筑波大学大学院博士課程

工学研究科修士論文

プロセッサ間通信記述用のビジュアルプログラミングシステムの提案

電子情報工学専攻

著者氏名　酒寄 保隆

指導教官　田中 二郎

平成 11 年 2 月

要旨

数値計算などの分野で必要とされる高速な計算を可能とするコンピュータとして, 複数のプロセッサを用いて計算を行なう並列計算機の研究が, 近年盛んに行われている. そのような並列計算機用のプログラム言語として C^* やその拡張である NCX などがある. これらの SPMD 型の並列言語処理系では, パフォーマンス向上のためのプロセッサ間通信のコンフリクトの待避を行う処理が行われている.

我々はこの処理の動作を図解的に考えるところから出発した. このことにより, 複雑になりがちなテキストでの処理ではなくグラフ上での最適化処理を構築した. 同時にこの処理のための入出力機構と, 最適化結果を実行可能なコードに変換する自動コード生成機構を設計した. そこで本論文では, そのようにして構築された並列計算機特有のプロセッサ間通信に特化したビジュアルプログラミングシステムを提案する.

このシステムでは, SPMD 型のプログラミングの際に現れるプロセッサ間通信の記述を, グラフィカルに行うことを可能とする. その入力によって得られた送受信関係等の情報は, 通信コンフリクトの待避などを行う最適化処理系を通し, 各種並列実行環境に沿ったコードに自動変換される. また, 最適化の結果をグラフィカルに出力することで, 効率化とともにユーザに対するデバッグなどのサポートも可能とする.

目次

図一覧

第 1 章

序論

近年の数値計算などに対する高性能計算を実現する手段として, 並列計算は特に主流なものとなっている. それを実現する環境は, 個々のワークステーションをバス結合した構成や, 単体の計算機の中に複数のプロセッサを内蔵させたもの, 更には数百から数千のプロセッサで構成された超並列計算機など, 様々なものが存在する. 超並列計算機に至っては, そのピーク性能は数百 Gflops や数 Tflops に至るものまで存在する. しかし, ピーク性能とは各々のプロセッサが何のロスも起こさず計算をし, そのうえプロセッサ間には一切の通信も行わない場合の計算機全体の性能であり, 実際のアプリケーションではプロセッサ間通信やキャッシュミスなどにより, 性能の低下は免れない. プログラムの構成によっては, その性能の低下が著しく現れることも考えられる. この現象は, 並列計算を行うアーキテクチャの構成に関わらず共通して現れてくる. 本研究では, その並列プログラミングの際に必要なプロセッサ間通信について注目した.

1.1 研究の背景

まず今回の研究の対象である, 並列計算機のアーキテクチャと, そのプログラミングについて述べていく.

1.1.1 並列計算機

現在多くの並列計算機は, 多少のローカルメモリを持つプロセッサどうしが互いに通信するという, メッセージパッシングというパラダイムの基で設計されている. 本研究での対象となる並列計算機の構成は, 分散メモリ型と呼ばれる並列計算機の中で

も, 特に MIMD と呼ばれるものを対象としている. MIMD とは **M**ultiple **I**nstruction **M**ultiple **D**ata の略であり, そのアーキテクチャは図 1.1に示した構成を持つ. この構成では, 各プロセッサに流される命令とデータは複数存在することになる. よって各々のプロセッサは, 各々異なる命令を任意のデータに対して実行することができる. また, この図に示される CPU とメモリの組を, 1 つの独立したコンピュータとしてみることができ, そのように見るとこれは分散環境と呼ばれるものに酷似している.

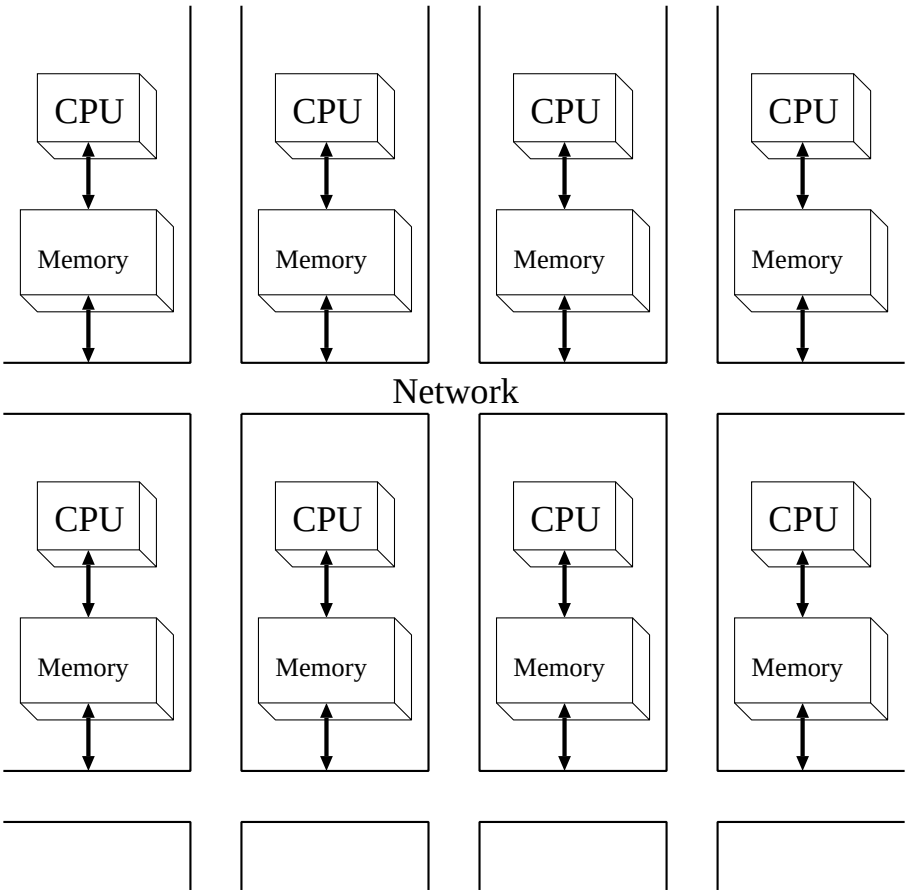

図 1.1: 分散メモリ型 MIMD 並列計算機

これに対し, SIMD 型と呼ばれるアーキテクチャも存在する. SIMD とは **S**ingle **I**nstruction **M**ultiple **D**ata の略で, 図 1.2に示すアーキテクチャで示される. SIMD 並列計算機は 1 つのコントロールユニット (Front End) が, 一様に用意された複数の

プロセッサ (Processing Elememt) に, 共通の命令を各々のデータに対し実行させる. つまり, SIMD では命令の流れは 1 つ, データの流れは複数となる.

