 31]．例えば，Vadhiyar らの方法では実際の環境で様々なデータサイズやホスト数の条件下で実際に集合通信のアルゴリズムを実行させて実行時間を測定した予備データを環境構築時に予め取得しておき，実際のアプリケーションによる通信時にはこの予備データを元に最大の性能を得られるアルゴリズムを選択し実行する手法を提案している．

STAR-MPI[6] はアプリケーション実行中に集合通信のアルゴリズムを動的に選択してチューニングを行うものであり，色々なアルゴリズムをアプリケーション実行中に実際に試行して最も高速なものを選択するという実装である．実行中に外乱の影響により速度が低下したら別のアルゴリズムに随時切り替えることによって，トータルの集合通信のスループットを高めるものである．

これらは，実行環境の状態に応じて相対的に性能の良いアルゴリズムが実行されるという点では様々な適応性が高い手法であると言える．しかし，根本的に通信のコンテンションの抑制や通信経路の最適化を図るものではない．

2.2 スケジューリング問題とみなした集合通信の最適化

ネットワーク遅延やバンド幅の値がホスト間ごとに全く相異なる環境で，一般的な集合通信の最適化を図る研究もなされている．

これらの手法の多くはホスト間の遅延やバンド幅，通信されるデータの大きさなどからそれぞれ個々の通信をモデル化し，集合通信を一種のスケジューリング最適化問題として考慮するものである [8, 11, 12, 14, 28]．モデル化の一例としては，例えば LogP モデルなどが用いられるものが存在する [21, 32].

これらの多くは [12] に代表されるように例えば遅延・バンド幅・通信データ長からホスト間の通信コストを算出し，全体の総通信時間が最小となるように各ホスト間の通信をスケジューリングを行う．ところがこれらの手法は同時に同一のリンクを多数の別々のホストからの通信が利用されるようにスケジューリングがなされてしまう可能性がある．そのような場合にはコンテンションの発生が抑えられない．

一方で Faraj らによる手法で [11] は，エンドホストと中間機器のスイッチのみで構成されたノードがツリー状の構造を持つ環境で，送信元のホストが異なる通信同士が同一のリンクを同時刻に使用することがないように各ホストの通信先を調整することによって，コンテンションを防止した all to all 操作を実現するアルゴリズムである．各ホストが相互に同期を取りあうことによって，一つのリンクが同時に複数ホストに使用されないようにしている．この手法はある程度通信を行うデータが大きい場合には効果があるが，データが小さかったりネットワークの遅延が大きい場合では，ホスト間で同期を取るために必要な時間が相対的に大きくなるために性能が悪化してしまうという欠点があり，あらゆる実行環境で最適な手法とは言い難い．

松葉らによる Aggregate Router[22] は， LAN 内から外に出て行くトラフィックを少数のサーバに分配・中継させることによって，通信がバースト的に集中してしまうことを防ぐとともに広帯域な WAN のバンド幅も占有することができるようなフレームワークである．しかし，これは LAN の中では通信のロスがないことを仮定しており，LAN 内でのコンテンションの発生までも抑制するものではない．

2.3 ネットワークの構成情報の取得を行うフレームワーク

白井らによるネットワークトポロジーを高速に推定する手法 [26] は，ネットワークトポロジーのツリー表現を扱い，エンドホスト間の RTT だけを用いてツリーを推定する手法であり，ネットワークへの負荷が低く高速 256 ホストのトポロジーを 15 秒程で推定することができる．この手法によりネットワークの階層構造及びホスト間の遅延の情報を取得することが可能である．

長沼らによる bhtree[24] は，大規模ネットワークのバンド幅マップを取得するフレームワークである．300 ホスト程が接続された広域分散環境において，およそ 2 分程度でバンド幅マップの構築が可能である．bhtree は与えられたネットワークトポロジーを細かく分解し，測定の競合しない部分トポロジーを並列に測定することで高いスケーラビリティを得ることができる．

これらの方法は．いずれも実行環境において特定の権限[1]を必要とせず，ユーザが独自に測定プログラムを走らせることでネットワークの構成情報を自由に取得できる点が特長である．本研究ではこれらの手法により取得したネットワークのトポロジー・ホスト間の遅延・バンド幅の情報をもとに，通信の制御を行ってゆく．

[1] おもに管理者権限

第3章　コンテンションの影響とその抑制法

3.1　コンテンションが発生する状況

本研究で焦点を当てている通信のコンテンションがどのような状況で発生し，それによって集合通信の通信性能にどれほどの影響があるのかをアルゴリズムの構築に先立って検討する．

帯域が大きいネットワークからルータやスイッチなどの機器を介して帯域の小さいネットワークへとデータが流れ込むときに，データが流入する速度が必然的に低下する．その場合中間のルータやスイッチなどの中継機器で溢れたデータがバッファされることになる．しかし，中継機器のバッファの容量以上のデータが短時間に流入した場合，データはバッファされずに破棄されてしまう．そのため特に多対一型の通信や多対多型の通信のように，通信が多数のノードから短時間の間に一点に集中する特徴を持つ集合通信で，それぞれの通信が協調せずに行われたような場合には，この現象がネットワーク中の至る所で発生することになる．

1. ルータやスイッチに接続されている 1 本のリンクに対して，同じルータ・スイッチに接続されている別の複数のリンクからの通信が同時に集中する，
2. データが流出する側のリンクの帯域が大きくとも，そのリンクに接続されている次の宛先となるマシンやルータ・スイッチなどの処理能力を超えてしまうような，大量のデータが集中する．

こういった状況下では，発生頻度はその流れ込むデータの量や受信側の機器のデータの処理速度やバッファの性能に依存するものの，スイッチのバッファ溢れによるパケットロスが生じることがある．

パケットロスが生じた場合，UDP ではデータが届かなくなるため直接的な影響が生じる．TCP では再送機構が働くため，データが消失するという意味では直接の影響はないが，後述するようにその再送機構のために逆に通信性能が非常に悪化してしまうことがある．

3.2　集合通信のスループットに関する検証

まず，通信のコンテンションが具体的な集合通信のスループットに与える影響について検証する．関連する現象については文献 [18, 19, 20] によっても関連する現象が報告されている．

簡単に影響を確認する方法としては，1 つのスイッチに複数のマシンを接続した環境下で，多対一型通信の一つ gather 操作を行うものが挙げられる．そこで，以下のような簡単な予備実験を行った．

この実験の主目的はでコンテンションの有無が gather のスループットに与える影響について検討することにある．

表 3.1: gather のスループット調査実験の実験環境

ホスト	台数:32 台　OS: Linux Kernel 2.6.18　CPU:Intel Core2 E6400 2.13GHz
スイッチ	FastIron LS 648
ネットワーク	Ethernet 1Gbps (各ホストに 1 本ずつ接続)

表 3.1 のような環境で，2 台から 31 台のホストが各々1MB のデータを，残りの 1 台のホスト[1]に向かって送信を行う．具体的には，次のような 2 通りのやり方で，gather 操作をする．

1. それぞれのホストが一斉にルートホストへの送信を開始する場合．
2. 1 ホストずつ逐次にルートホストへデータを送信する場合．ルートホストは一つの送信ノードからデータを受け取ったら直ぐに別のまだデータを送信していない送信ホストへと 1byte のデータを送信する．送信ホストはルートホストからの前述のデータを受け取った場合にのみ送信を行うものとする．

前者はコンテンションを発生させることを意図した方法である．ルートホストに接続されているリンクに対して，最大で 31 台のホストからの通信が同時に集中するためコンテンションが発生しやすい状態になっている．後者は逆にコンテンションを発生させないように通信を行わせるものである．ルートホストは同時に高々1 ホストとの間でしか通信を行わないからである．このような 2 つの通信方法で gather を行った場合にスループットにどれだけの差が生じるかを比較した．なお，以降の結果におけるスループットとは，ルートホストが受信するデータの総量を送信ホストのいずれかがルートホストへ通信を開始してから，ルートホストが通信を終了するまでの時刻で割ったものとして算出している．

図 3.1 に上記実験の結果を示す．図 3.1 の縦軸は送信を行ったホスト数であり，縦軸がスループットである．図 3.1 の結果を見ると，逐次に通信を行った場合 (sequential) は送信ノード数の多寡に関わらずおよそ 700Mbps から 800Mbps のスループットが得られている．一方で，全てのホストが一斉にデータを送信した場合 (concurrent) は，送信ホスト数が 3 ホストを超えたところで 200 から 300 Mbps とスループットが半分以下に激減してしまっていることが分かる．

3.3 ルートホストのデータ受信速度

同様の環境・条件でデータ送信側のホストを最大の 31 台に固定した場合で gather 操作を行い，ルートホストがどのようなペースでデータを受信していくのかということについてを調査を行った． 図

[1]以下ルートホスト

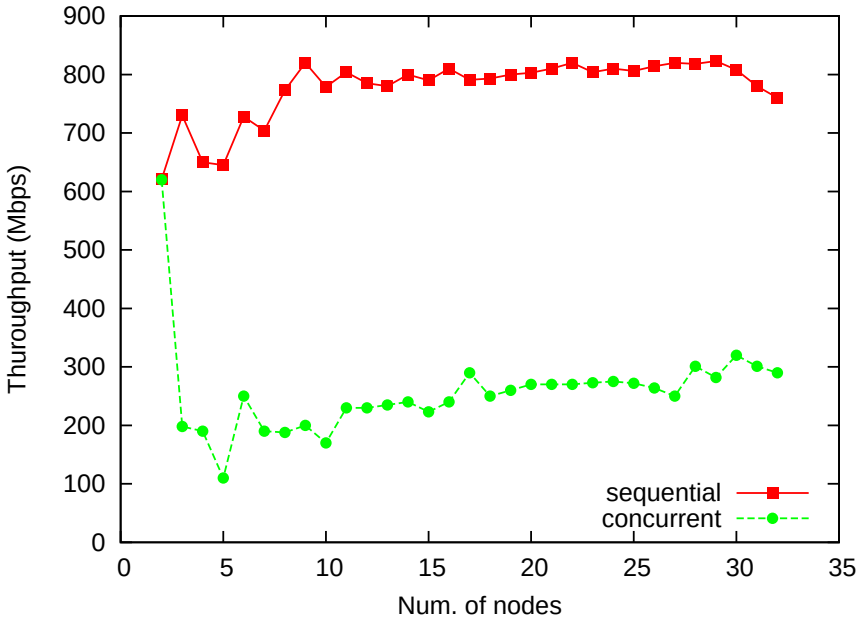

図 3.1: コンテンションの有無が gather のスループットに与える影響

3.2 は，受信ホストを除く 31 ホストがそれぞれ 1MB のデータを送信し，ルートホストが合計 31MB のデータを受信していくペースを時系列に沿ってプロットしたものである．

図 3.2 を見ると，逐次的に通信を行った場合 (sequential) は時間軸に対してほぼ線形に受信したデータ量が増えていっている．そのため，ルートホストは間断なくデータを受信することができていると考えられる．一方で，送信ホストが一斉にデータを送信開始した場合 (concurrent) には，図 3.2 中では 4 箇所ほど，ルートホストがデータを 1byte も受信していない時間帯が存在する．このようなルーホストが全くデータを受信しない時間が存在することが，図 3.1 において concurrent が sequential に比べてスループットが激減している原因だと判断できる．

3.4 スイッチにおけるパケットロスの検証

次に，送信ホストが一斉にデータをルートホストに送信した場合，中間のスイッチにおいてパケットロスが発生しているかどうかについて検証を行った．

ここでは，比較のため別途表 3.2 に示すような 1 つのスイッチに 14 台のホストが接続されているような環境を 2 つ用いた．そこでバンド幅測定ツールである iperf[3] を用いて， 13 台のホストが gather 操作の要領で UDP のパケットを 1 秒間集中的に送信しパケットがルートホストに到達する状況を観察した．ある一回の一連の送受信についてルートホスト側での通信状態の様子は表 3.2 のよ

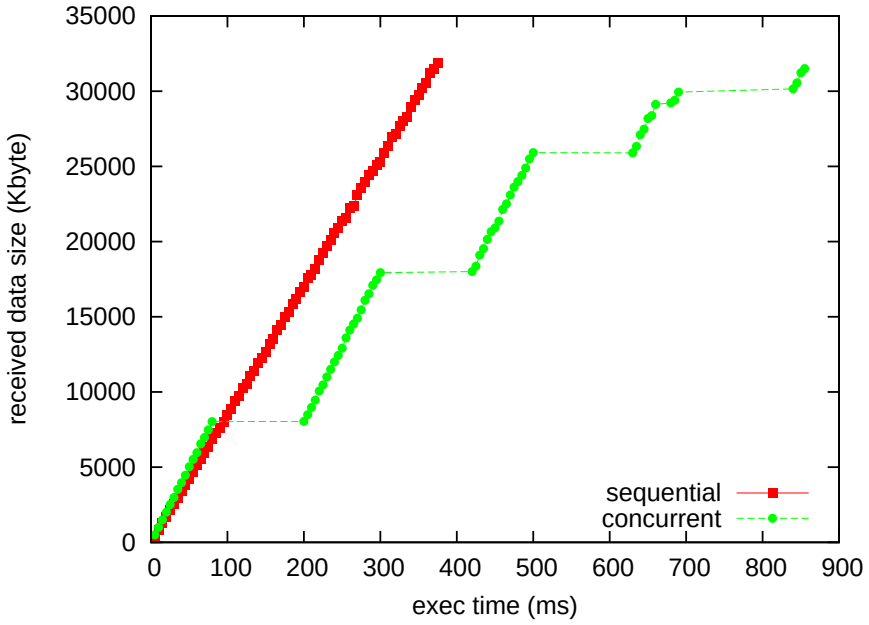

図 3.2: 1 クラスタ 32 ホストでのデータの受信ペース

うであった．この結果は，スイッチの性能により差はあるものの，中間のスイッチにおいてパケットロスが発生しているということを意味している．

表 3.2: iperf によるスイッチでのパケットロスの調査

スイッチ	受信データ量	受信パケット数	損失パケット数
PowerConnect 5324	11.17MB	8108	180
FastIron LS 624	11.9MB	8510	21

3.5 集合通信のスループットが低下する要因

以上の調査により，gather においてデータを協調せずに一斉に特定の 1 ホストに向けて送出すると，スループットが明確に低下することが分かった．また図 3.2 でデータを受信できている時間帯での両者のグラフの傾きを比較するとほとんど差がないことから，ルートホスト側では到達したデータを処理することができており，ルートホスト自身がスループットを低下させている原因ではなく，3.2 の結果から途中のスイッチにおいてパケットロスが生じていることが原因であると考えられる．