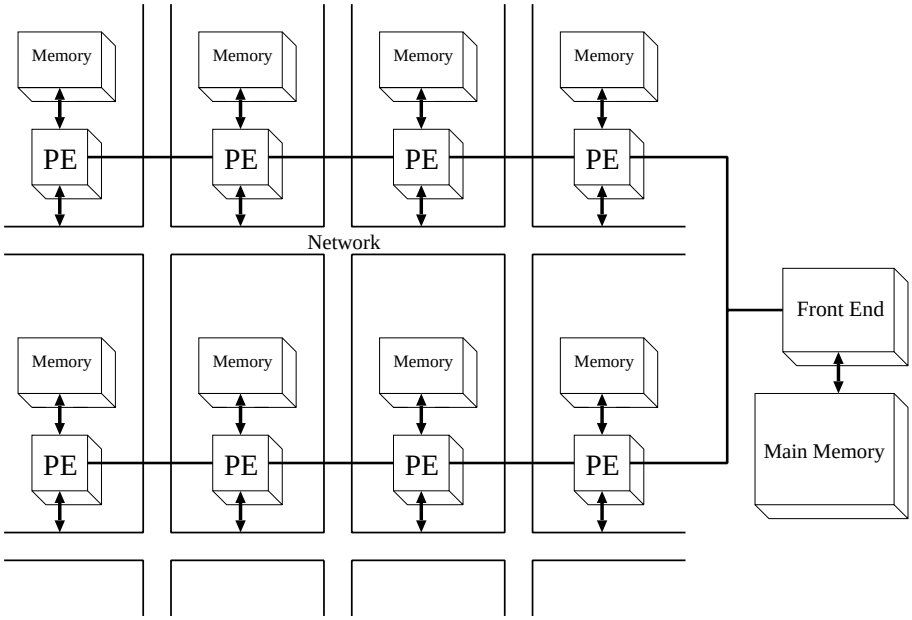

図 1.2: 分散メモリ型 SIMD 並列計算機

SIMD 並列計算機は, コントロールユニット以外のプロセッサにはレジスタと計算機能以外を持つ必要がない. よってその構成は MIMD に比べ簡単になり, 多数プロセッサの構成にしやすい. しかし, 同時に実行する命令は 1 つであるため, 条件文などの実行には大きなロスが引き起こる. よって各プロセッサの性能を十分に引き出すことが難しく, 一般的にその構成は, MIMD 並列計算機よりも低速 · 安価なプロセッサを多数用いるものが多い.

プログラミングの際の違いでみると, SIMD のコードはフロントエンドが実行するスカラーコードと, バックエンドが並列に実行するパラレルコードに分けられる. SIMD のメモリは PE 間で直接参照できないため, メッセージのやり取りによって参照される. これに対し MIMD は, 全てのプロセッサがコントロールユニットとしての機能を持っていると言える. よって MIMD 型でも SIMD 型の様なコーディングも可能であるが, 各々のプロセッサは独立にメモリを持っているため, SIMD のスカラーコード

を各々のプロセッサにコピーして, 各々シーケンシャルに実行することができる. この様なプログラム形態は SPMD(Single Program Multiple Data) と呼ばれる.

その他, 分散メモリ型に対して共有メモリ型と呼ばれる並列計算機のアーキテクチャも存在する. 共有メモリ型では, 複数のプロセッサが共通の 1 つのメモリにアクセスするため, プロセッサ間のデータのやりとりは, メインメモリを介して行われる. そのため分散メモリ型の並列計算機に比べて, 共有メモリ型の並列計算機ではそのプログラミングは容易なものとなる. しかし主メモリの帯域幅の限界などのから, そのプロセッサ台数は数台から数十台規模が限界となる. 近年では分散共有メモリ型と呼ばれるアーキテクチャも登場してきたが, やはり同様の問題から超並列と呼ばれるような構成は不可能である.

この様な特徴から, 現在の並列 · 超並列と呼ばれるような高性能を求めるコンピュータの構成は, その多くが分散メモリ型 MIMD 並列計算機のアーキテクチャを持っている.

1.1.2 並列プログラミング

ではその分散メモリ MIMD 型並列計算機における, 一般的なプログラミング形態はどのようなものがあるか. それについて以下に述べていく. 一般的に MIMD 型並列計算機においては, そのプログラミングの形式は以下の 2 つが多く用いられる.

1.SPMD(Single Program Multiple Data) 形式

2.Master-Slave 形式

SPMD 形式では, 全てのプロセッサが同じプログラムを実行する. 各プロセッサは同じプログラムのプロセスが生成され, 各々のプロセスは独立に実行が可能である. 各プロセッサ間でデータの依存関係が発生する場合には, プロセッサ間通信によってメッセージをやり取りすることにより, その処理が実行される. また, 各プロセッサで各々違った処理を行いたい場合には, 各プロセッサに割り与えられた ID(プロセッサ番号) を引数とした条件文を用いることにより実行ができる.
Master-Slave 形式では, プログラムは 1 つの Master プログラムと, 1 つ以上の Slave プログラムから構成される. プログラムを実行すると, 1 つのプロセッサ上で Master プログラムのプロセスが生成される. Master は必要に応じて並列実行する Slave プ

ログラムのプロセスを生成し, それを各プロセッサ上で実行する.
Master-Slave 形式では, より柔軟なプログラミングとプログラミングのモデル化が可能であるが, 実際のプログラミングは SPMD 形式に比べて複雑になる.

1.1.3 プロセッサ間通信

プログラマがプロセッサ間通信の記述をする際の手段は, どのようなものがあるか. 最も高性能な通信速度を実現する方法は, 環境に用意されたネイティブな関数を用いることである. 例えば, 筑波大学計算物理学研究センターで稼働中の超並列計算機 CP-PACS[1, 2] では, 遠隔メモリへの直接書き込みを行う, RDMA(**R**emote **D**irect **M**emory **A**ccess) 通信を実現する通信関数 [3, 4] が用意されている. この様な関数を用いると, 1 回当たりののデータ転送に際しては最も優れたパフォーマンスを示す. しかし, 多機種間でのプログラムのポータビリティは失われ, かつ通信の最適化はユーザ自身が行う必要がある. ユーザに十分な知識があれば, 通信最適化もプログラムに特化したものが可能となるが, そうでない場合にはパフォーマンスの悪化も考えられる. しかも記述の際には, 複雑な仕様を理解・習得しなければならない.

この問題を幾分か解決したものに, PVM[5] や MPI[6] などといった, 仕様が共通化された並列実行環境を用いる方法がある. これらの環境を用いることにより, 仕様が共通化されているため, ある程度のプログラムのポータビリティの確保が可能となる. また, ある 1 つのプロセッサが全プロセッサに対してデータを送信する Broadcast(図 1.3左) や任意の複数のプロセッサに対してデータを送信する Scatter(図 1.3中央), 任意の複数のプロセッサからデータを受信する Gather(図 1.3右) 等の通信関数も用意されているが, 最も基本となるプロセッサ間の単一のセンド・レシーブを明示的に記述できるため, ある程度のプログラムに特化した最適化も可能である. しかし, ネイティブな関数に比べパフォーマンスの低下はやむをえず, またこれらの環境は現在様々なものが多数存在しており [7], これらを理解するのはやはりユーザにとって大きな負荷となりうる.

以上に挙げたものは, ユーザが実プロセッサ間のデータの送受信を意識してプログラミングする場合の手段である. これらは既存のプログラミング言語にプロセッサ間通信環境を用意した形式を取っているが, これに対し, 言語処理系によって実プロセッサ間通信を意識しないプログラミング手法も存在する. 並列計算専用に用意された言語処理系は, 基本的に以下の 2 つに大別される.

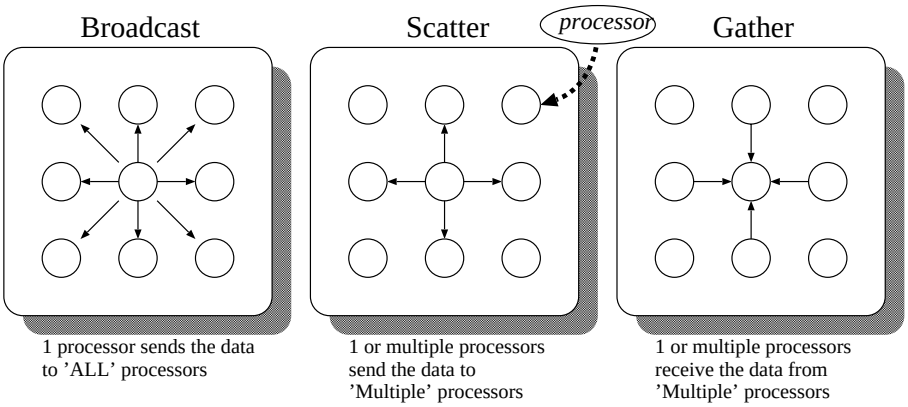

図 1.3: Broadcast, Scatter, Gather の動作イメージ

1. 並列コンパイラ
   C*[8, 9] や NCX[10] などのデータ並列型言語では, 仮想プロセッサという概念を導入し, 並列性を明示的に記述するが, 実プロセッサを意識しないでプログラミングが可能.

2. 並列化コンパイラ
   既存の逐次に書かれたプログラムを自動, もしくは半自動で変換し, 並列実行可能なプログラムにする. プログラムのポータビリティは逐次型 · 並列型計算機間にも確保が可能. また, ソフトウェア資産の流用も可能で, ユーザに対し並列性を意識させる必要もない.

これらのコンパイラを用いることにより, ユーザにとってプログラミングがより容易になるとはいえるが, 最適化の多くをシステムに頼ることになるため, プログラムに特化した最適化はより難しくなる. またこれら言語処理系は, 一旦それらで書かれたプログラムを変換し, PVM や MPI を用いたプログラムを生成 · コンパイルする手法であるため (図 1.4に例として NCX コンパイラの構成を示す), 2 重にシステムに頼る形になってしまう.

よってユーザにとってより容易なプログラミングを実現できる環境を用いるほど, そのパフォーマンスは悪くなるといえる.

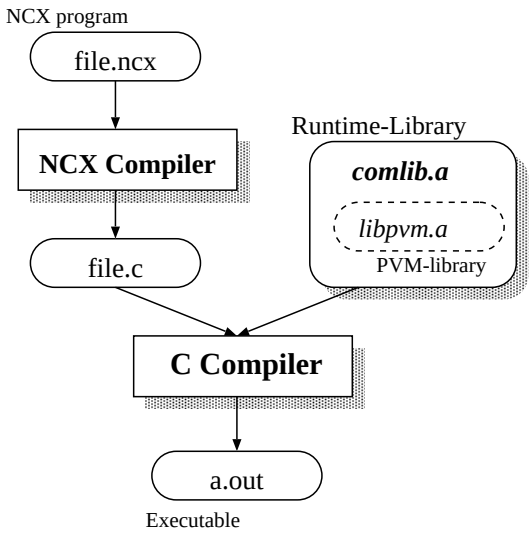

図 1.4: NCX コンパイラ

1.2 研究の目的

並列プログラムと逐次プログラムとの違いで, まず挙げられるのがプロセッサ間通信の存在である. プロセッサ間通信をより効率的に行うことが, 計算の実行を, よりその環境のピーク性能に近づけることにつながる. また, 逐次実行のプログラミング言語と大きく仕様が異なってくる部分も, プロセッサ間通信そのものか, それによって派生した部分である. つまりプロセッサ間通信実現の容易化が, 並列プログラミングの容易化と密接に関係してくると言える. そこで本研究では, そのプロセッサ間通信に着目した. そして本研究の命題は, ユーザフレンドリで, かつ高速なプロセッサ間通信を実現するためにはどのような方法があるか, この追求である.

まず本研究では, プロセッサ間通信実行時に現れる衝突 (コンフリクト) を回避するための, スケジューリングの動作を図解的に構築していくことから出発した. 過 敏意によって提唱されたスケジューリング技法 [11] では, コンフリクト待避のためのスケジューリングを施すことにより, 数十 % の実行時間の短縮化が可能であることが CP-PACS 上での実験により示されている. 本研究では通信実行のスピードアップのために, その過の研究や Kaushik らの研究 [12] で用いられているスケジューリング

アルゴリズムの中から, SPMD プログラムの中の通信実行に対し適用できるものを選択・改良した. そしてその構築の際, そのアルゴリズムを複雑になりがちなテキストではなく, より直観的な図で示した.

また, 通信関数の仕様の多様化・複雑化という問題からユーザを解放するために, プロセッサ間通信の記述に関してビジュアルプログラミング手法を採用することにした. このビジュアルプログラミングとは, 従来の文字列や記号を中心としたインターフェイスではなく, アイコン, 図形, アニメーションという人間がより理解しやすい, ビジュアルな表現を用いたプログラミング形態のことを指す. 新たに PVM や MPI に変わるようなテキスト入力の仕様を提唱するのでは, 多様化を自ら行うことになる. その点ビジュアルプログラミングを採用すると, ユーザは基本的に「どこから」, 「どこへ」, 「何を」送るかという直観的な情報のみを気にすればよい. 実行可能な PVM や MPI のコードへの変換をシステムが行うようにすると, ユーザはその仕様を理解しなくてもコーディングか可能となる. さらに上で述べたスケジューリングアルゴリズムをシステムに組み込むことにより, ユーザに自由な入力を許すことにした.

本研究はこの様な経緯から, 並列計算特有のプロセッサ間通信に特化したビジュアルプログラミングシステムをこの命題の一つの解として考案し, その設計・構築を目的として行った.

第 2 章

ビジュアルプログラミングシステム *GRIX*

本研究では, 並列計算特有のプロセッサ間通信に特化したビジュアルプログラミングシステム **GRIX** (GRICCS : **G**raphical **I**nterprocessor **C**ommunication **S**upport **S**ystem)[13, 14] を提案する.

GRIX が使われるのは, SPMD 型のプログラムを作成している段階で, プロセッサ間通信のコーディングが必要とされる場面である. その際にこのシステムを立ち上げ, 主にグラフィカルな入力でターゲットコードを生成させる. GRIX システムは以下に示すようなコンセプトで, その処理系の構築を行った.

- GUI(Graphical User Interface) に基づいた入力
- ポータビリティの高い最適化処理
- 最適化結果の GUI 出力
- 任意の実行環境に対するコード生成

GUI に基づいた入力機構を用意することにより, ユーザは直観的な入力が可能になる. その入力情報を, あらゆる環境に対応できるポータビリティの高い最適化を施すことにより, 任意の実行環境に対応する自動コード生成が可能となる. これは将来的にどんな環境が現れても, それ用の処理系を従来のものに加えるだけで, 対応が可能になる. そして最適化結果も GUI 出力することにより, ユーザに対する最適化結果と生成されるコードの直観的な理解, 及びデバッグの際のサポートも可能になる.

2.1 処理系の構成

GRIX システムの処理系の構成を, 図 2.1に示す. GRIX はその処理の特徴から, その内部処理系を 3 つに分けることができる.