TCP では，パケットロスが発生した場合は送信ホストが受信ホストから重複した ACK を 3 回受信すると，そのパケットが相手に届いていないと判断しパケットの再送を行うという高速再送のアルゴリズムが働く [13]．しかし，通信開始直後のウィンドウサイズが小さい状態での通信や，データストリームの末尾など後続するパケットが十分存在しない部分でパケットロスが起きてしまうと，受信側のホストではそれ以上のデータを受け取らないため ACK を送ることができなくなってしまう．その場合，送信ホスト側はある一定の待ち時間だけ待ってからパケットの再送を行う．この値は RTO(Retransmittion Time Out) と呼ばれるものであり，図 3.1 および図 3.2 の実験に使用したホストの Linux kernel 2.6.18 では RTO の値は過去の RTT の重み付き平均と分散から動的に調整される [25] が，その最小値はデフォルトで 200 ミリ秒に設定されている．従って図 3.2 の結果は，次のように説明できる．

1. 時刻 $t = 0$ に複数のホストが同時にルートホストに向けて送信を開始する．
2. 通信の衝突により一部のホストからの通信同士の間で中間のスイッチにおいてパケットロスが発生する．そこで高速再送が行われなかった場合は，パケットロスが発生してしまった通信を行っていた送信側のホストは 200 ミリ秒の RTO の間だけ通信が停止する．
3. 先の通信開始直後 ($t = 0$ の近傍) で通信が停止してしまったホストが，$t = 200$ ms から通信を再開する．残りの通信が完了せず停止しているホストも，通信が停止した時刻の 200 ミリ秒後に順次通信を再開してゆく．
4. また一部のホスト同士の通信が衝突することになり，(2) のステップへと戻る．

また図 3.1 からも分かる通り，高々2 ホストが同時に 1 ホストにデータを同時に送信する場合でもスループットが悪化しているので， gather 操作だけに限らず一般の多対一型あるいは多対多型の集合通信で高いスループットを得るためには，別々のホスト同士の通信を上手く制御しちゃることで通信が衝突することを防ぎ，コンテンションの発生を抑制することが必要不可欠である．

3.6 コンテンションフリーな通信の実現

複数のホストが同時に同一のホストに対して一斉に送信を開始した場合，リンクの帯域・中継機器のバッファ容量の限界，さらに TCP の性質が組み合わさることで通信性能が劇的に悪化することが実際に容易に発生することをこれまでの予備実験で検証した．

この現象が起きることを避けるためには，それぞれのリンクを同時に必ず一組のホスト同士が通信に使用すれば良い．この排他性を実現するために，本研究では 3.6.1 および 3.6.2 に挙げる 2 通りの考え方を用いる．

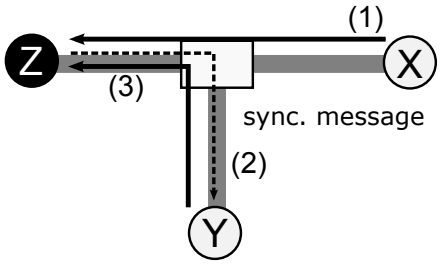

図 3.3: 同期を取りながら通信を行う場合

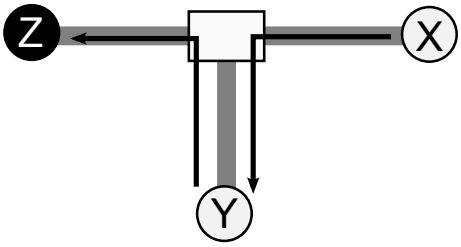

図 3.4: データを転送させながら通信を行う場合

3.6.1 ホスト間で同期を取りつつ逐次的に送信

一つ目は図 3.3 に示すようにホスト同士で同期を取りながら通信を行う方法である．

1. X がまず Z に対してデータを送信する．
2. Z は X からのメッセージを受信したら，Y に対して 1 バイトのパケット (同期メッセージ) を送出する (これを同期操作と呼ぶ).
3. Y は Z からの確認メッセージを受信するまでブロックし，同期メッセージを受信し次第 Z に対してデータを直接送信する．

一見して自明なこの手法であるが，1 つ欠点が存在する．(2) における同期のコストは遅延に比例するので，クラスタ間をまたがった遅延が大きいホスト間では無視できないものとなることである．

3.6.2 別の第三ホストへデータ転送

二つ目は図 3.4 に示すようにホスト間でデータを転送しあう方法である．ここでは，X は Z に対して直接データを送るのではなく，別のホスト Y に対して Z に送るべきデータを送信する．一方 Y は，自身が本来保持しているデータを Z に送りつつも， X からのデータを待ち，データを受け取り次第即座に Z に対して再び送出する．このように，$X \rightarrow Y \rightarrow Z$ とあたかも一本のパイプラインホストが結んであるかのようにデータをホスト間でリレーしていく．

この手法では，パイプライン上のあらゆるホストは別のあるホストに対してのみ独占的にデータを送り，別のホストから独占的にデータを受け取る．ゆえに複数のホストから同一の宛先ホストへと向かうデータは存在しえないため，1 つのリンクをそれぞれのホストは排他的に使用することが可能である．

3.7 2 手法の通信コストに関する比較

これらの理論的な通信コストの比較を行い，手法のメリット・デメリットについて論じる．

簡単の為，１クラスタ内でリンクの遅延 L やバンド幅 B が全て等しい環境を考える．このとき，ルートホストが 1 つのメッセージを受信し始めてから受信し終わるまでの時間 T_r は，通信パターンによらず一定で，バンド幅，メッセージサイズ M を用いて $T_r = \frac{M}{B}$ と表せる．ここで，ルートホストが一つのメッセージを受信してから次のメッセージを受信するまでの受信待ちをしている間隔を到着間隔 G とする．すると，G の値が小さければ小さいほど効率の良い通信パターンであると言える．

以上の定義を行った上で，

1. N ホストが全て逐次にデータを送信
2. MPICH のように Binomial tree を使い，かつ同世代内では逐次に同期を取りつつ通信
3. N ホストが全てパイプライン転送

の 3 通りの手法の総通信時間 (コスト) の比較を行う．メッセージ自体を受信している時間 T_r は一定であるので，G の値のみを比較すれば良い．

まず 1 の手法では，同期に要する時間 $2L$ がそのまま受信間隔となる．同期回数は N 回なので，総受信間隔 W_s は

$$W_s = 2LN \tag{3.1}$$

となる．同様に 2 の手法は，同期に要する時間がそのまま受信間隔となるが，Binomial tree を組むことにより同期回数を $\log N$ 回に抑えることができるから，総受信間隔 W_t は

$$W_t = 2L \log N \tag{3.2}$$

である．最後に 3 の手法は，総受信間隔 W_p は L, M, B の値に依存し次式のようになる．

$$W_p = \begin{cases} 0 & L \leq T_r \text{ のとき} \\ (L - T_r)N & L > T_r \text{ のとき} \end{cases} \tag{3.3}$$

一見コンテンションを防止する自明な方法と考えられる 1 や 2 の手法は W_s や W_t が遅延の 2 倍 $2L$ に比例しているため，T_r が L と比べて大きく，なおかつクラスタ間の通信のように L が大きくなる場合は $W_s, W_t > W_p$ となる．よって，1 や 2 の手法を任意の環境で単に機械的に当てはめるだけでは，効率が悪くなってしまう．

一方で 3 の手法では M が非常に小さく $T_r << L$ となるようなときには他手法と比べて不利となる場合がある．例えば，$N = 8,\ 4T_r < L$ であるとき，上記の式にあてはめ W_t と W_p を比較すると $W_t - Wp < 0$，すなわち $W_t < W_p$ となってしまう．従って全てのデータをホスト同士の間でパイプライン状に転送していく方法もまた最適な方法になるとは限らないということを意味している．一部だけ同期を取って逐次的に通信しても，同期を取るコストを考慮しても通信性能は悪くならないことがあるからである．

次章では，上記の 2 通りの考え方もとにコンテンションフリーな集合通信のスケジューリングを行うフレームワークを構築していく．

第4章　コンテンションフリーな通信

本章では特に多対一・多対多型の集合通信において，集合通信実行中のコンテンションを抑制させることを目的とした「コンテンションフリー」な通信とそれを実現するアルゴリズムについて述べる．

4.1　前提条件

アルゴリズムを説明するにあたり，ネットワークやホスト間で行われる通信についていくつかの前提及び仮定を設定する．

4.1.1　ネットワークの構造

- 本研究で対象とするネットワークの構成は，通信に参加するエンドホストとルータやスイッチのような通信の中継に関与する機器の２種類のみで構成されているものとする．そして，通信の合流・分岐は全て１つ以上の中継機器を介して行われるものとする．つまり，２つ以上のエンドホスト同士が中継機器を間に挟まず直接接続されているような構造を取ることは無い．
- 一つのエンドホストと中継機器，もしくは中継機器同士の接続のにおいて複数本のリンクが束ねられられていることがある[1] ものとする．このような接続においては一つの通信で束ねられたリンクの本数分のバンド幅を得ることができるため，この場合は単純に束ねられたリンクの本数分のバンド幅を持つようなリンクが存在するものと考えて良い．
- ネットワーク上の任意のホスト H_x から H_y への経路はただ一つに定まる．これはネットワークがツリー構造をなしていれば必ず成り立つ仮定である．ネットワークがツリー構造をしていない場合でも，この仮定が成り立てば後述のスケジューリング手法は上手く行く．

4.1.2　ネットワークの構成情報の取得

提案する本アルゴリズムでホスト間の通信の経路やタイミングを正確に制御するためには，集合通信のアプリケーションが実行さるネットワークに関するネットワークのトポロジーや遅延・バンド

[1] bonding, trunking

幅の情報をあらかじめ取得しておく必要がある．これらのネットワークの構成情報は，第 2 章で述べた [24, 26] などの手法で得ることできる．

これらの手法によるネットワーク構成情報の取得時間は，ネットワークの規模に依存するが数秒から数十秒程度を要する．この時間は一度の集合通信の実行時間に比べて非常に大きくなってしまう場合がある．しかしネットワークの静的な状態は頻繁に変わるものではないと考えられるので，本手法ではあらかじめ事前に推定・取得しておいたネットワークの構成情報を入力として用いるものとし，ネットワーク情報の取得に要するコストを考慮しなくても良いものとする．

なお，提案するアルゴリズムで最低限必要な情報は，ネットワークのトポロジー，ホスト間の片道の遅延及び，ネットワークを構成する一本一本のリンクのバンド幅である．これらは全て，上記の方法で取得することが可能である．

4.1.3 耐故障性

本研究で提案するアルゴリズムにおいては，スケジューリングの対象としている集合通信の実行中にホストが故障したり，ネットワークが切断されるなどの理由でネットワークのトポロジーの構造が変動することは考慮していない．

4.1.4 ホスト間での個々の通信

本アルゴリズムにおいては制御対象の集合通信は 1 つのネットワーク中である時刻にただ一つだけ発生するものと仮定する．そのため複数の集合通信同士が互いに干渉しあうことはなく，一度にただ一つの集合通信に関してのみホスト間の通信を制御することを考慮していればよい．

はじめに，個々のホスト間での通信は次に示す性質を持つと考えられる．

定義 1 (ホスト間通信の性質).

ホスト X からホスト Y へと M byte の通信が行われるとする．X と Y 間の経路上のリンクを $e_0, e_1, \cdots, e_k$ とし，この順にデータが流れていくとする．このとき，リンク e_i の遅延を l_i，バンド幅を b_i とすれば， X と Y 間の遅延とバンド幅は

$$L_{xy} = \sum_{i=0}^{k} l_i \tag{4.1}$$

$$B_{xy} = \min(b_0, b_1, \cdots, b_k) \tag{4.2}$$

である．したがって，時刻 $t = T$ に X から Y へ向かう通信が開始した場合， X を Mbyte のデータ列の末尾が出発する時刻 α_{xy} は

$$\alpha_{xy} = T + \frac{M}{B_{xy}} \tag{4.3}$$

であり，Y にデータ列の末尾が到達する時刻 β_{xy} は

$$\beta_{xy} = T + L_{xy} + \frac{M}{B_{xy}} \tag{4.4}$$

と表せる．なおここでは送受信のオーバーヘッドなどは無視できるものとする．

それぞれのホスト間の通信を以降の説明のため,「タスク」と呼ぶことにし，ホスト X から Y への通信を行うタスクを J_{xy} と表す．さらに J_{xy} の通信に要する時間 C_{xy} は

$$C_{xy} = \beta_{xy} - \alpha_{xy} = L_{xy} + \frac{M}{B_{xy}} \tag{4.5}$$

であり，以降この C_{xy} を J_{xy} の通信コストと呼ぶことにする．

ここでネットワークは全二重であるとし，各ホストは接続されている一本のリンクに対して，送信操作及び受信操作を同時刻にそれぞれ独立に行うことができるものとする．すると本研究で考慮すべき集合通信の実体は，いくつものタスクの集合であって，それらのタスクを時系列に沿って何らかの順序で依存関係を持たせて並べていったものと考えることができる．そこで以降では，このタスク同士の並び順を決定することを「スケジューリングを行う」と呼ぶことにし，その手順について述べる．

4.2 タスクのグループ分け

ネットワーク中のそれぞれのリンクについて，個々のホスト間の通信を逐次的・排他的に行うためには，同一のリンクを使用する通信同士が確実に時系列で重ならないように実行されれば良い．そこでグループという概念を導入し，タスクをいくつかのグループに分類してグループ内で順序関係をつけることによって，これを実現する．

4.2.1 グループの定義

まずいくつかの定義を行う．はじめに，複数の通信が 1 つのリンクを共通して使用するという状態を具体的に定義する．すなわち，ある 2 つのタスクがリンクを共有しているかどうかは以下のように定義される．

定義 2 (リンクの共有するタスクの存在).

タスク J_{ab} と J_{cd} が存在し，a から b に至るまでの通信経路上のリンクの集合が $E = \{e_0, e_1, \cdots, e_k\}$, c から d に至るまでのリンクの集合が $F = \{f_0, f_1, \cdots, f_l\}$ であるとする．このとき両者の集合に共通の要素が存在する，すなわち $E \cap F \neq \emptyset$ であるならば，タスク J_{ab} と J_{cd} は互いにリンクを共有するタスクである．

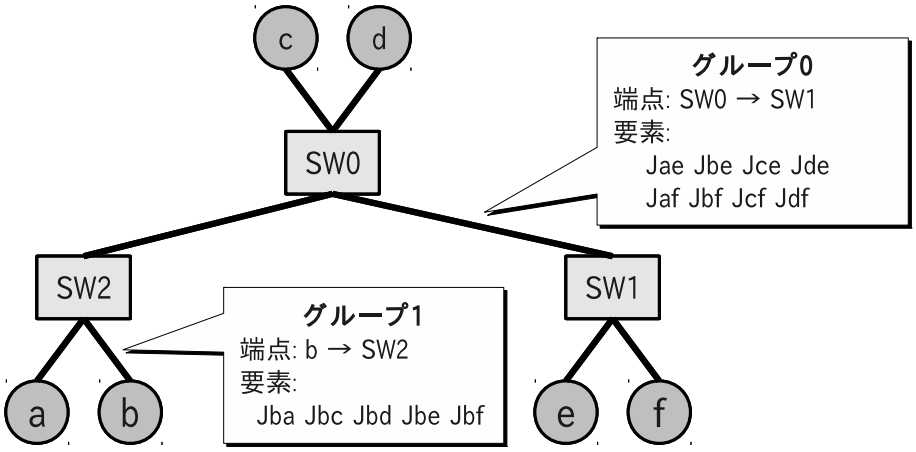

図 4.1: リンクを共有するタスクのグループ化

さらに 1 つのリンクに関して，互いに依存関係のあるタスク同士を分類することを目的として以下のようにグループを定義する．このグループの実体はネットワーク上の一本一本リンクの一方向ごとに定義されるタスクの集合である．

定義 3 (グループ)**.**
あるタスクの集合 $G = \{J_0, J_1, \cdots, J_k\}$ が存在し，G に属するどの 2 つのタスク J_i, J_j を選んでも，両者はリンクを共有するタスクであるとする．さらに，G からどのようにタスクを 2 つ選んだとしても，共有されているリンクのうちにある一つのリンク e_c が含まれているとする．このとき，G は e_c に属するグループであるとし，その要素は $J_0, J_1, \cdots, J_k$ であるものとする．

具体例を挙げる．例えば全対全通信を考えると，図 4.1 のようなネットワークにおいては，8 つのタスク $J_{ae}, J_{af}, J_{be}, J_{bf}, J_{ce}, J_{cf}, J_{de}, J_{df}$ がそれぞれ $SW0$，$SW1$ を端点とするリンクを共有している．このときその共有リンクのグループ 0 の要素が $\{J_{ae}, J_{af}, J_{be}, J_{bf}, J_{ce}, J_{cf}, J_{de}, J_{df}\}$ である，というように表すものとする．同様にして，例えばホスト b とスイッチ $SW2$ を端点とするグループ 1 の要素は $\{J_{ba}, J_{bc}, J_{bd}, J_{be}, J_{bf}\}$ であると表せる．

4.2.2 タスクをグループに分類する意義

このようにしてそれぞれのリンクを共有するタスク同士をグループ振り分けたとき，必ず同一グループに所属するタスク J_{prev} が完全に終了したことを何らかの方法で確認した上で，次のタスク J_{next} が実行されるようにタスクを実行するという順序関係をみたしているのであれば，そのリンクについて言えばコンテンションが発生することがないコンテンションフリーな通信である．

この関係をネットワーク内に存在するあらゆるグループ内で満たすことができていれば，それはそのネットワーク全体でコンテンションフリーな通信となる．すなわち，本研究で目的とするコンテンションフリーな集合通信を実現することができるわけである．なお，上記のタスクが完全に終了したことを確実に確認・検知する方法については後ほど詳細を述べるものとする．

4.3 送信元が同じホストであるタスクの分類

送信元が共通であるタスク J_{ab}，J_{ac} が存在し，それらのタスクがリンク e を共有するタスクであったとき， J_{ab}，J_{ac} はリンク e における同じグループ G に属するものとする．

これは，4.1 に述べた通り，あるホストが同時に 2 つ以上のホストを相手として通信を行うことがないためである．同じ送信元を持つタスク同士は互いに排他的に実行されるため，共有しているリンク上ではお互いに必ず同じグループに所属するものとして考えることが妥当である．

4.4 リンクのバンド幅を考慮したタスクの分類

4.4.1 定義

前節までで述べた各リンクを利用するタスクごとにタスクをグループ分けを行う方法では，一つのリンクの一方向ごとに高々一つのグループしか存在し得ないすることになる．これは各リンクをある時刻に使用できる通信はただ一つだけであるということと同値である．

ところが，現実には複数の小さいバンド幅を持つリンクから，一本の大きいバンド幅を持つリンクにデータが流入する場合はリンクを同時に複数の通信が使用しても同時刻に流入しようとするデータの総量はリンクやそのリンクに接続されている機器の受信容量を超えることはないと考えられる．

これらの状況では，同時にリンクを使用できる通信の数をただ一つに制限してしまっては制約が厳しすぎるために通信性能が逆に低下してしまう可能性がある．したがって，実際にはリンクのバンド幅を考慮した上でタスクを 1 つ以上のグループに分類すべきである．

具体的にはネットワーク上の任意のスイッチ s について，s に接続されているリンク n 本のうち一本を出力 e_{out}，残りのリンクうち k 本が入力 $e_k(0 < k < n)$ とする．このとき，それぞれのリンクのバンド幅を b_{out}，b_k とすれば

$$b_{out} > \sum_{i=0}^{k} b_i \tag{4.6}$$

を満たすならば，b_{out} へ向かう通信においてはコンテンションは発生しないと考えるのが妥当である．その逆も成り立つ．

4.4.2 手順

あるスイッチ S を経由して，バンド幅が B であるリンク e を経路として使用する k 個のタスクが存在するとする．さらに，タスク J_i が S を経由する直前に通過していたリンクのバンド幅を b_i と表すことにする (以降これを入力バンド幅と呼称する) と，リンク e を経路として使用するタスクと入力バンド幅のペア (J_0, b_0)，$(J_1, b_1), \cdots, (J_k, b_k)$ が存在することになる．ここで，簡単のため，$b_0 \geq b_1 \geq \cdots \geq b_k$ であるとする．

この条件のとき，リンク e を経路として使用するタスクをコンテンションフリーなグループに分類する手順は次の通りである．

1. J_0 を新しいグループ G_0 の要素とする．さらに，バンド幅 B の値から J_0 の入力バンド幅 b_0 の値を減じておく．減じた値を C_e とする．

2. J_1 ,$\cdots$, J_k をこの順に一つずつ選択する．選択されたタスク J_i について，もし入力バンド幅 b_i が C_e の値よりも小さければ，そのタスク J_i を新しいグループ G_{new} の要素とし，C_e の値から b_i の値を減じる．そうでなければ，タスク J_i は G_0 の要素とする．

3. ただし，もし J_i の送信元のホストと同じ送信元ホストを持つタスクが $J_0, \cdots, J_{i-1}$ の中に存在した場合 (そのタスクを J_x とする)，J_i は入力バンド幅の多寡に関わらず，J_x と同じグループの要素となる．

以上のステップをまとめた擬似コードが図 4.2 のようになる．

このグループ分けの手順の結果，1 本のリンクあたり最低 1 つのグループが生成されることとなる．全体のネットワークとリンク，グループおよびタスクの間の従属関係を表すと，図 4.3 のようになる．ネットワークには複数のリンクが存在し，各リンクに複数のグループがそれぞれ所属する．タスクは複数のグループに所属することになるが，同一リンクに属する複数のグループに対して所属することは無い．

コンテンションフリーな通信を行うためには，各々のグループの要素となっているタスクは各々のグループ内のタスク同士で排他的に通信が実行される必要がある．ところが定義及び上記手順の結果より，それぞれ別々のグループの要素となっているタスク同士が仮に同時刻にリンク e を使用したとしても，入力のバンド幅の和がリンク e のバンド幅を超過してしまうことはあり得ないためコンテンションが発生することは無い．よって，別々のグループに所属しているタスクはそれぞれ独立にスケジュールを行って良いと言える．

4.5 通信データ量を考慮したグループ分け

一つのリンクについてただ一つのグループのみが存在するとしてしまう場合に通信性能が返って悪化すると考えられるケースはもう一つある．通信されるデータの量が非常に小さい場合がそれで

```
setup_group_big_message (Edge e, Task J_0, ··· , J_k) :
  for (J_i in J_0, ··· , J_k) :
    /* in case there is no group */
    if (no_group_exists()) :
      G_0 = make_new_group();
      add_task_to_group(G_0, J_i);
      C_e = C_e − b_i;
    else :
      if (same_source_host_has_already_exists(J_0, ··· , J_{i−1}, J_i)) :
        G_s = get_group_same_source_host(J_0, ··· , J_{i−1}, J_i);
        add_task_to_group(G_s, J_i);
      else if (C_e ≥ b_i) :
        G_new = make_new_group();
        add_task_to_group(G_new, J_i);
        C_e = C_e − b_i;
      else :
        add_task_to_group(G_0, J_i);
```

図 4.2: 入力バンド幅を考慮したタスクのコンテンションフリーなグループ分け

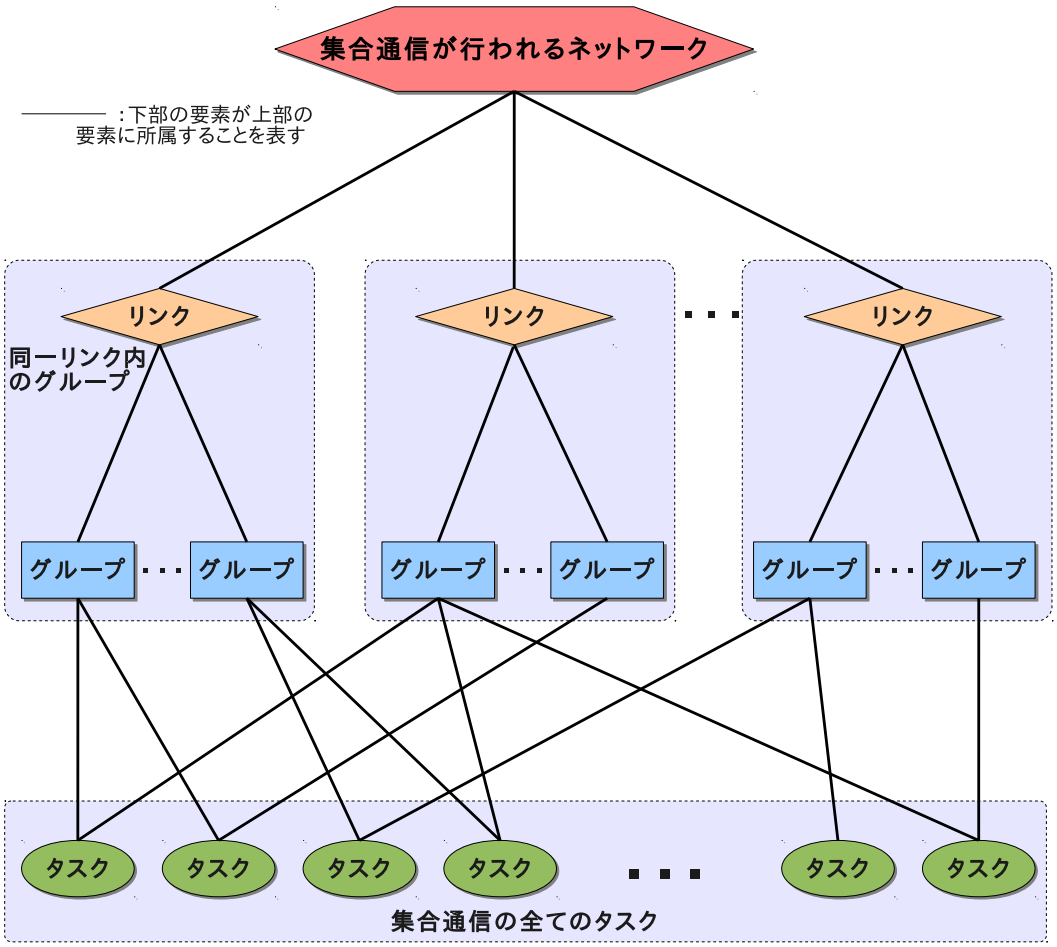

図 4.3: ネットワーク内のリンク，グループ及びタスク間の関係

```
setup_group_small_message (Edge e, Task J_0, ··· , J_k) :
  for (J_i in J_0, ··· , J_k) :
    if (same_source_host_has_already_exists(J_0, ··· , J_{i-1}, J_i)) :
      G_s = get_group_same_source_host(J_0, ··· , J_{i-1}, J_i);
      add_task_to_group(G_s, J_i);
    else:
      add_task_to_group(G_0, J_i);

setup_group (Edge e, Task J_0, ··· , J_k) :
  /* in case all massages through the link is small */
  if (sum_of_all_messages(J_0, ··· , J_k) < edge_width(e) * edge_length(e)) :
    setup_group_small_message(e, J_0, ··· , J_k);
  else :
    setup_group_big_message(e, J_0, ··· , J_k);
```

図 4.4: 一つのリンクに対するタスクのコンテンションフリーなグループ分け

あり，リンクの遅延とバンド幅の積，すなわち帯域遅延積に比べて，リンクを同時刻に通過するデータ量の総量が非常に少ないならば，リンクや接続されている機器の容量を超えることは無いはずである．そのため，このような状況ではコンテンションが発生する可能性はごく小さいと考えられる．つまり．あるリンク e_{out} の遅延 L_{out}，バンド幅 B_{out} に対し，M byte のメッセージが k 個同時に通過しうるとき，すなわち

$$\frac{kM}{B_{out}} < L_{out} \quad \Leftrightarrow \quad kM < L_{out}B_{out} \tag{4.7}$$

を満たすとき，e_{out} を同時に使用する通信は e_{out} においてはコンテンションは発生しないものと考えるべきである．

ところが，一般にあるスイッチに n 本のリンク接続しているとき，そのスイッチに対してデータが流入する組み合わせは 2^n 通りである．したがって，タスクごとに通信するデータ量が異なることまで考慮すると，グループの組み合わせを計算する際の計算量は $O(2^n)$ となり，現実的ではない．

そこで本アルゴリズムでは簡単のため，あるリンクを経路として使用する全てタスクの通信データ量の和がそのリンクの帯域遅延積よりも小さい場合は，タスクの数だけグループを生成し，一つ一つのグループはただ一つのタスクを要素として持つようにする．これは，あるリンクを通過するデータが全て非常に小さい場合は，それらのデータが仮に一斉に同時に通過したとしてもコンテンションが発生することはないために，タスク同士に順序関係を設ける必要がないということを意味している．