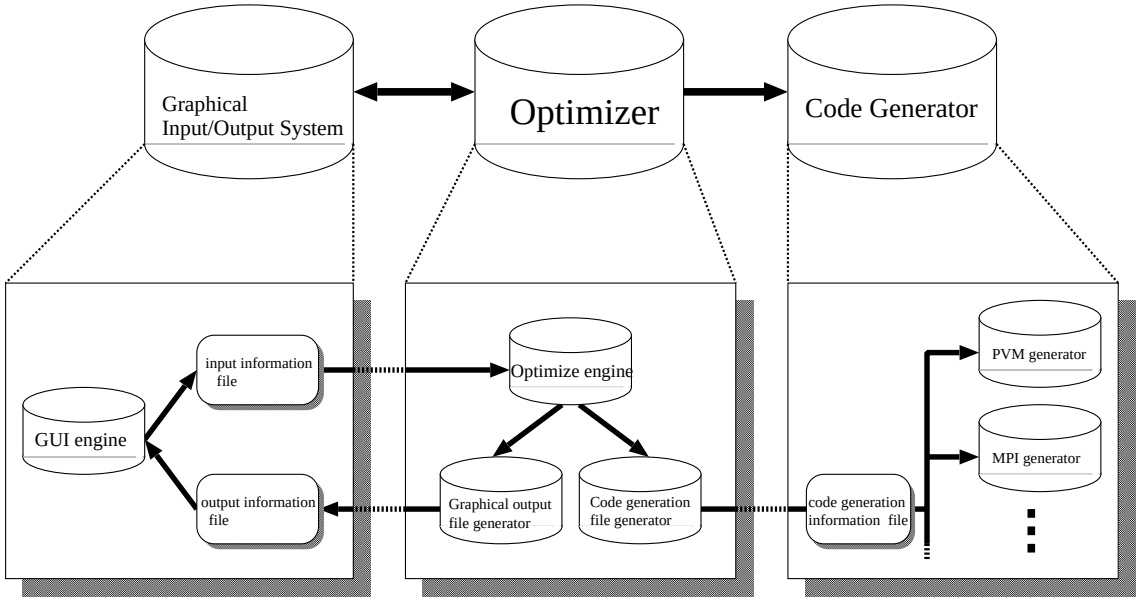

図 2.1: GRIX システムの処理系の構成

Graphical Input/Output System は, GUI による入出力を処理する. このサブシステムに含まれる GUI エンジンが, ユーザの入力によって得られた情報をファイル化し, 最適化を行う処理系である Optimizer に渡す. Optimizer はそのファイルから得られた情報を基に, 通信時間の短縮と自動コード生成のための最適化を施す. そして最適化によって変更された情報を示す, GUI 出力用とコード生成用のファイルを生成する. Graphical Input/Output System は Optimizer からファイルを受取り, 最適化結果を GUI 表示する. この最適化結果の GUI 表示により, ユーザに対する直観的な理解を助け, 生成されるコードの実行状況の理解やデバッグなどの補助をする. またコード生成用のファイルは, Code Generator 内部に複数の環境用のコード生成エンジンを用意することにより, 各々の環境に対応したコードを自動生成する. このコード生成用のファイルを保存しておくことにより, 将来新しい環境用のコード生成エン

ジンを加えた際に, 従来の環境用のコードを速やかに新しい環境用に変換することができる.

2.2 処理系の実装

処理系の実装は, Graphical Input/Output System 内部の GUI エンジンを *TCL/TK* で, システムの内部処理を行う Optimizer, Code Generator に関しては *C*, *C++* で実装を行った.

第 3 章

関連研究

今回の研究に関連する他の研究を, プロセッサ間通信の最適化からの観点と, 並列計算用の GUI ツールからの観点で挙げていく.

3.1 プロセッサ間通信の最適化

本研究での, プロセッサ間通信の最適化の技術は, 多くを並列コンパイラや並列化コンパイラに採用する技術を基に設計している.

NCX コンパイラにおける, プロセッサ間通信の実行時処理を行うライブラリに関する研究 [15, 16, 17] では, 頻繁なメモリコピーを防止した通信効率を向上や, CP-PACS のネイティブな関数である RDMA 通信を用いた場合の各種通信パターンの実装を行っている. また, 並列化コンパイラに関する過 敏意の研究 [11] や Kaushik らの研究 [12] では, プロセッサ間通信実行時に発生するコンフリクト待避の処理とその効果について記されている. 本研究では, それらに用いられているプロセッサ間通信の実行 · スケジューリングアルゴリズムの中から, GRIX システムに組み込めるものを採用 · 改良した. また, GRIX システムに採用されているスケジューリングアルゴリズムは, 同期通信を前提としている. 今後, 非同期通信の適用をしていくにあたり, *PRAM*[18],*LogP*[19, 20] などの並列計算モデルの採用 · 応用が考えられる.

3.2 並列計算用の GUI ツール

GRIX は, プロセッサ間通信記述用に特化したビジュアルプログラミングシステムである. プロセッサ間通信記述に特化したことにより, GRIX は従来にない並列計算用の GUI ツールである.

では並列計算用の GUI ツールという枠組みでは, どのようなものがあるか. P. Newton, J. C. Browne らは, *CODE*[21] というビジュアル並列プログラミング言語を提唱している. また文献 [22] の中で, Newton は同様なビジュアル並列プログラミング言語である *HeNCE*[23] と CODE との比較を行っている. CODE, HeNCE とも, 並列プログラムを記述するという特徴では, GRIX と同様なビジュアルプログラミングシステムと呼べるが, 我々はプログラム全体ではなく, 並列計算固有のプロセッサ間通信にビジュアルプログラミングを適用することが効果的と考える. しかし CODE, HeNCE は, ビジュアルプログラミングにアイコンを用いたより直観的な手法を用いている. この点は今後, システムの拡張の参考になりうる.

また PVM プログラムを制御 · 可視化する, *XPVM*[24] という PVM 用のツールがある. XPVM はプロセッサ間通信をコントロールする点で, GRIX と共通の特徴を持つが, 我々は特定の環境に特化したシステムではなく, あらゆる環境に柔軟に対応できるシステムを目指している.

これ以外にも, 並列プログラム実行時の各プロセッサの負荷を図解的に表示 · 制御する, *ParaGraph*[25] などのパフォーマンスモニタも並列計算用の GUI ツールである. これらのツールは, GUI を主に出力表示目的で用いているため, GRIX が入力用ツールである点で目的が異なってくる. しかしパフォーマンスモニタは, 複数のプロセッサの状況を直観的に表示する技術は特に洗練されている. 更に品野の研究 [26] では, 超並列と呼ばれるような数百, 数千のプロセッサをグループ化を用い表示するパフォーマンスモニタを提案している. 今後の GRIX の GUI の強化, 更には超並列のサポートの際に, これらパフォーマンスモニタのインターフェイスは大いに参考になりうる.

第 4 章

システム設計

この章では, GRIX の各サブシステムの行う処理の詳細について述べる. なお一般に, 並列計算の分野ではプロセッサのことをノードと表現する場合が多いが, 以下で使うノードという表現は, GUI でいうノード, つまりプロセッサを GUI 画面上で抽象的に表したもののことを指すことにする.

4.1 GUI による入力

GRIX では, 入力機構に GUI を用いたことによりプロセッサ間通信の直観的な入力が可能となってる. また出力側にもグラフィカルな出力を用いることにより, ユーザに対する最適化結果の直観的な理解を可能としている. 次に示す図 4.1,4.2が, GRIX の入出力画面のスナップショットである.