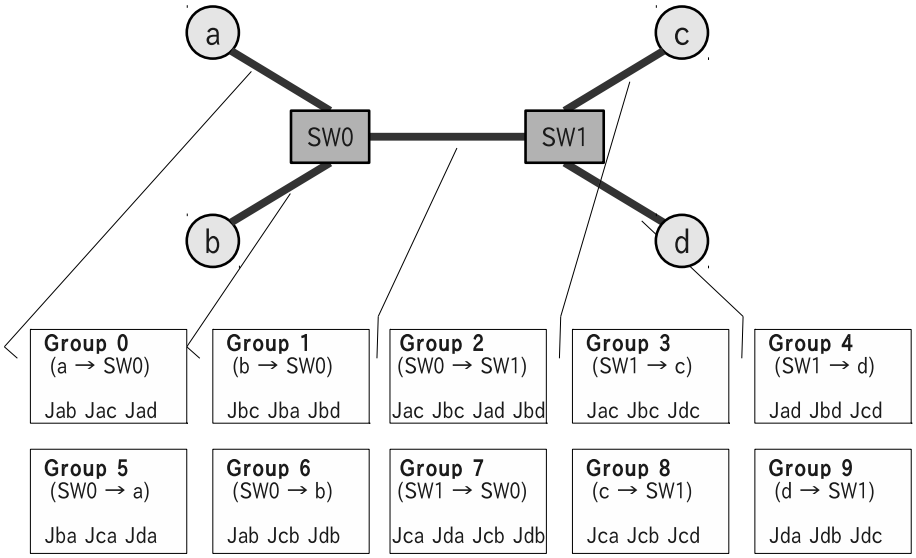

図 4.5: 4 ホスト，2 スイッチの環境における全対全通信のタスクのグループ分け

逆に，データ量の和が帯域遅延積を超過する場合は，4.4 の手順に基づいてグループ分けを行うものとする．以上に述べたタスクのグループ分けの手順ををまとめると，4.4 のような擬似コードとなる．

ここで，実際に 4.4 のプログラムを図 4.5 のような 4 ホストに 2 つのスイッチが存在しているネットワークで全対全の通信を行った場合に対して適用した場合を考える．この場合のタスクの個数は明らかに $4 \times 4 = 16$ 個である．仮に 1 つのリンクの 1 方向についてグループが 1 つずつ構成されたとすると 10 個のグループができ，タスクがそれぞれ図 4.5 のようにグループに所属することになる．

これまで述べたようにこれら各々の 10 個のグループに所属しているタスク同士が排他的に実行されるようにそれぞれ順序関係を持つことになる．1 つのタスクは複数のグループに所属しているので，各グループ内での依存関係を考慮した全タスク間の順序関係のグラフの一例は，図 4.6 のようになる．グループの定義より，タスクが図 4.6 の左端から右端に向かってこの順番で一つずつ実行されていく限り，図 4.6 で表された全対全通信は必ずコンテンションフリーな要件を満たす通信である．

一方で，それぞれのグループ内でのタスク同士の順序関係の付け方によって，図 4.6 のような全タスク間の依存関係に関するグラフの形状も変化し，全タスクの実行に要する時間も変化する．したがって，各々のグループ内でのタスクの順序関係の付け方を工夫することで，集合通信の総実行時間を短くすることが可能である．そこで次章では，このようにグループ分けされたタスクのスケジューリングを上手く行うことで，効率の良い集合通信を実現するための手順について述べる．

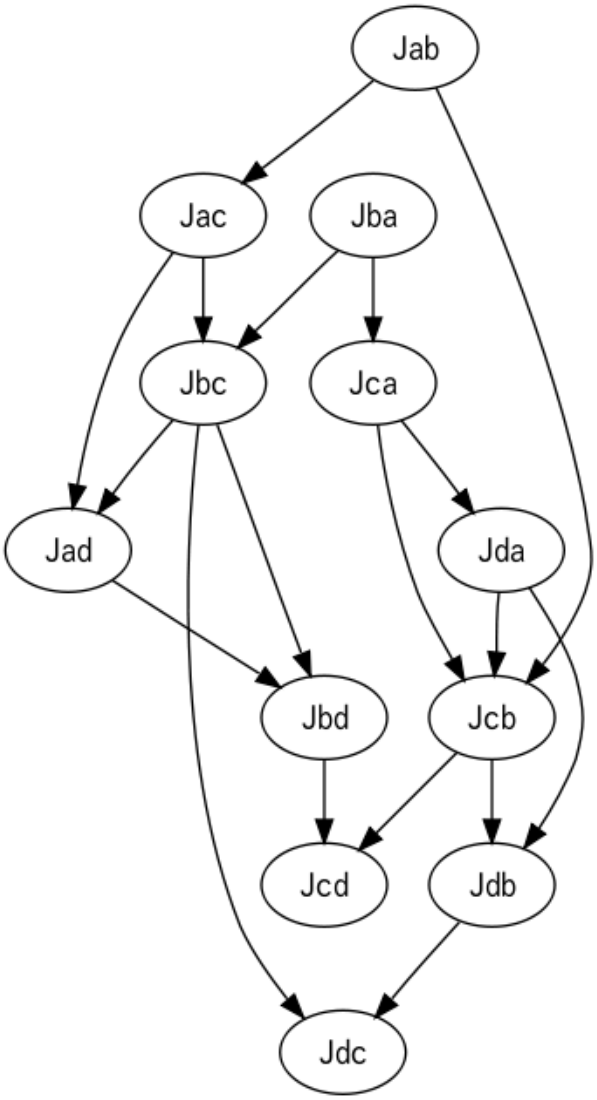

図 4.6: 全対全通信での全てのタスク間の依存関係

第5章　集合通信のスケジューリング

本章では，前章で紹介したコンテンションフリーな通信を行うフレームワークを，実際の集合通信に適用する方法について述べる．さらに，1つの集合通信の総実行時間をなるべく短くするようなヒューリスティックについても考察する．なお以降では，タスク同士に順序関係を設定していく作業のことを「スケジューリング」と呼ぶことにする．

5.1　スケジューリングの概要

k 個のタスク J_0，$J_1, \cdots, J_{k-1}$ が n 個グループ G_0，$G_1, \cdots, G_{n-1}$ に所属しているとする．スケジューリングの最終的な目標は J_0，$J_1, \cdots, J_{k-1}$ 全体がコンテンションフリーな通信となるようにタスク間の依存関係を設定することであり，そのためには各グループ G_i ごとに，その要素となっているタスク同士に順序関係を定めればよいことになる．そこでまず，スケジューリングのアルゴリズムに必要となる値について定義する．

一つ目はタスクの実行に要するコストについてである．送受信のオーバヘッド等を無視できるとしているので，タスクの実行コストはホスト間の通信時間と一致する．よってホスト間の遅延が L_{xy}，バンド幅が B_{xy} なる2つのホスト X から Y へ M_{xy} byte の通信を行うタスク J_{xy} におけるタスクのコスト C_{xy} は，

$$C_{xy} = L_{xy} + \frac{M_{xy}}{B_{xy}} \tag{5.1}$$

であるとする．

二つめはそれぞれのグループ G_i に関してである．各グループは，それぞれ既に順序関係が設定済みなタスクの線形リスト S_i と，まだスケジューリングがなされておらず順序関係が設定されていないタスクの集合 U_i を持っているものとする．S_i にタスク $J_0, J_1, \cdots, J_{k-1}$ がこの順序でリストされていた場合，タスク $J_0, J_1, \cdots, J_{k-1}$ は $J_0 \to J_1 \to \cdots \to J_{k-1}$ なる全順序関係を持つとする．すると，グループ G_i の要素となるタスクは必ず S_i か U_i のどちらかに含まれていることになる．なお，スケジューリングの作業が一つも行われていない段階では，全てのタスクは U_i に含まれているものとする．

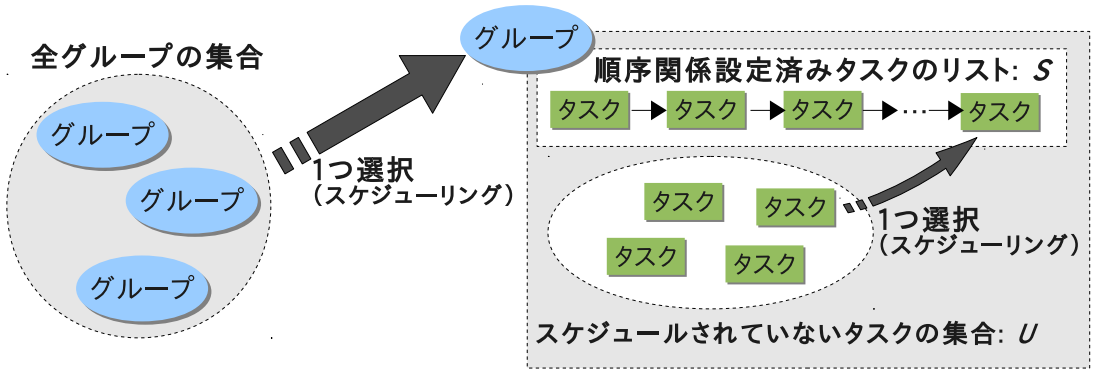

図 5.1: スケジューリング 1 ステージのイメージ

5.2 スケジューリングのステップ

タスクに順序関係を設定していくスケジューリングの作業の概要は，次のような 5 段階となる．ここで以下のリストの (1) から (3) までの一連のステップを，便宜的に 1 ステージと呼ぶことにする．

1. 全グループの集合 G の中から，$U_i \neq \emptyset$ なるグループ g_i を 1 つ選ぶ (select_group).
2. g_i 内の U_i から，$J \in U_i$ なるタスク J を一つ選び，リスト S_i の末尾に付け加える (select_task). その後同様に，J がその要素となっている残りのグループ g_j 内の U_j から J を取り出し，S_j のそれぞれ末尾に追加する．
3. タスクの移動が行われたグループ g について，$U = \emptyset$ であれば，g を G から取り去る．この結果 G が空となったとき，全てのタスクがスケジュールし終わったことを意味し，スケジューリングの作業を終了する．そうでなければ，再び (1) へと戻る．

この一連のスケジューリング 1 ステージのイメージは 5.1 のようになっている．以降で上記中の select_task および select_group の手順について詳しく述べる．

5.3 グループの選択

全てのタスクをスケジュールし終わったときの，4.6 のような依存関係を示すツリーの深さは，その集合通信の実行時間と比例する．依存関係のツリーの深さが浅いということは，全タスクの実行開始から終了までの時間が少なくなることを意味し，結果的に集合通信の総通信時間が短くなるからである．

そこで select_group の操作の際に選択されるグループは，全グループの集合 G に存在するグループ g_i の中で，グループの残りコストが最も大きいものを選ぶものとする．ここでのグループの残り

コスト R_i とは，g_i 中の U_i に k 個のタスク $J_0, \cdots, J_{k-1}$ が存在するとすれば，次のように計算される量である．

$$R_i = \sum_{j=0}^{k-1} C_j = \sum_{j=0}^{k-1} \left(L_j + \frac{M_j}{B_j} \right) \tag{5.2}$$

定性的には，ある時点で R_i が大きいグループは，まだ通信が行われるべきタスクが大量に残っていることを意味する．そこで select_group ではそのようなグループを優先的に選択していくようにすることで，各グループが均等にタスクのスケジューリングを行えるようにするため，結果として全体の依存関係のツリーの深さを浅く保つことができると考えられる．

5.4 タスクの選択

select_task 操作は select_group で選ばれたグループに属するタスクを 1 つ選ぶことである．ここで，まず select_task でどのタスクを選べば適切かを議論する前に，実際に通信を行うにあたってタスク同士の順序関係を確実に満たした上で排他的に通信を行うための機構について述べる．

5.4.1 確実にタスク間の順序関係を満たした通信を行う方法

同一のグループに所属しているタスク J_x と J_y において，$J_x \rightarrow J_y$ なる順序関係が存在するとき，コンテンションフリーな通信を行うには J_y の通信は J_x の通信が完全に終了してから実行される必要がある．

そこで，このタスクが完全に終了したことの周知・検知に前述した同期の機構を用いることにする．2 つのホスト間で同期を行う方法は単純で，図 5.2 のように先行する通信の受信ホスト Z が送信ホスト X からのデータを全て受け取り終わると同時に，後続の通信の送信ホスト Y に対して，1 byte のパケットを送信する．Y はこの Z からのパケットの受信によって，先行する X から Z への通信が終了したことを知ることになる．逆に，それ以外の方法で Y が $X \rightarrow Z$ の通信終了を知る手段は存在しない．

一般には，ネットワークに関する前提より 1 つのタスクにおける通信経路は複数のリンクまたがることになるため，select_task で選ばれるタスク J は複数の異なるグループ $g_0, \cdots, g_{n-1}$ の要素となるはずである．$g_0, \cdots, g_{n-1}$ においてそれぞれ直前に実行されているいるタスク，すなわち各グループ g_i 中の S_i の末尾に存在しているタスクを $J_0, \cdots, J_{n-1}$ とすれば，J は $J_0, \cdots, J_{n-1}$ との間で $J_0 \rightarrow J, \cdots .J_{n-1} \rightarrow J$ なる最大 n 個の順序関係を持つことになる．このような複数の順序関係が存在するときは，直前に先行して実行されているタスク $J_0, \cdots, J_{n-1}$ の通信の受信側となっている全てのホストから J の送信側のホストが同期のパケットを受信し終わらない限り，J の通信を開始することができないものとする．

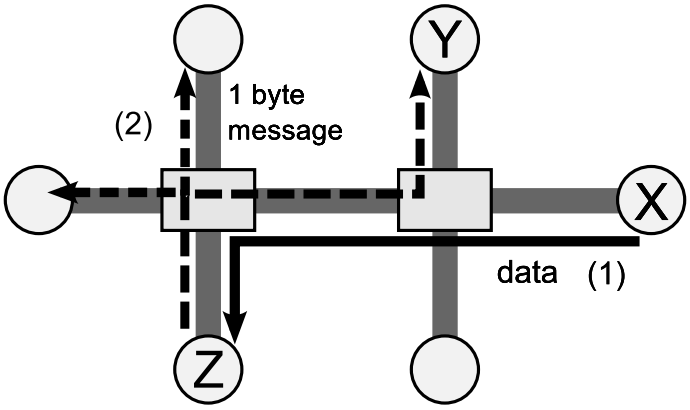

図 5.2: データ受信後に複数ホスト間での同期を取る機構

5.4.2 同期に要するコスト

2 つのタスク同士の同期に必要なコスト (時間) は，先行するタスクの受信ホスト Z と後続のタスクの送信ホスト Y との間の遅延 L_{xy} に相当する．同期作業により発生するパケットはごくごく小さいサイズなので，が集合通信として行われている本来の通信に対して影響を与えたり，コンテンションを発生させたりする可能性は非常に低く，無視できるものとする．

ただし，先行するタスクの送信ホストと後続のタスクの送信ホストが同じ場合 ($X = Y$ であった場合) は，2 つのタスクの間で同期を取る必要性がない．そこで，このような場合は同期に要するコストが 0 であるものとみなす．

5.4.3 タスク選択のポリシー

そもそもある 1 つのグループの要素となっているタスクのコストの総和は常に不変である．したがって，select_task の方法によって集合通信の総通信時間に影響を与える要素は，タスク同士の同期に要するコストの多寡のみである．そのため集合通信の総通信時間を短くすることを考えた場合，それぞれのグループについて select_task で選ばれるタスクは，直前に実行されているタスクとの間の同期に要するコストが最も小さいものであれば，もちろんこの方法でも最適解が得られるわけではないが，総通信時間を短くする上でよい近似解が得られると考えられる．

5.4.4 タスクの選択手法が総通信時間に与える影響

グループ内におけるタスクの選び方が，集合通信全体の実行時間にどれほどの影響を与えるかについて検証するため，シミュレーションを行った．

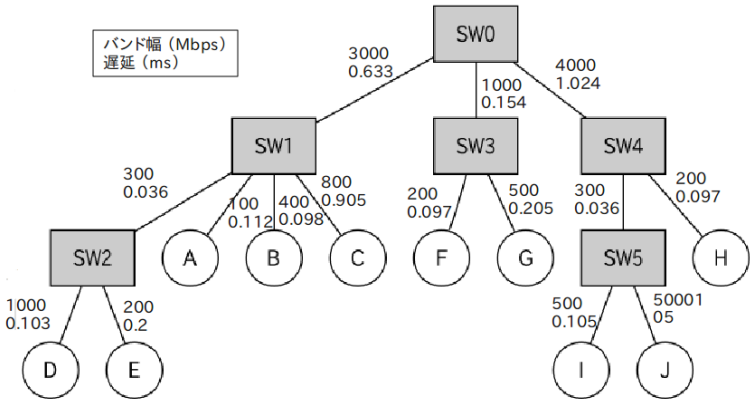

図 5.3: シミュレーションに利用するネットワーク構造

シミュレーションの環境として図 5.3 に表したようなトポロジーを持つネットワークを仮定し，全対全のホスト間で通信を行う alltoall 操作を行った場合の，集合通信の開始から終了までに要する時間を算出した．その他のシミュレーションの条件を以下に列挙する．

- 図 5.3 中の A から J までは通信を行うホストであり，$SW0$ から $SW5$ まではスイッチである．
- それぞれの通信をタスクとみなして 5.2 にあるスケジューリングのステップに基づいてタスクに順序関係を割り当てる．この割り当てが終了したのち，ホスト間の遅延やバンド幅の情報を用いてタスクの通信が実際に行われたと仮定したときの実行時間を追跡してゆき，初めのタスクの実行開始から最後のタスクの実行が終了する時刻を算出したものを集合通信の総通信時間であるとする．
- タスクの選択方法によらず，どの方法もコンテンションフリーな条件を満たしながら通信を行うものとする．
- ステップ 2 におけるグループの選択方法は 5.3 に準ずる．
- 各ホストが別の 1 ホストに対して送信するデータサイズは 100 byte から 1 MB まで変化させてそれぞれのタスクの選択方法が総通信時間に与える影響を検証する．

このシミュレーションでは，すなわち，各ホストは自分以外の全てのホストと 1 回ずつ通信することになる．タスクの選択方法としては，次に述べる 5 通りの方法を用いた．それぞれグループのスケジュールされていないタスクのリスト U から下の手順でタスクを選ぶものとする．

1. ランダムに選択する (Random)
2. 最も大きなコストを持つタスクを選択する (Max Cost)
3. 最も小さなコストを持つタスクを選択する (Min Cost)
4. 直前に実行されたタスクとの同期に要するコストが最も大きいタスクを選択する (Max Sync)

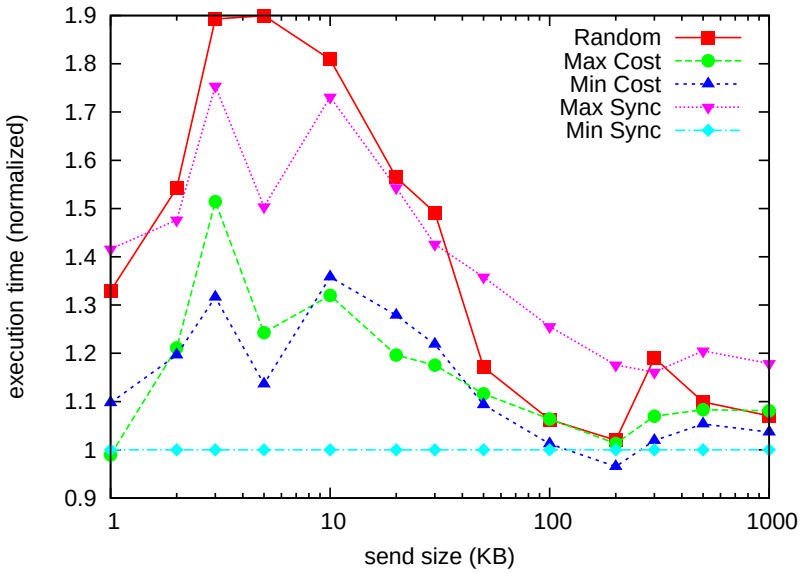

図 5.4: タスクの選び方が総実行時間に与える影響

5. 直前に実行されたタスクとの同期に要するコストが最も小さいタスクを選択する (Min Sync)

このようにしてシミュレーションを行った結果を図 5.4 に示す．結果は Min Sync の場合の総通信時間を 1 として，残りの 4 手法の総通信時間との比率を求めたものである．

結果を見ると，各々のホストが送信するデータのサイズが 100KB を超える領域では．手法間で大きな違いは生じていない．これは，データ長が大きく同期にかかるコストがデータ自身の通信時間 (= タスクのコスト) に比べて小さく無視できるためである．

一方，2KB から 200KB までの 1 ホストあたりの送信データサイズが比較的小さい場合では Min Sync の方法と比べて最大でおよそ 90%総通信時間が増加している．この違いは同期に要するコストの大小に起因しており，特にコストが比較的大きくなりがちな Max Sync や Random の手法の場合には特に顕著である．したがって，Min Sync の方法のように，select_task のステップでは同期のコストが最小となるタスクを選ぶことの意義が大きいことを確認することができる．

5.5 ホスト間でのデータ転送

5.5.1 データ転送を行うことの意義

select_task において同期コストが小さくなるようにタスクを選択することも必要であるが，同様に重要なのが 3.7 で示したような第三者のホストにデータを転送することでそもそもの同期回数自体

を削減することで集合通信の通信時間の短縮を目指すことである．

ところで，ホスト X から Z への通信を Y を経由して行うようにした場合，ホスト間でデータの転送を行うことはタスクを分割することに相当する．言い換えると，データの転送はタスク J_{xz} を J_{xy} と J_{yz} の 2 つに分割したものに相当するとみなすことができる．このようにして分割された 2 つのタスクは次の 2 つの性質を持つことになる．

- Y は X からのデータが到着しない限り Z に対してデータを送信することはできない．すなわち J_{xy} と J_{yz} は $J_{xy} \rightarrow J_{yz}$ なる順序関係を持つ．
- Y はデータを X から受け取り次第 Z へ向けて順次送信できるものとする．J_{yz} の通信が開始できる時刻は J_{xy} の通信の先頭，すなわちホスト X から出発したデータ列の先頭が Y に到達した時刻であるとする．
- タスクの分割が行われた際には，元々のタスクは所属するグループから取り除かれ，あらたに分割後のタスクをそれぞれが所属すべきグループの要素とする．どのグループの要素となるかを解析する手順は 4 章に述べた手順とする．

5.5.2 タスク分割が可能な条件

そこで，仮にタスク J_{xz} を J_{xy} と J_{yz} と分割したとき，同期に要する時間も含めて本来 J_{xz} の実行にかかっていた時間と，2 つに分割することで実行にかかる時間を比べて，もし分割した方が実行時間が少なくなるときには，データを転送することが有効であると考えられる．

単純にデータを転送すると言っても，どのホストからどのホストへと転送していくかの組み合わせの計算量は $O(n!)$ であり，全ての場合について転送した場合の実行時間を計算することは現実的ではない．そこで，転送を行うホストの候補に制限を設けることにする．そこでまずそれに先立ち，独立なタスクという概念を導入する．

定義 4 (独立なタスクとその性質)**.**

ある 2 つのタスク J_x と J_y が存在し，それぞれのタスクにおける通信の経路上のリンクの集合を E_x，E_y とする．このとき，$E_x \cap E_y = \emptyset$ であれば，J_x と J_y は互いに独立なタスクであるとする．

さらに互いに独立なタスクであれば，グループの定義よりそれぞれのタスクが同一のグループの要素となることは無く，そのため 2 つのタスク間で同期に要するコストがかかることは全く無い．

この定義のもとでは，ある 2 つのタスクについて以下の定理が成り立つ．

定理 1 (タスクの独立性)**.**

ネットワークがツリー構造をなしているか，もしくはホスト間の通信が必ず一意な最短経路を通るように行われるならば，ホスト X から Y への通信を行うタスク J_{xy} とホスト Y から Z への通信を行うタスク J_{yz} は互いに独立なタスクである．

証明 1.

ホスト X から Y への通信を行うタスク J_{xy}，経路上のリンクの集合を $E_{xy} = \{e_0, \cdots, e_k\}$， Y から Z への通信経路のリンクを $E_{yz} = \{f_0, \cdots, f_l\}$ とする．仮に 2 つのタスクが独立でないとすると，定義より $E_{xy} \cap E_{yz} \neq \emptyset$ であり，$e_i = f_j \equiv C$ なるリンクが 1 つ以上存在することになる．

ところで，ネットワークがツリー構造をなしているか，もしくはホスト間の通信が必ず一意な最短経路を通るように行われるならば，そのネットワーク中の任意の 1 本のリンクを取り去ることでネットワーク上の全ホストを別々の 2 つの集合に分割することが可能である．リンク C を取り去ることによってネットワークを $S_0 \cap S_1 = \emptyset$ を満たす S_0 と S_1 の 2 つのホストの集合に分離するとする．通信経路上のリンクが取り去られているので，明らかに X と Y および Y と Z もそれぞれが互いに別々のホストの集合に属することになるが，ここで通信の方向を考慮すると例えば $X \in S_0$ かつ $Y \in S_1$ とするならば，$Y \in S_0$ かつ $Z \in S_1$ となる[1]．しかしこれは $S_0 \cap S_1 = \emptyset$ に反するため，はじめの仮定が誤りであることを意味する．よって 2 つのタスク J_{xy} と J_{yz} は独立であることが示された．

以上を踏まえ，本アルゴリズムでは同一グループに所属するタスク J_{xy} と J_{wz} が存在し，直前にスケジュールされているタスク J_{xy} がホスト X から Y への通信を行うものであり，今まさにスケジュールするために選択されたタスク J_{wz} が W から Z への通信を行うものであったとしたとき，その場合転送が行われる経路の候補は

$$W \to X \to Y \to Z$$

のみであると制限する．ただし次の要件が存在する．

- X と W は異なるホストである．
  なぜならそもそも送信ホストが互いに同じ 2 つのタスクは 5.4.2 より同期に要するコストは 0 なので，わざわざ転送する必要は無いからである．
- W と Y は異なるホストである．
  これは仮に W と Y が等しいとすると，定理 1 より J_{xy} と J_{wz} は独立なタスクとなってしまい，同一グループに属しているはずではなくなってしまうからである．
- Y と Z は同じホストであっても異なるホストであっても良い．
  同じホストの場合は $W \to X \to Y$ というように転送が行われることになる．

このとき，分割された 3 つのタスクについて次の性質が成り立つ．

定理 2 (分割されたタスクの独立性)**.**

同一グループに所属するタスク J_{xy} と J_{wz} が存在し，直前に J_{xy} がスケジュールされているとする．ここで J_{wz} を転送のために J_{wx}, J_{xy}, J_{yz} の 3 つに分割したとき，分割された 3 つのタスクは互いに独立である．

[1] $X \in S_1$,$Y \in S_0$ としても同様

証明 2.

定理 1 より，J_{xy} と J_{wx} および J_{xy} と J_{yz} はそれぞれ互いに独立である．よって，J_{wx} と J_{yz} の 2 つが独立であることを示せばよい．

ここで，元のタスク J_{xy} と J_{wz} 同一グループに属しているので，経路上に共通のリンクが存在し，各ホストの位置関係は図 5.5 のようになっているはずである．すると，共通のリンクでネットワークを分断したとき，ホスト X と W および Y と Z はそれぞれの組が同じホストの集合に属した状態で，2 つの組は別々のネットワーク上に存在することになる．そのため明らかに W から X へと Y から Z へは共通するリンクは存在することはない．したがって定義 4 より J_{wx} と J_{yz} もまた独立である．よって題意は示された．

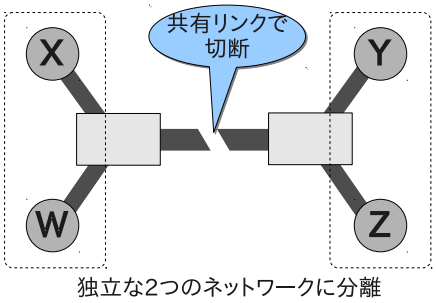

図 5.5: タスク分割が行われたときのホストの位置関係

定理 2 が意味することは，データの転送により分割されたタスク J_{wx}, J_{xy}, J_{yz} は，各々が異なるグループに所属するため相互に同期を取る必要性が無いということにある．また，直前にスケジュールされていたタスク J_{xy} と J_{wz} を分割後に新たにできたタスク J_{xy} は明らかに同じグループに属するが，送信ホスト X が同じなのでこちらもタスク間の同期を取る必要がない．したがって結果としてデータを転送することよって同期に要するコストを削減できる可能性が生じるわけである．

5.5.3 再帰的なタスクの分割と分割回数の制限

分割によって生じた 3 つのタスクを見てみると，2 番めのタスクは送信ホストと受信ホストは必ず一致するが，1 番目のタスク J_{wx} と 3 番目のタスク J_{yz} はさらに 5.5.2 の条件にしたがって再帰的に分割できる可能性がある．

ところが再帰的に分割していくと言っても，分割候補を求める探索空間が広くなりすぎるために実際は無制限に分割できるわけではなく，何らかの制限を設ける必要がある．そこで具体的には，一連の再帰的なタスクの分割によって生成されていくタスクが所属するグループが重複してはならな

いという制限を設ける．タスクを分割していくうちに分割によって生成されるタスクが所属するグループに重なりが出た時点で，タスクの分割を打ち切るものとする．

グループの定義より，所属するグループが重複するタスクは共通するリンクを持つことを意味する．5.5.2 の条件のもとでタスク J を $J_0, \cdots, J_{k-1}$ と分割していったとき，$0 \geq i < j < k)$ なる J_i, J_j の 2 つのタスクが所属するグループが重複していたならば，これは同一のリンクを 2 回同じデータが通過することになり直感的に非効率であるし，データの転送経路がループ状になってしまっていることを意味する．そのため，タスクの分割は重複するグループがでてきた時点で打ちきるものとする．

5.5.4 転送の有無での通信コストの見積り

実際にそのタスクをスケジュールするにあたっては，そもそもタスク分割を行うべきではないのか，行う場合はどこまで分割するのが妥当であるかを決定する必要がある．具体的には，スケジュールしようとしているタスクで送信されるデータの末尾が出発する時刻の大小を指標とすることにする．

ホスト H_k から H_{k+1} への通信を行うタスクが属するグループの集合を $G = \{g_0, g_1, \cdots, g_{n-1}\}$ とし，この集合の中で最も時刻が進んでいるグループを g_{max} とする．ここで，グループの時刻とは次のように定められる量である．

定義 5 (グループの時刻).