Graphical Input System は, ユーザがプロセッサ間の送受信関係の記述する際の処理を行う. その中でプロセッサを表すノードは円で, そのプロセッサ間の通信関係は矢印で表される. 図 4.1の中で, 縦に並ぶノードがプロセッサの集合, 同じ列の横に並ぶノードは同じプロセッサを示し, 左側が送信, 右側が受信ノードとなる. また送受信をするデータのサイズやアドレス等は, テキスト入力を行う機構により入力される.

4.1.1 マクロ的視点からの入力

プロセッサ間通信の送受信関係を入力する際, ユーザがイメージするその状況の一つに, 並列計算を行う全プロセッサを視点にいれたものが考えられる. つまり, 並列実行プロセッサ台数があらかじめ分かっていて, それを前提にプロセッサ間通信の入力

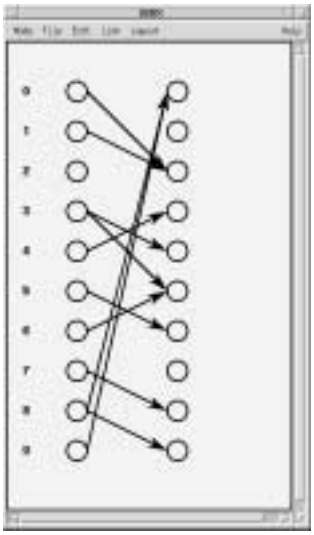

図 4.1: 入力画面

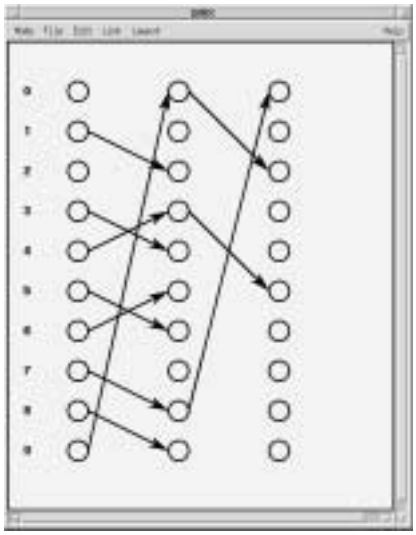

図 4.2: 出力画面

を行っていく場合である. この様な状況下での入力を, 並列計算をする計算機, または計算機の集合全体を入力の単位とすることから, **マクロ的視点からの入力**と呼ぶことにする.

図 4.1は, 10 台のプロセッサで固定された環境での入力を行ったものである. この例でマクロ的な入力の手順を説明する. まずユーザは, システムに対し描画するノードの数を通知しなければならない. これはシステムを立ち上げる際に, 入力視点の選択 (図 4.3) と同時にシステム側から問い合わせが来るので (図 4.4), ユーザはそれに答える形式となる. ユーザがノードの数を通知すると, システムは図 4.5に示す初期画面を表示する. この画面で送信側のノードの脇に表示された番号は, 0 から始められたプロセッサの相対ノード番号を示している.

ユーザはこの画面上に, 送信ノードから受信ノードへマウスドラッグする操作で矢印を描いていく. この操作は, 入力画面上に矢印を描く最も基本的な動作であり, ユーザはどんなプロセッサ間通信の関係でも, この操作で自然に記述できる. さらに入力システムには, しばしば並列プログラミング中に現れる, 通信パターンを記述するための入力用のオプションも用意している. オプションによる入力については, 後節

図 4.3: マクロ的視点入力の宣言

図 4.4: ノード数入力ダイアログ

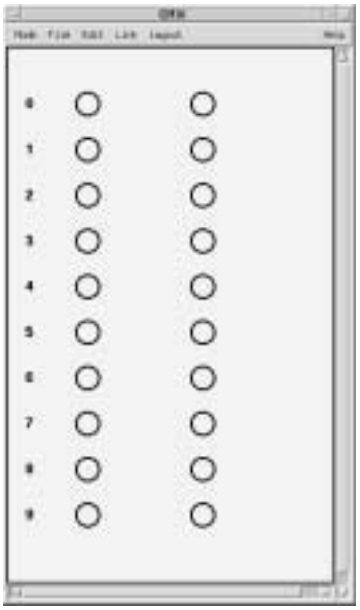

図 4.5: 10 台のプロセッサでの初期入力画面

4.1.3で述べる.

4.1.2 ミクロ的視点からの入力

プロセッサ間通信をユーザがイメージする際には, マクロ的なイメージではなく, 全プロセッサの中の一つに注目した視点で考える状況がある. 例えば, 「ID が偶数のプロセッサは -2,+1,+3 のプロセッサにデータを送信する」というような Scatter のイメージをユーザが持った場合である. この状況下での入力を, マクロ的視点に対し並列計算をする計算機の集合の 1 つのプロセッサに着目する入力視点なので, **ミクロ的視点の入力**と呼ぶことにする.

この例で, ミクロ的視点からの入力法を説明する. まずシステムは, ミクロ的視点の入力の宣言 (図 4.6) の際に描画したいノードの数をダイアログ (図 4.7) を表示して聞いてくる. この例では, owner[1]であるノードからの距離が -2 であるものから +3 のものまで必要となるから, 最低 6 個のノードの表示が必要である. よって表示ノード数 6 を, ダイアログを通してシステムに伝えなければならない.

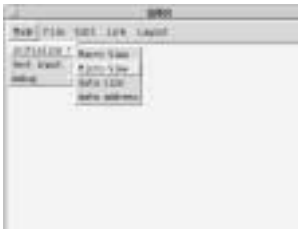

図 4.6: ミクロ的視点入力の宣言

図 4.7: ノード数入力ダイアログ

それを受け取ると, システムは 6 組の塗り潰しなしの円を表示する (図 4.8左上). この画面でユーザが owner のノードを選択すると, owner のノードは強調色で塗り潰され, それ以外のプロセッサを示す円の横には owner から見たそのノードへの距離が表示される (図 4.8右). 仮にここで, ユーザがノードの数を増やしたいと考えたとする. その際には, ユーザがメニューから「`Edit` → `add nodes`」を選ぶことにより (図

[1] ミクロ的視点の入力の際に, ユーザが選択するデータを配信 (または受信) する 1 つのノードを owner と表現する.