グループ g のスケジュール済みタスクのリストを S とし，リスト内のタスクがこの順に $J_0, J_1, \cdots J_{k-1}$ とする．さらに．2 つのタスク J_i, J_j 間の同期のコストを $Sync(J_i, J_j)$ とする．このとき，グループ g の時刻 T_g を

$$T_g \ = \ \sum_{i=0}^{k} C_i \ + \ \sum_{i=0}^{k-1} Sync(J_i, J_{i+1}) \tag{5.3}$$

と定義する．

g_{max} の時刻を $T_{g_{max}}$ とし，さらに g_{max} で直前にスケジュールされていたタスクが H_x から H_y への通信であったとする．このとき転送による依存関係が存在しない場合にホスト H_k から H_{k+1} へのデータの末尾が出発できる時刻 $\alpha_{k,k+1}$ は，グループの時刻にタスク間の同期のコストとタスクのコストを加えた

$$\alpha_{k,k+1} = T_{g_{max}} + L_{H_y H_k} + \frac{M_k}{B_{k,k+1}} \tag{5.4}$$

と表すことができる．

一方で．転送による依存関係が存在し，$H_{k-1} \rightarrow H_k \rightarrow H_{k+1}$ のようにデータが転送されているとき， H_k から H_{k+1} へのデータの末尾が出発できる時刻 $\beta_{k,k+1}$ は，H_k から H_{k+1} の通信が行われた場合のデータの末尾が出発する時刻と，転送の前段のホスト H_{k-1} からのデータの末尾が H_k に到着する時刻 $\gamma_{k-1,k}$ のうち，大きい方を取ったものである．よって，$\beta_{k,k+1}$ および $\gamma_{k,k+1}$ は，それぞれ

$$\beta_{k,k+1} = \max\{\alpha_{k,k+1},\, \gamma_{k-1,k}\} \tag{5.5}$$

$$\gamma_{k,k+1} = \beta_{k,k+1} + L_{H_k H_{k+1}} \tag{5.6}$$

なる漸化式に表すことができる．

これまでと同様に同一グループに所属するタスク J_{xy} と J_{wz} があり，J_{xy} がそのグループで直前にスケジュールされているとすれば，転送を行わない場合のデータの末尾が出発可能となる時刻は α_{wz} であり，転送を行う場合は β_{wz} である．上の 3 式から両者の値を計算し，$\alpha < \beta$ であれば転送を行わず，そうでなければ転送を行えば良い．

転送を行う場合は，さらにどこまでタスクを分割していくかを求める必要がある．そこでも同様に 3 つの式を用いてタスクの分割候補ごとに β の値を計算する．この値が最も小さくなる場所までタスクを分割することになる．

5.5.5 転送による総通信時間の短縮効果

転送を全く行わない場合[2]に比べ，総通信時間が短くなることが見込まれる場合に転送を行うとき[3]にどれほど総通信時間の短縮に効果があるかに関して調査するため，再び図 5.3 のネットワークを用いて集合通信の総実行時間のシミュレーションを行った．

今回シミュレートした集合通信は alltoall と gather の 2 種類である．gather に関しては図 5.3 中のホスト A をルートとして gather 操作を行っている．転送なしと転送ありの場合とでタスクの選択方法は同じものを用いることとし，5.4.3 の通りタスク間の同期コストが最小となるタスクを選んでスケジュールしていくものとする．転送ありの場合は，5.5.4 に述べた判断基準を元に，転送を行うかどうか，行う場合はどのようにデータを転送するかを決定するものとする．

以上の条件で行われたシミュレーションの結果が図 5.6 である．結果はそれぞれの集合通信について「転送ありの場合の総実行時間 ÷ 転送なしの場合の総実行時間」として比率を求めている．これを見ると，alltoall に関しては転送を行う場合と行わない場合とでほとんど実行時間に差は生じておらず，データの転送自体がほとんど行われていないと判断できる．実際，このシミュレーションでは全通信中でデータの転送に起因する通信の割合は 0.5%から 5%程度にとどまっていた．一方で gather に関しては転送を行うことで最大で約 30 %にまで総実行時間を短縮することができており，データを転送することの有効性が認められる．

alltoall のように 1 つのホストあたりの通信量が多いとき，あるホストがデータを転送してもらおうと本来の宛先のホストではない別のホストにデータを転送してしまうと，転送を請け負ったホストの通信量がさらに増加し，総実行時間を直接増大させる要因となってしまう．ところが，gather の

[2]以下「転送なし」
[3]以下「転送あり」

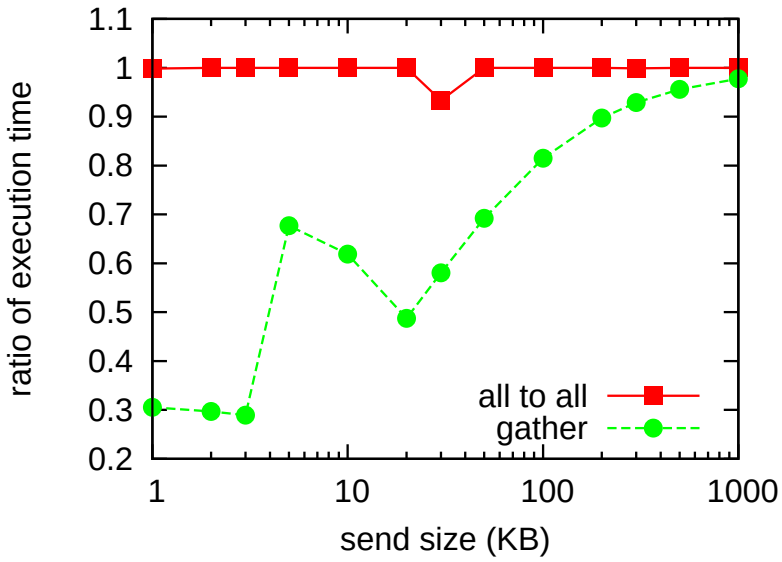

図 5.6: データ転送の有無が集合通信の総通信時間に与える影響

ようにルートのホスト以外のホストはルート宛のデータを送り終わってしまえば後は何もデータを通信する必要がないような通信では，ルートホストが全部のデータを受信し終えるまで他のホストは遊休状態となる．そのためそういったホストがいくらかの転送を請け負ったとしても，総通信時間を直接増大させるには至らず，逆に同期の数を減らして同期時間を削減することで，総通信時間の短縮する効果が大きいものと考えられる．

第6章　実環境での実装と性能評価

6.1　アルゴリズムの実装

第5章までに述べたアルゴリズムを実際に実装していくつかの評価を行った．アルゴリズムは全てC言語で実装し，ホスト間の通信は断りがない限りソケットを用いてTCPで行っている．なお，今回用いた環境ではFirewallなどソケットを用いて完全にコネクションが張れない環境は存在せず，少なくとも一方向からのconnect()が成功する環境である．

実装したプログラムは大きくスケジューラ部と通信機構部の2つに分かれている．アプリケーション起動時にそれぞれ一つずつ別々のスレッドを立ち上げて2つの機構の作業をそれぞれ割り当てるものとする．

スケジューラ部はこれまでに紹介してきた集合通信のスケジューリングを担当する．スケジューリングには通信メッセージ量のパラメタが必要であり，通信メッセージの量はAPIの呼び出しごとに毎回異なる可能性があるから，基本的にスケジューリングは集合通信のAPIが呼び出されるたびに全部のホストで毎回同じように実行される．通信機構部はスケジューラが作成したタスク同士の依存関係を表す有向グラフを受け取り，このグラフに沿って通信を行う．具体的には，次のような動作をする．

- 最初に同期を待つ必要のないタスク(グラフの根の部分のタスク)を一斉に送信を行い，残りのタスクの通信については待機状態としておく．
- あるタスクのデータを全て受信し終わった場合，グラフを参照して受信したタスクの依存関係を解析して，同期が必要なタスクを抽出する．それらのタスクの送信ホストとなっているホストに対して1byteのデータを送信する．
- 1byteのデータを受け取った場合，同様にグラフを参照してタスクの依存関係を解析する．そのタスクについて同期が必要な全てのホストからの1byteのデータを受け取っていたならば，当該タスクのデータを送信する，
- 転送するタスクについては，データの断片を受信し次第すぐにパイプライン状に転送を行う．

これらの機構が実装されたプログラムの起動は，GXP[1]を用いて行った．GXPとはグリッド用ツールであり，複数ホストにプロセスを一度に立ち上げたりそれらのプロセス間で簡単に通信を行

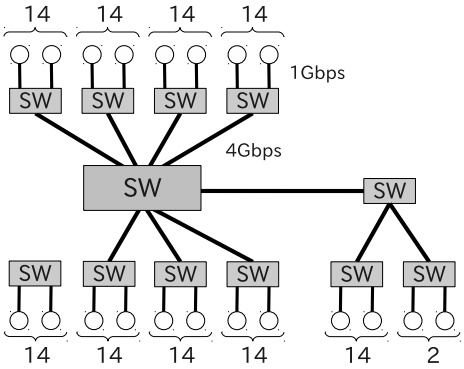

図 6.1: 単一クラスタ (istbs クラスタ) のネットワーク構成

える仕組みを提供するものである．本研究では gxp のコマンドの一つである mw コマンドを用いている．mw コマンドは,

```
$ gxpc mw "program" <args ...>
```

と記述することで，引数 program に指定したプログラムを複数のホスト上で同時に起動させられる．これを利用して，全ホスト上でスケジューラと通信機構部が実装されたプログラムを同時起動することによって集合通信を行わせる．

6.2 スケジューリングに関する性能評価

はじめにスケジューリング部の性能について，調査・検討を行った．実験は図 6.1 のようにホストが 14 台ずつ 1 つのスイッチに接続されており，全体として階層構造を成している研究科が所有するクラスタ (istbs クラスタ) を用いた．

まず，alltoall 操作で，通信に参加するホスト数を変化させたとき，生成されるタスクとグループの数がどのような関係を持つかを示したのが図 6.2 である．ここで，通信ホストは図 6.1 の左上側に描かれているホストから追加していくものとした． 図 6.1 において，生成されるタスクの数はホスト数の 2 乗個である．このとき，生成されるグループの個数はタスクの個数に比例して増加していくが，増加速度は通信が行われるデータの大きさによって異なる．これは通信データ量が少ない場合は，各リンクについて 1 つのタスクごとにグループが 1 つずつ生成されるので，ほぼタスクの個数に比例してグループの個数は増えていくためである．

さらに，ホスト数とスケジューリングの所要時間の関係を表したのが図 6.3 である．特徴としてはスケジューリングに要する時間はほぼタスクの数．すなわちホスト数の 2 乗に比例して増加している．所要時間は例えば 64 ホストで数十ミリ秒程度であるが，これは集合通信に要する時間と比べて

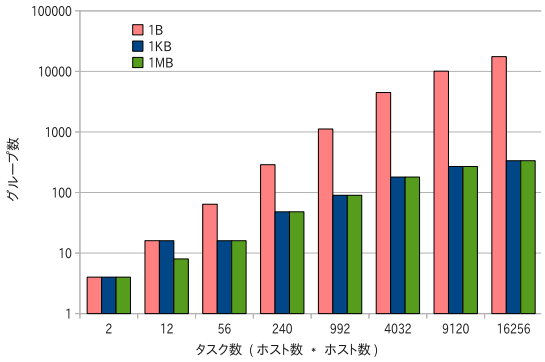

図 6.2: 生成されるグループの個数

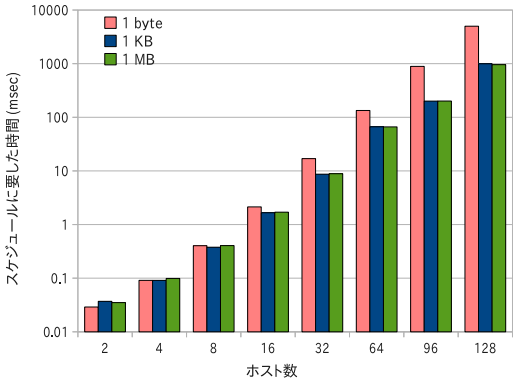

図 6.3: 1 回のスケジューリングあたりの所要時間

小さいので，スケジューリングに要するコストはこの程度ならば問題にならないが，ホスト数が増加していくと特に通信数の多い全体全通信では問題になると考えられる．

対処方法はいくつか考えられる．一般に並列アプリケーションで集合通信の API は同じ操作で同じ大きさのデータを何度も繰り返し通信するように呼び出されるケースが多い．そこでそのような場合には，スケジューリングを行った結果をキャッシュしておくことで，スケジューリングを行う回数を減らすことができると考えられる．その他には図 6.2 での 1KB と 1MB で生成されるタスクの数，グループの数にほとんど差が無いことを考慮し，アプリケーション開始時にあらかじめスケジューリングを行った結果を保持しておき，実行時にその結果を利用することなどが挙げられる．

6.3 単一クラスタ環境における性能評価

提案手法のスケジューリングに基づく集合通信が LAN で接続された単一のクラスタ内でいかほどのスループットを得られるのかについて検証した．この実験では，多対一型の通信からは gather，多対多型通信からは alltoall の集合通信を用いて性能評価を行う．なお，実験は図 6.1 のクラスタで行った．

6.3.1 gather 及び alltoall のスループット

はじめに gather および alltoall のクラスタ内での実行時間の測定を行った．比較対象として，Open-MPI,MPICH および MPICH2 の 3 つの MPI ライブラリを用いた．3 つの MPI ライブラリについては，gather を行う API として MPI_Gather， alltoall には MPI_Alltoall の API が存在しているため，それらの呼び出しにかかった時間を集合通信の実行時間として扱うことにする．

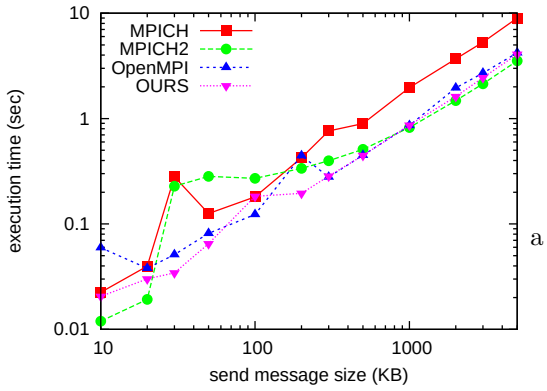

図 6.4: gather の実行時間 (96 ホスト)

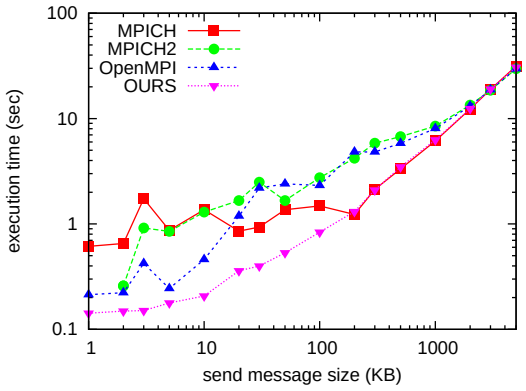

図 6.5: alltoall の実行時間 (64 ホスト)

まず．ホストの数を 96 台とし，さらに 1 ホストが送信するデータのサイズを 10KB から 500KB まで変化させた条件下で gather の実行時間の測定を行った．結果を図 6.4 に示す．横軸は 1 ホストあたりが送信するデータ量であり，縦軸は 1 回あたりの gather の開始時刻から終了時刻までの時間である．

図 6.4 を見ると，総じて送信データ量にほぼ比例して gather の総通信時間が増加しているが，特に MPICH や MPICH2 で数十 KB の領域で総通信時間に大きな変動がある箇所が数点存在する．これが通信のコンテンションによる性能悪化であると考えられる．

同様に alltoall の実行時間を測定した結果を図 6.5 に示す．ただし gather の場合と異なりホストの数を 64 台としている．縦軸が集合通信の実行時間を，横軸は 1 つのホストが別の 1 ホストに対して送信したデータ量を表している．したがって，alltoall 操作によって 1 ホストあたりが自分以外のホストに送信したデータ量の総計は，横軸に記された値の 63 倍である．

図 6.5 では，図の左側 3 分の 2 の送信データ量が 1KB から 200KB 程度においては，本アルゴリズムのスケジューリングによって通信を制御したもの (OURS) は，MPICH や OpenMPI による MPI_Alltoall と比べて実行時間が短くなっており，平均しておよそ 32%程度の時間で alltoall 操作を行うことができている．一方で，1 ホストあたりの送信データ量が数 MB と大きい領域では，3 つの手法で実行時間に大きな差は生じていない．この理由は 2 つ考えられる．1 つ目は通信のコンテンションの影響よりも，図 6.1 のネットワーク中のスイッチとスイッチの間に位置するリンクがデータで飽和してしまってボトルネックとなっており，そのリンクを全部のデータが通過する時間が支配的となっていることである．2 つめは全実行時間が数秒から数十秒程度と RTO の最小値 (200 ミリ秒) に比べて十分長いため，コンテンションによる通信の停止時間による違いがほとんど結果に現れないためであると推測できる．

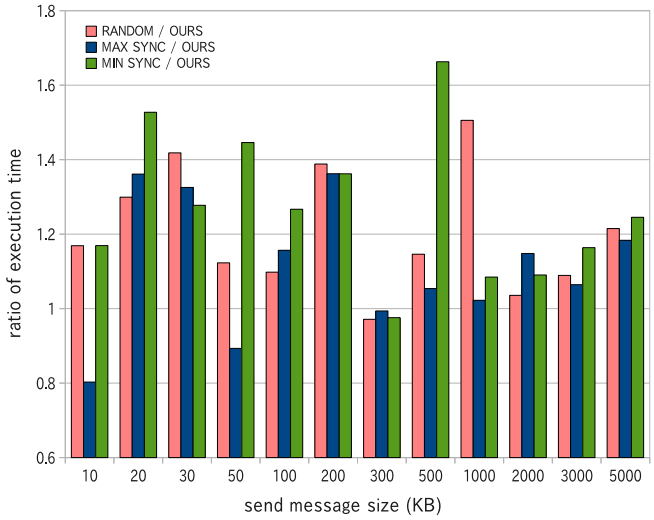

図 6.6: 転送の有無による gather の実行時間の差異

6.3.2 データ転送の有無に起因するスループットの差異

既に述べたように図 5.6 のシミュレーションにおいて，特に gather のような多対一型通信ではデータの転送を行うことが効率化する上で有効であるという結果が得られている．そこで 5.4 で述べた select_task のいくつかの方法を用いたスケジューリングを行って gather を実行した際の性能比較を行った．比較に用いた手法は，まず提案手法によるスケジューリングを行った場合 (OURS) ，転送を全く行わずにランダムにタスクを選ぶ (Random), 最小の同期コストとなるタスクを選ぶ (Min Sync), 最大の同期コストとなるタスクを選ぶ (Max Sync) の方法の 4 つである．その結果を図 6.6 に示す．

図は転送を行った場合の gathern の総実行時間を 1 とし，転送を全く行わない残りの手法の実行時間との比率をグラフにしている．転送を行った場合では総じて転送を全く行わない場合に比べて比較的高いスループットが得られ，平均で約 14% の性能向上となっている．しかし前章でシミュレーションを行った場合と比べてさほど各手法間の結果のばらつきは少ない．これは，クラスタ内のネットワークのバンド幅・遅延がほとんど均一であるため，各ホスト間の同期のコストや通信のコストに差がつかないために，どのようなタスクの選択方法を用いても同じような通信の順序関係がついてしまうからである．結果の変動は通信を行うたびに毎回変動する実行時間の揺らぎ (ホスト間の遅延のゆれや各ホストの処理時間のずれ) の影響が大きい．

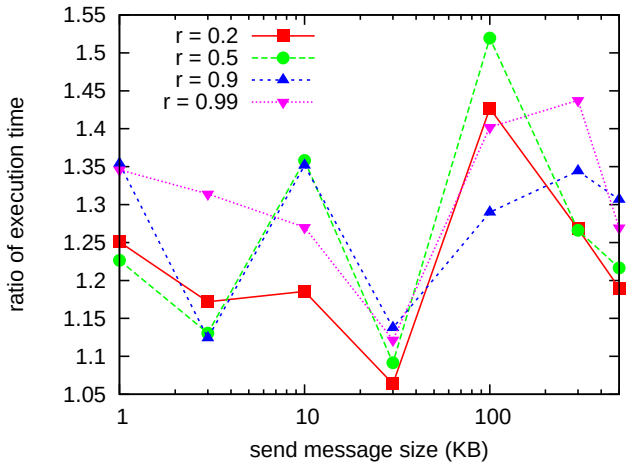

図 6.7: 遅延・バンド幅の変動が通信時間に与える影響

6.3.3 遅延・バンド幅の変動が通信時間へ与える影響

提案手法ではネットワークのバンド幅，遅延のパラメータを用いて通信のスケジューリングをしている．そのため実際のネットワークの遅延やバンド幅の値とスケジューリングに使用した値とが大きく異なっていた場合，通信間の最適な順序関係がつけられずに通信性能が悪化してしまう可能性がある．

そこで，スケジューリングに使用するネットワークの遅延・バンド幅の値を人為的に変動させた場合の総通信時間の変化について検証した．ここでは，使用するネットワーク中の全遅延 L およびバンド幅 B の値に対して，以下の変換を行って得られた値 L'，B' を用いてスケジューリング及び通信を行った．

$$L' = ((2R-1)r+1)L \tag{6.1}$$

$$B' = ((2R-1)r+1)B \tag{6.2}$$

上式の R は 0 以上 1 未満の値を取る乱数である．r は 0 以上の任意の値で変化率に相当し，変換後の値は元の値の $\pm r\%$の範囲に収まることを意味する．この r の値を変化させたとき，64 ホスト間の alltoall の実行時間は図 6.7 のようになった．縦軸の結果の値は $r = 0$ の場合の総実行時間を 1 として，それ以外の場合との実行時間の比率を表している．

この結果から言えることは，スケジューリングに使用する遅延やバンド幅の値と，実際に通信を行うネットワークの遅延やバンド幅の値が大きく異なってしまうと，実際の通信時間もおよそ 5%から 50%程度まで悪化してしまうということである．そのため値の乖離が起きないように実環境で実

際に測定した遅延やバンド幅の値を利用してスケジューリングを行うことが，提案手法によるコンテンションフリーな集合通信を行う上で重要であると結論できる．

6.4 複数クラスタ環境での性能評価

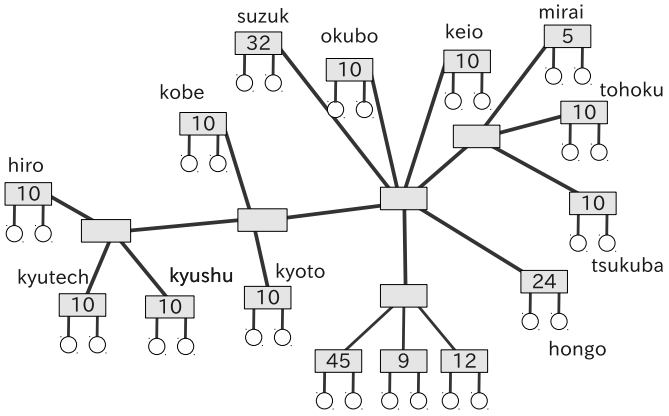

図 6.8: 使用したネットワークの構成

さらに，複数クラスタが存在する環境で性能の検証を行った．実験環境としては InTrigger プラットフォーム [2] を利用した．使用した各クラスタの仕様およびネットワークの構成は表 6.1 の通りである．なお以降の実験にはこの環境の一部もしくは全部を用いることとする．

6.4.1 2 クラスタでの gather および alltoall のスループット

まず suzuk クラスタと chiba クラスタからそれぞれ 32 台ずつ，合計 64 台のホストを用いて単一クラスタの場合と同様 alltoall と gather の 2 つに関して提案手法の評価を行った．比較対象としては 6.3.2 と同様で，Random, Max Sync, Min Sync の 3 つと，それに加えて何のスケジューリングも施さずに一斉に送受信を開始した場合 (Concurrent) の合計 4 つである．なおルートホストは chiba クラスタの 1 台のホストを用いた．

gather を行った場合の実行時間の結果は図 6.9 である．図 6.9 での送信データのサイズが 20KB から 200KB 程度までの領域で Concurrent では実行時間が 200 ミリ秒を超えており，急激に性能が悪化してしまっている．これはコンテンションが原因だと考えられる．また，送信データ長が 300KB 以上と大きい領域では，Concurrent に比べて Max Cost や Min Cost のスループットが低くなっている．これは，クラスタ間の遅延が大きいために通信間で同期を行う時間が無視できないためである．

表 6.1: 各クラスタの仕様

クラスタ名	設置場所	ホスト数	ネットワーク (クラスタ内/クラスタ出口)
chiba	国立情報学研究所	32	1Gbps or 2Gbps / 10Gbps
hongo	東京大学 (本郷)	24	1Gbps or 2Gbps / 10Gbps
kyoto	京都大学	10	1Gbps / 10Gbps
okubo	早稲田大学	12	1Gbps / 10Gbps
suzuk	東京工業大学	32	1Gbps
mirai	はこだて未来大学	5	3Gbps (1Gbps * 3)
keio	慶応大学	10	3Gbps (1Gbps * 3)
hiro	広島大学	10	3Gbps (1Gbps * 3)
kyushu	九州大学	10	3Gbps (1Gbps * 3)
kyutech	九州工業大学	10	2Gbps (1Gbps * 2)
tohoku	東北大学	10	2Gbps (1Gbps * 2)
tsukuba	筑波大学	10	2Gbps (1Gbps * 2)

一方，提案手法は Random や Min Sync と同様に送信データサイズが小さい領域では Concurrent のように性能が急激に悪くなることもなく，また，送信データサイズが大きい領域では転送を行わない Min Sync の手法と比べて実行時間が平均 50%程度に抑えられるまでに高いスループットを得ることができており，クラスタを複数またいだ環境でもデータ転送を行う有効性が示されている．

次に alltoall を行った結果を図 6.10 に示す．alltoall では提案手法と Concurrent, Min Sync との間に差異はほとんどない．一方それに比べて Random や Max Sync ではおよそ 10 倍も通信性能が悪化している．まず提案手法と Min Sync の差異がほとんど無いのは，これまで述べてきたように両者の違いであるデータ転送がほとんど行われないためである．次に Concurrent との比較であるが，gather の場合と異なり，chiba クラスタと suzuk クラスタを接続するリンクには合計で 1 ホストが送信するデータサイズの 1024 倍のデータが流れることになる．そのため，この WAN のリンクがボトルネックとなってしまうことでコンテンションによる性能の悪化が無視できる程度になってしまったためだと考えられる．また，また Random や Max Sync のタスク選択方法だと，遅延が大きい chiba-suzuk 間を同期パケットが通過しなければならないような同期が頻繁に行われ通信が逐次化する．2 つの手法の性能がかなり低くなっているのは，このクラスタ間をまたぐ同期に要している時間が非常に大きいためだと考えられる．

6.4.2 クラスタ数の増加とそれに伴う通信性能の変化

前述の実験と同様の手順で，3 つ以上の複数クラスタを利用した場合の性能について評価を行った．ここでは，通信を行うクラスタの数を 1 つずつ増加させていったときの alltoall の実行時間を測

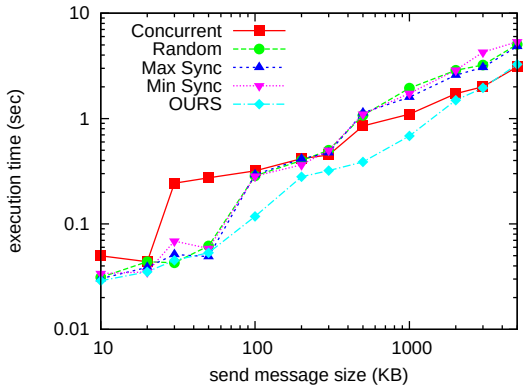

図 6.9: 2 クラスタでの gather の性能

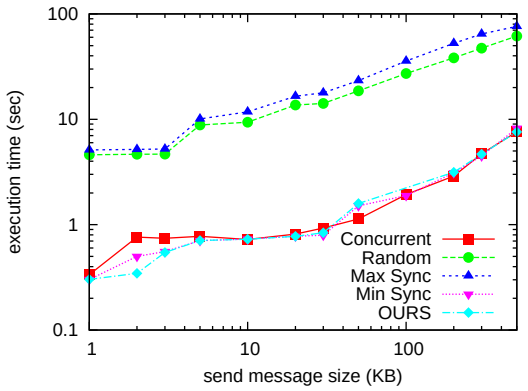

図 6.10: 2 クラスタでの alltoall の性能

定した．

表 6.2: alltoall の通信に参加させたクラスタ

図 6.11 との対応	クラスタ名	ホスト数	通信に参加した全ホスト数
1	chiba	32	32
2	hongo	16	48
3	suzuk	24	72
4	okubo	10	82
5	keio	10	92
6	tsukuba	10	102
7	tohoku	10	112
8	kyoto	10	122
9	hiro	10	132
10	kyushu	10	142
11	kyutech	10	152
12	mirai	5	157