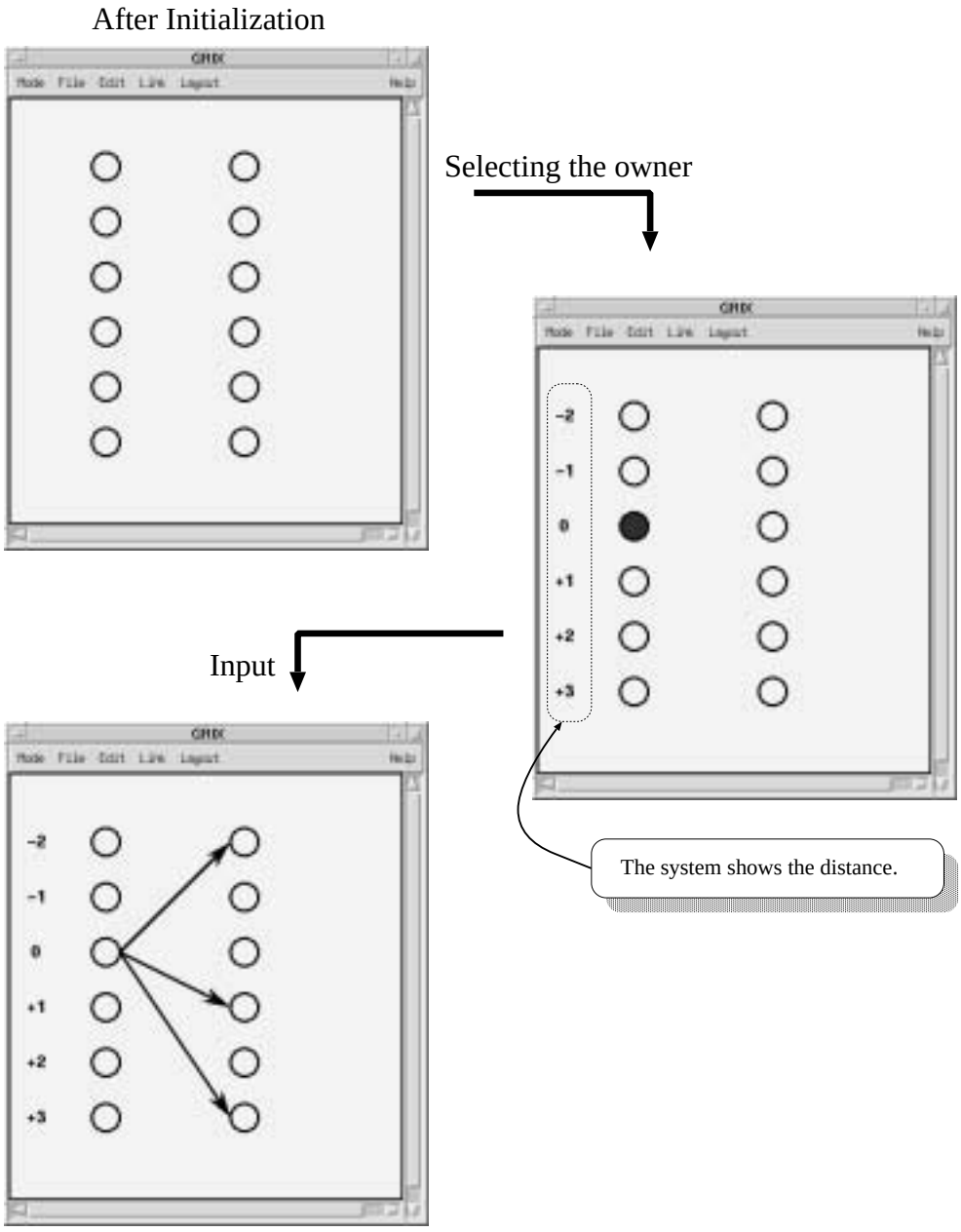

図 4.8: ミクロ的視点からの入力

4.9), システムが追加するノードの数を聞いてくるため, そこに必要なノードを入力することになる (図 4.10). そして新たに owner を選択することにより, 変更された距離が表示される (図 4.11).

図 4.9: ノード追加宣言

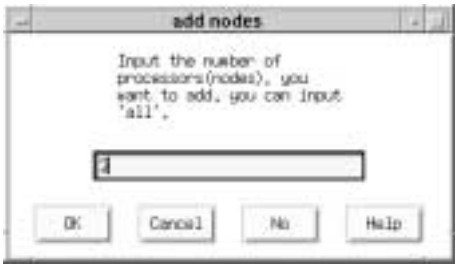

図 4.10: 追加ノード数入力ダイアログ

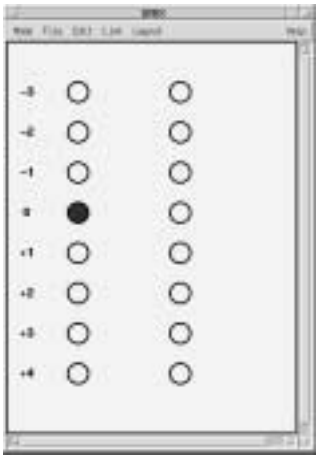

図 4.11: owner 変更後

この様に入力画面上に owner を選択した後に, マクロ的視点からの入力と同様にユーザは入力を行う (図 4.8左下).

ではどの owner がアクティブであるかを入力するにはどうすればよいか. この例の場合「ID が偶数のプロセッサ」がアクティブである. アクティブノードを定義する際には, メニューの「`Edit` → `define active nodes`」を選択することから始まる. デフォルトでシステムのメニューに選択肢として用意されているものは, 「`all`(全てアクティブ)」, 「`1 node`(ある 1 つのノード)」「`even`(ID が偶数)」, 「`odd`(ID が奇数)」の 4 つである (図 4.12).

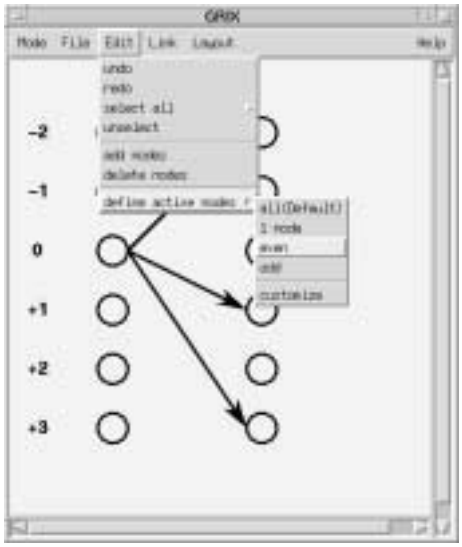

図 4.12: アクティブノードの定義

それ以外の場合にはその下の「`customize`」を選択し, ユーザ自身がシステムが出すダイアログの中に新たに定義することになる (このダイアログは「`1 node`」選択時にも現れる). 例えば, 「0,1」「4,5」「8,9」「12,13」 · · · がアクティブ, つまり 2 個のノードを 1 つのグループとしてそれらが 1 つおきにアクティブであったとする (図 4.13).

この定義の際にシステムは, ユーザにプログラム中の条件文で用いるような形式での定義を要求する. この場合プログラム中では

```
if (((id / 2) % 2) == 0){
     .....
}
```

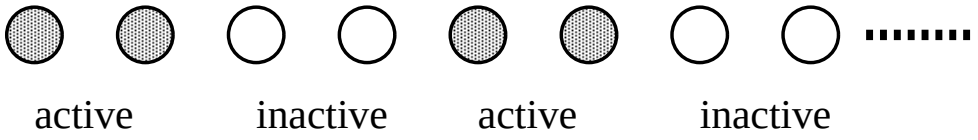

図 4.13: アクティブノードパターンの例

がアクティブノードの判別に用いられる条件文と予想できる. ユーザは入力箇所に

```
((id /2) % 2) == 0
```

を書けばよい. ただし, ここでユーザに許されている入力表現は, 自ノード ID を示す変数「id, または ID」と四則演算子, 論理演算子, 及び定数である. この様にして定義した条件は, システムに保存しておくことで繰り返し使えるようになる. なおアクティブノードの定義は, 通信パターンの入力の前後どちらでも構わない. またこの操作を行わないと, システムは全てがアクティブノードだとして処理を進める.

以上が Scatter の入力例だが, 逆に Gather を入力したい場合には, 最初の owner 選択の際にユーザが受信側のノードを選択することで可能となる. しかしミクロ的な入力の際には 1 つの制約がある. それは, ユーザが選択した owner 以外のノードからの送信 (または owner 以外のノードへの受信) が行えないことである (図 4.14).