追加していったクラスタは，図 6.8 に載せたものであり，実行時間がなるべく単調に増加することを目的として基本的に chiba クラスタに地理的に近いクラスタから順にクラスタを増やしていった．1 ホストあたりが別の 1 ホストに送信したデータのサイズは一律 100KB として， Concurrent, Max Sync, Random および提案手法の 4 つで性能を比較した．実行結果は図 6.11 であり，横軸がクラスタの数，縦軸が 1 回の alltoall に要した時間である．クラスタ数とクラスタ名および実際のホストの数との対応関係は表 6.2 に列挙してある．

2 クラスタの場合と同様に，提案手法と Concurrent との間に差異はクラスタ数が 4 の時を除いてほとんど見られないものの，比べて Random や Max Sync では 1.2 倍からも 10 倍程度通信の性能が悪化している．結果，同じコンテンションフリーな通信でも同期の回数や時間を削減するような通

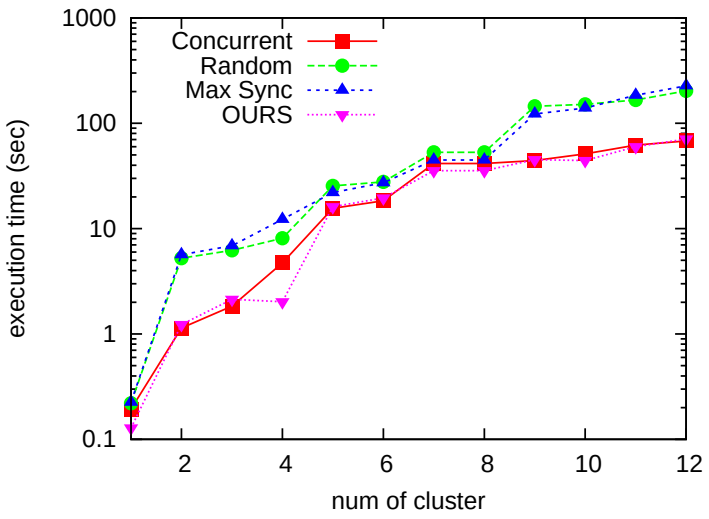

図 6.11: クラスタ数の増加と通信性能

信の順序づけを行うことが広域環境では非常に重要であることが確認できた.

第7章　おわりに

7.1　まとめ

本論文ではネットワーク上でのコンテンションの発生を防止することで，高性能な集合通信を実現するアルゴリズムを提案し，実装を行った．コンテンションはリンクや受信機器のバッファの容量を超えるデータが同時にネットワークの一部のリンクに集中することが原因であるから，これを発生させないためには個々のリンクで通信を排他的に行う必要がある．そこでまずは個々の通信を一種のタスクと見なした上で，各々のタスクをいくつかにグループ分けして相互に依存関係をつけて通信させるコンテンションフリーな通信を行うためのフレームワークを提案した．

さらにそのフレームワークを利用して，実際に集合通信のスケジューリングを行うアルゴリズムを提案した．個々のリンクで厳密に通信を排他的に行うようにするためにはホスト間で同期を取る必要があるが，遅延が大きいネットワークや通信データ量が小さいときには同期のコストが相対的に大きくなり通信効率が悪くなるため，本手法ではネットワークや遅延のパラメータを利用することで，ホスト間で同期を行うか，あるいはデータを別のホストに転送させるかを切り替えることで，コンテンションを起こさない条件を満たしつつも高い効率が得られるようになっている．そのため，ネットワークがどのような構造をしていようとも，提案手法のアルゴリズムだけを適用して通信をスケジューリングすることで，コンテンションを発生させない集合通信を行うことを可能にした．

そして，提案手法を実際に実装し，多対多型通信の代表である alltoall，および多対一型通信の gather に関して本アルゴリズムによるスケジューリングを行った結果を利用した集合通信と，他手法との性能の差異を検証した．その結果単一クラスタのみを用いたものでは既存の OpenMPI や MPICH,MPICH2 のライブラリと比べて，最大でおよそ 32%程度にまで実行時間を短縮させることができた．さらに複数のクラスタを利用した場合，単一クラスタと同様のアルゴリズムを適用しても，何もスケジューリングを行わない場合と比較しておよそ 30%程度にまで通信時間を削減することができた．また，タスクの選択方法によって同期に要する時間が異なるため，広域環境においてはスループットが大きく変動することになるが，データを転送を行うことによって同期回数を削減することで通信性能を向上させることの有効性も示した．

7.2 今後の課題

今後の課題としては次のような点が挙げられる．

- broadcast や reduce 操作などへの対応.
  今回提案したアルゴリズムでは，各ホストがそれぞれユニークなデータを保持しており，送信元や送信先のホストごとに異なるデータを送る (もしくは受け取る) 場合を前提としていた．broadcast や reduce のように，送信元や送信先が異なっても同じデータを送る場合でも本手法は適用可能であるが，同一のデータを繰り返し送信 (もしくは) 受信することになってしまうために効率が悪くなってしまう．そこでこのような集合通信でも高い性能が得られるように，アルゴリズムを改良する必要がある．

- スケジューラの高速化.
  実験結果にもあるように，ホスト数が多くなってくるとスケジューリングの計算に要する時間が無視できなくなってくる．そこで，スケジューラ自体の内部構造を変えるなどして，より少ない計算時間でスケジューリングが行えるようにする．

- 接続性が限られているような環境での実行.
  本論文では全対全で接続され，ホスト間では直接通信が行えるような環境を前提としてアルゴリズムを提案した．ところが，デッドロックフリールーティング [16] のように，広域な環境で通信を行うためにはコネクションを張る数を制限するなどの工夫が必要となってくることがある．このように接続性が限定されている条件下でも同様のアルゴリズムが適用可能となるように，手法を見直していく必要がある．

参考文献

[1] Gxp. GXP : Grid and Cluster Shell. http://www.logos.ic.i.u-tokyo.ac.jp/gxp/.

[2] Intrigger platform. http://www.intrigger.jp/.

[3] iperf. http://sourceforge.net/projects/iperf/.

[4] Mpich. http://www.mcs.anl.gov/mpi/mpich.

[5] Openmpi. http://www.open-mpi.org/.

[6] Xin Yuan Ahmad Faraj and D Lowenthal. Star-mpi: Self tuned adaptive routines for mpi collective operations, 2006.

[7] Mike Barnett, Satya Gupta, David G. Payne, Lance Shuler, Robert Geijn, and Jerrell Watts. Building a high-performance collective communication library. In *In Supercomputing'94, Washington D. C*, pages 107–116. IEEE Computer Society Press, 1994.

[8] Prashanth B. Bhat, C. S. Raghavendra, and Viktor K. Prasanna. Efficient collective communication in distributed heterogeneous systems. *JPDC*, pages 15–24, 1999.

[9] Ahmad Faraj and Xin Yuan. An empirical approach for efficient all-to-all personalized communication on ethernet switched clusters. In *In ICPP, 2005*, pages 321–328.

[10] Ahmad Faraj and Xin Yuan. Automatic generation and tuning of mpi collective communication routines. *ACM International Conference on Supercomputing (ICS2005)*, pages 393–402, 2005.

[11] Ahmad Faraj, Xin Yuan, and Pitch Patarasuk. A message scheduling scheme for all-to-all personalized communication on ethernet switched clusters. *IEEE Transactions on Parallel and Distributed Systems*, 18:2007, 2007.

[12] Junichi Hatta and Susumu Shibusawa. Scheduling algorithms for gather operations in distributed heterogeneous systems. *Technical report of IEICE*, pages 33–40, April 2000.

[13] Van Jacobson and Michael J. Karels. Congestion avoidance and control, 1988.

[14] Yuya Jinno, Masashi Ito, and Akihiro Fujiwara. Efficient scheduling algorithms for total exchange on grid environment. *In proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications, Vol.1*, pages 81–87, 2003.

[15] Nicholas T. Karohis, Brian Toonen, and Ian Foster. Mpich-g2: A grid-enabled implementation of the message passing interface.

[16] Hideo Saito Ken Hironaka and Kenjiro Taura. High performance wide-area overlay using deadlock-free routing. *International ACM Symposium on High Performance Distributed Computing (HPDC2009)*, pages 81–90, 2009.

[17] Thilo Kielmann, Rutger F.H.Hofman, Henri E.Bal, Aske Plaat, and Raoul A.F.Bhoedjang. Magpie: Mpi's collective communication operations for clusterd wide area systems. *PPoPP'99*, page ., May 1999.

[18] Alexey Lastovetsky and Is haka Imkwawa. Modeling performance of many-to-one collective communication operations in heterogeneous clusters, 2006.

[19] Alexey Lastovetsky and Maureen O'Flynn. A performance model of many-to-one collective communications for parallel computing. *Proc. of IPDPS*, page ., 2007.

[20] Alexey Lastovetskym, Maureen O'Flynn, and Vladimir Rychkov. Optimization of collevtive communications in heterompi. *LNCS 4757*, pages 135–143, 2007.

[21] Glenn R. Luecke, Bruno Raffin, and James J. Coyle. Comparing the Communication Performance and Scalability of a Linux and an NT Cluster of PCs, a SGI Origin 2000, an IBM SP and a Cray T3E-600. *The Journal of Performance Evaluation and Modelling for Computer Systems*, March 2000. http://hpc-journals.ecs.soton.ac.uk/PEMCS/.

[22] Hiroya Matsuba and Yutaka Ishikawa. Aggregate router: An efficient inter-cluster mpi communication facility. *IPSJ Online Transactions*, 2:215–224, 2009.

[23] Motohiko Matsuda, Yutaka Ishikawa, Tomohiro Kudoh, Yuetsu Kodama, and Ryousei Takano. Efficient collective algorithms for grid environment. *IPSJ SIG Notes Vol.2006*, pages 257–262, Jul 2006.

[24] Sho Naganuma, Kei Takahashi, Hideo Saito, Takeshi Shibata, Kenjiro Taura, and Takashi Chikayama. Improving efficiency of network bandwidth estimation using topology information. *SACCIS 2008*, pages 359–366, Jun 2008.

[25] M. Allman V. Paxson and W. Stevens. Tcp congestion control. RFC 2581, 1999.

[26] Tatsuya Shirai, Hideo Saito, and Kenjiro Taura. A fast topology inference-a building block for network-aware parallel computing. *In Proceedings of the 16th IEEE International Symposium on HPDC*, pages 11–21, Jun 2007.

[27] Luiz Angelo Steffenel, Maxime Martinasso, and Denis Trystram. Assessing contention effects of all-to-all communications on clusters and grids. *International Journal of Pervasive Computing and Communications*, pages 440–459, 2008.

[28] Anthony T. C. Tam and Cho-Li Wang. Contention-free complete exchange algorithm on clusters. In *In Proceedings of the IEEE International Conference on Clusters*, pages 57–64, 2000.

[29] Rajeev Thakur and Rolf Rabenseifner. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19:49–66, 2005.

[30] Vinod Tipparaju and Jarek Nieplocha. Optimizing all-to-all collective communication by exploiting concurrency in modern networks. *Proceedings of the 2005 ACM/IEEE conference on Supercomputing table of contents*, 19:46–50, 2005.

[31] Sathish S. Vadhiyar, Graham E. Fagg, and Jack J. Dongarra. Performance modeling for self adapting collective communications for mpi. In *In LACSI Symphosium*, pages 15–18. Springer, 2001.

[32] Sathish S. Vadhiyar, Graham E. Fagg, and Jack J. Dongarra. Towards an accurate model for collective communications. *In Proceedings of the International Conference on Computational Science, volume 2073 of LNCS*, 18:41–50, 2001.

発表文献

1. 吉富翔太, 田浦健次朗. メッセージ衝突を防止する適応的な集合通信. 情報処理学会全国大会 2010. 東京, March 2010.
2. 吉富翔太, 田浦健次朗. メッセージ衝突を防止する適応的な集合通信. 並列／分散／協調処理に関するサマーワークショップ（SWoPP2009）, 仙台，August 2009.
3. 吉富翔太, 弘中健, 田浦健次朗. メッセージ衝突を防止する適応的な集合通信アルゴリズム. 先進的計算基盤システムシンポジウム (SACSIS2009), pp.71-78, 広島，May 2009.
4. 吉富翔太, 斎藤秀雄, 田浦健次朗, 近山隆. 自動取得したネットワーク構成情報に基づく MPI 集合通信アルゴリズム. 並列／分散／協調処理に関するサマーワークショップ（SWoPP2008）, pp.7-12, 佐賀，August 2008.
5. 吉富翔太, 斎藤秀雄, 田浦健次朗, 近山隆. 自動取得したネットワーク構成情報に基づく MPI 集合通信アルゴリズムの改良. 情報処理学会全国大会 2008. つくば, March 2008.

掲載予定

1. 吉富翔太, 田浦健次朗. メッセージ衝突を防止する適応的な収集操作アルゴリズム．電子情報通信学会 情報・システムソサイエティ論文誌 「情報爆発」特集号 (2010 年 6 月号).

謝辞

本研究を進めるに当たって，近山隆教授，ならびに田浦健次朗准教授には大変貴重なご指摘，ご助言をいただきました．近山隆教授には特に発表練習のときにプレゼンテーションの要点や研究の表現方法などに関して的確なアドバイスをいただきました．田浦健次朗准教授には研究の方針から論文の論理構成や発表方法など，研究者として必要な技術についても数多くのご指導をいただくとともに，研究に行き詰まって突然相談に伺ったときでも，常に親身になって的確なご助言のもと研究の道筋を作っていただきました．卒論生として研究室に配属されてからの 3 年間で数え切れないほどのご指導を賜りました．心より感謝申し上げます．

助教の横山大作さんをはじめ，鴨志田良和さん，柴田剛志さんにはミーティングなどを通して様々な角度からアドバイスをいただき，常に新たな視点で研究を見直し，そして掘り下げていくことができました．研究室の先輩である斎藤秀雄さんや弘中健さんには，日頃から研究方針からどんな些細な質問にいたるまで何でも気軽に相談にのっていただき，今思えば本当に数多くのことを教えていただきました．また後輩の原健太朗君や中島潤君，加辺友也君には日々の研究の議論だけでなく，サーバ管理などを通して研究環境を維持して頂けたことで安心して数多くの実験を行うことができました．そして，研究に行き詰ったときには同期の長沼翔君，佐伯勇樹君をはじめ研究室の多くの方に話を聞いていただき，また励ましていただきました．ここまで研究を続けられたのも，ひとえに皆様のお陰です．本当にありがとうございました．

平成 22 年　2 月　9 日