以上は定量的に表示するノードの数が分かっている場合だが, 例えば「ノード ID が奇数のものが『全ての』ノードに対してデータを送信する」といった場合にはどうすればよいか. この『全ての』という情報を表示する場合には, 表示ノード数を入力する際に「ALL」と書けばよい. 「ALL」入力時の入力画面の初期状態は, 図 4.15の形式となる. この画面では, owner として選択できるノードは, 中央の送信・受信いずれかのもののみである. 上下に描かれた円は, 上のものがマイナス方向にあるノードを, 下のものはプラス方向にあるノードを抽象的に表現しているだけである. 更に, 基本的にこの状態では, owner を送信側にした場合 Broadcast を, 受信側にした場合には All Gather のみが入力されると想定できる. よって送受信いずれかの owner を選択すると, ユーザは矢印を描く事なく, Broadcast(図 4.16) か All Gather(図 4.17) の表示画面に切り替わる.

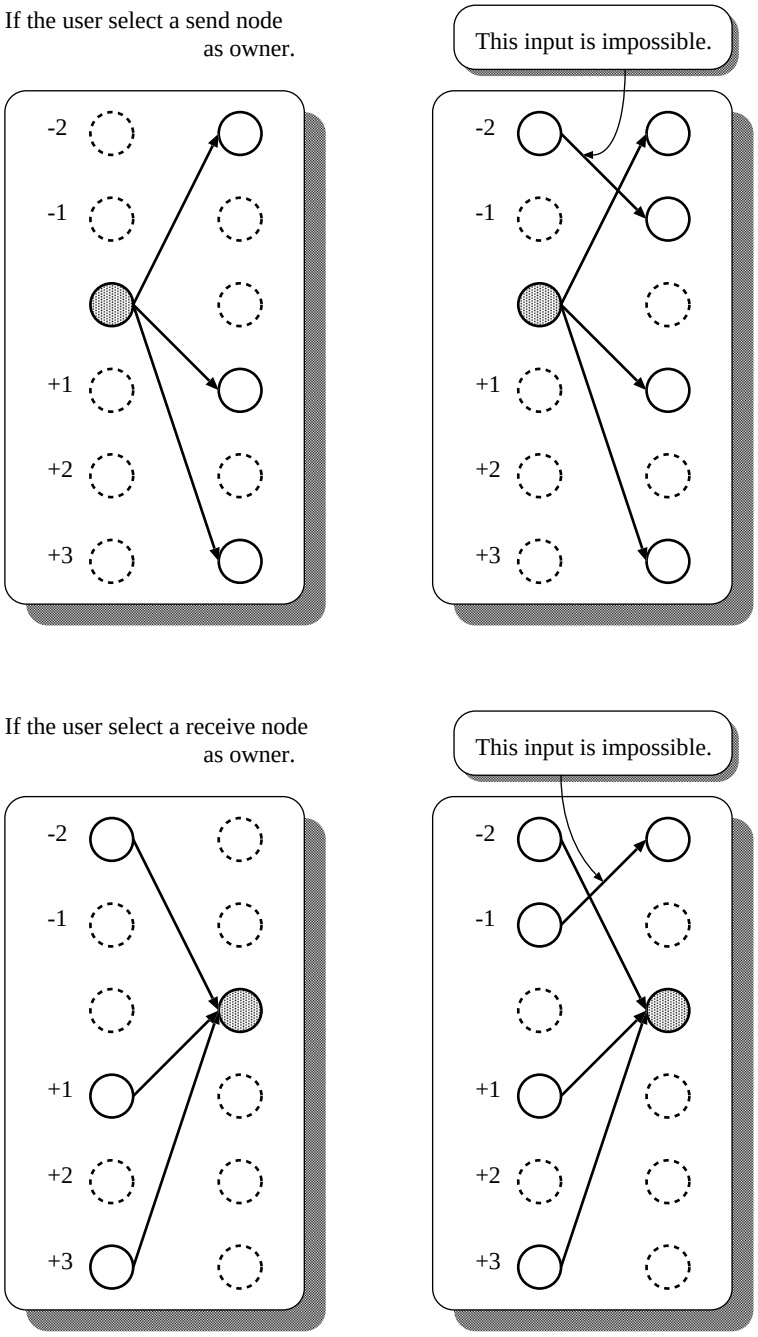

図 4.14: ミクロ的視点からの入力の際の制約

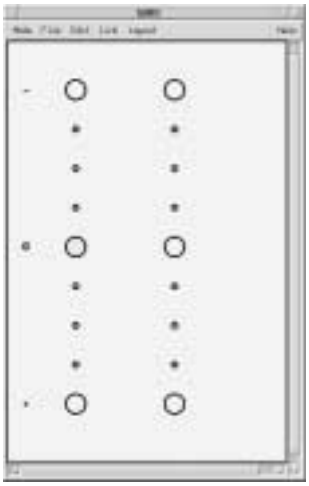

図 4.15: 'ALL' 入力時

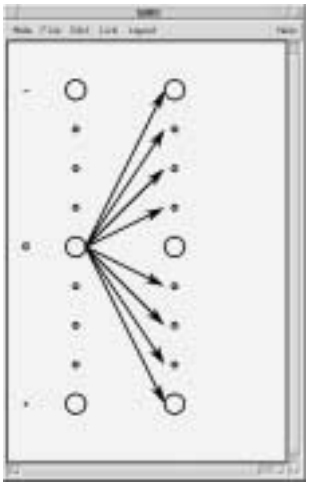

図 4.16: Broadcast

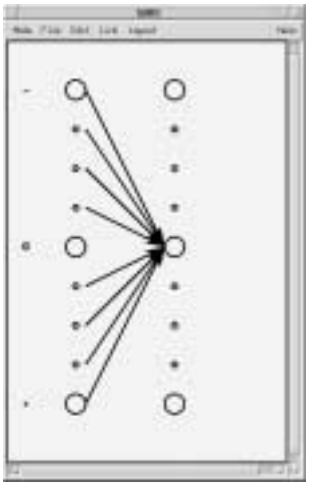

図 4.17: All Gather

この後, ユーザがアクティブノードを設定することにより, 入力が完了する. 例えば All-to-All Broadcast の場合は, 図 4.16の後メニューから「`define active nodes` → `all`」を行い, 1-to-All の Broadcast の場合は同様の画面の後, メニューから「`define active nodes` → `1 node`」として, そのとき現れるダイアログに送信するノードの ID を入力することになる.

4.1.3 入力オプション

GRIX には, しばしば並列プログラミング中に現れる通信パターンを記述しやすくするために, またその他の状況にも簡単に対処できるように, 以下の入力用のオプションが用意されている.

- Broadcast, Scatter, Gather
- All-to-All Broadcast, Shift
- Unlink
- テキストを用いた入力

Broadcast, Scatter, Gather の入力に関するオプション

ユーザが Broadcast や Scatter といった, 1 つのプロセッサから複数のプロセッサへデータを配信する通信パターンを入力する場合には, 図 4.18,4.19, 4.20のような手順での入力も可能である. まずユーザは図 4.18の様に, 1 つの送信ノードと複数の受信ノードを選択する. そしてメニューからの操作 (図 4.19), もしくはキーボードの'e'(Edit の頭文字を取った) を押すことにより, 図 4.20の様に描画される. 逆に Gather の様なデータを集める通信の場合には, 複数の送信ノードと 1 つの受信ノードを選択し, 「Link → gather」または同様に'e' を押すことにより描画される. 以上の操作はマクロ的視点の入力, ミクロ的視点の入力いずれの場合でも可能である.

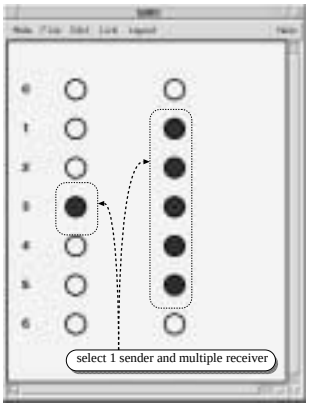

図 4.18: ノード選択

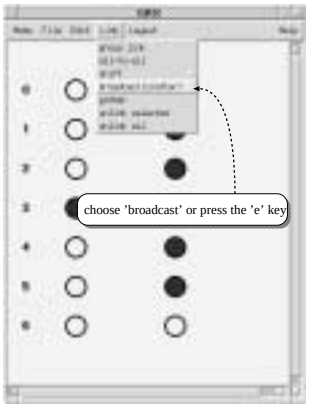

図 4.19: リンク命令

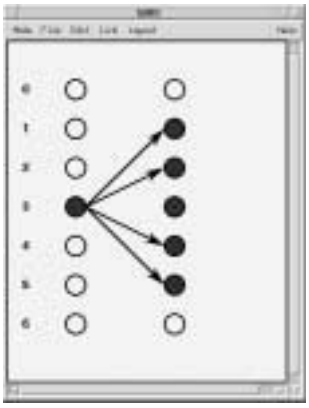

図 4.20: 描画

All-to-All Broadcast, Shift に関する入力オプション

マクロ的視点の入力の際には, 選択したノードに対し All-to-All Broadcast, Shift の通信パターンの入力がオプションによってサポートされている. まずユーザは, 送信ノード側のノードを複数選択する (図 4.21左上). そして Broadcast のオプション同様に, 'e' のキー入力によって All-to-All Broadcast のパターンが描画される (図 4.21右上). その後更にもう 1 度'e' を押すと +1 ストライドの Shift パターンに変わる. 繰り返し'e' を押すことにより, そのストライド幅は +1 ずつインクリメントされ, 全てのストライドが表示された次の入力で初期状態に戻る.

また, ノード選択後メニューから「Link → all-to-all」を選択することにより, All-to-All Broadcast の描画も可能である (図 4.21左上 → 右上). 同様に Shift もメニューから選択し, 描画することも可能である. ただし, その際にシステムがデフォルトで用意しているストライド幅は +1 と -1 だけで, 任意のストライド幅を入力する場合には, その下の「n」を選択後表示されるダイアログに, ユーザがそのストライド幅を入力することになる (図 4.21中央).

矢印を消す操作 –Unlink

システムには, 選択されたノードに繋っている矢を消去するオプションも用意されている (図 4.22). その際には, ユーザは描画の後に, 任意のノードを選択. その後, メニューから「Link → unlink」を選択することにより, そのノードにつながれている矢は全て消去される.

テキストを用いた入力

次にテキスト入力を用いた, 送受信関係の入力について述べる. GRIX には, 専用のテキスト入力用の機構を用意している. テキスト入力でノード選択, 通信パターンの入力はメニューから「Mode → textinput」を選択し, その中から「select nodes → send (またはreceive)」 (図 4.23), 「communication」 (図 4.24) 各々を用いる. その際に表れるダイアログ (図 4.25,4.26) に, 以下の例に示すようなノード選択と, 通信パターンの表現で入力していく.

ノード選択

- **1-5,7,9,10**: ノード番号 1 から 5,7,9,10 を選択
- **All**: 全てのノードを選択
- **All except 2-4,6**: ノード番号 2 から 4 と 6 を除く全てのノードを選択
- **80% random**: 全ノード数の 80% にあたる数のノードをランダムに選択
- **7 random**: 10 個のノードの内 7 個のノードをランダムに選択

通信パターン

- **all to all**: 選択ノード群で All-to-All Broadcast

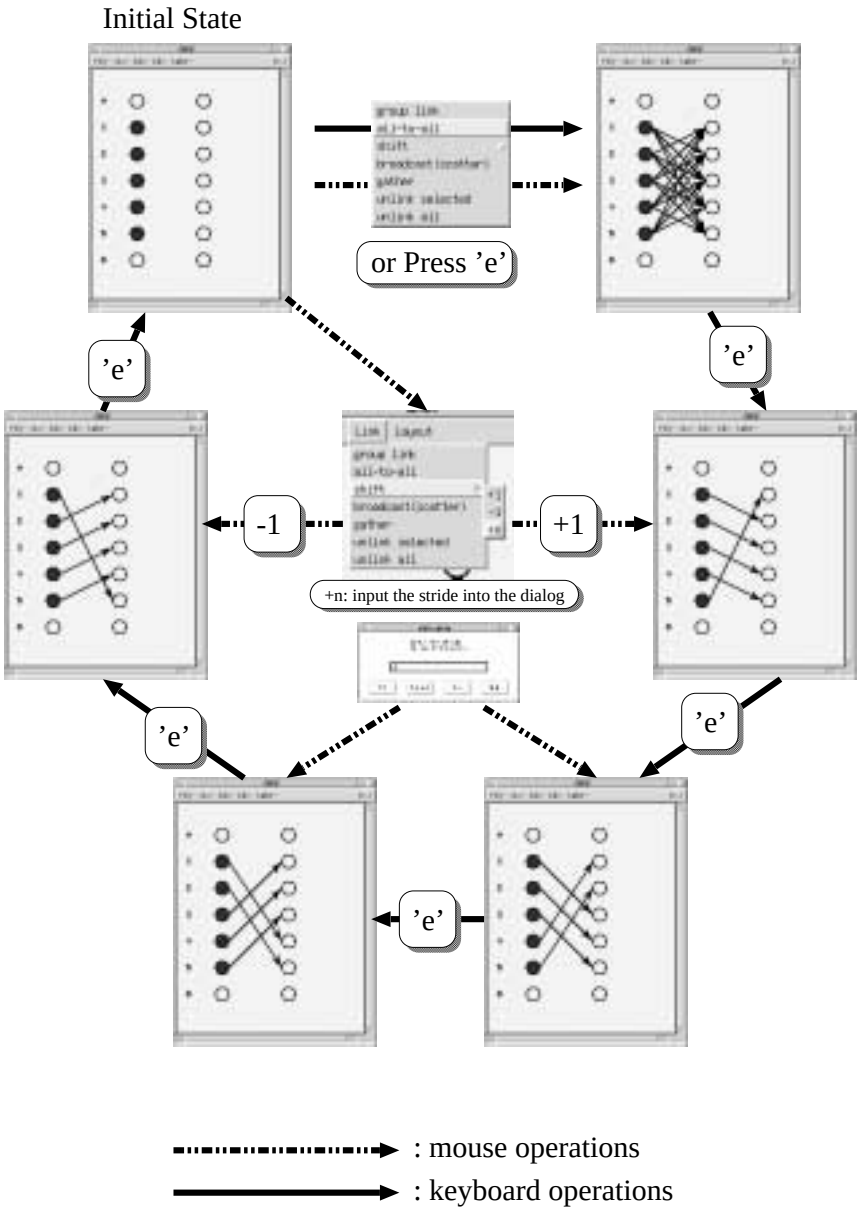

図 4.21: all-to-all Broadcast と Shift の入力

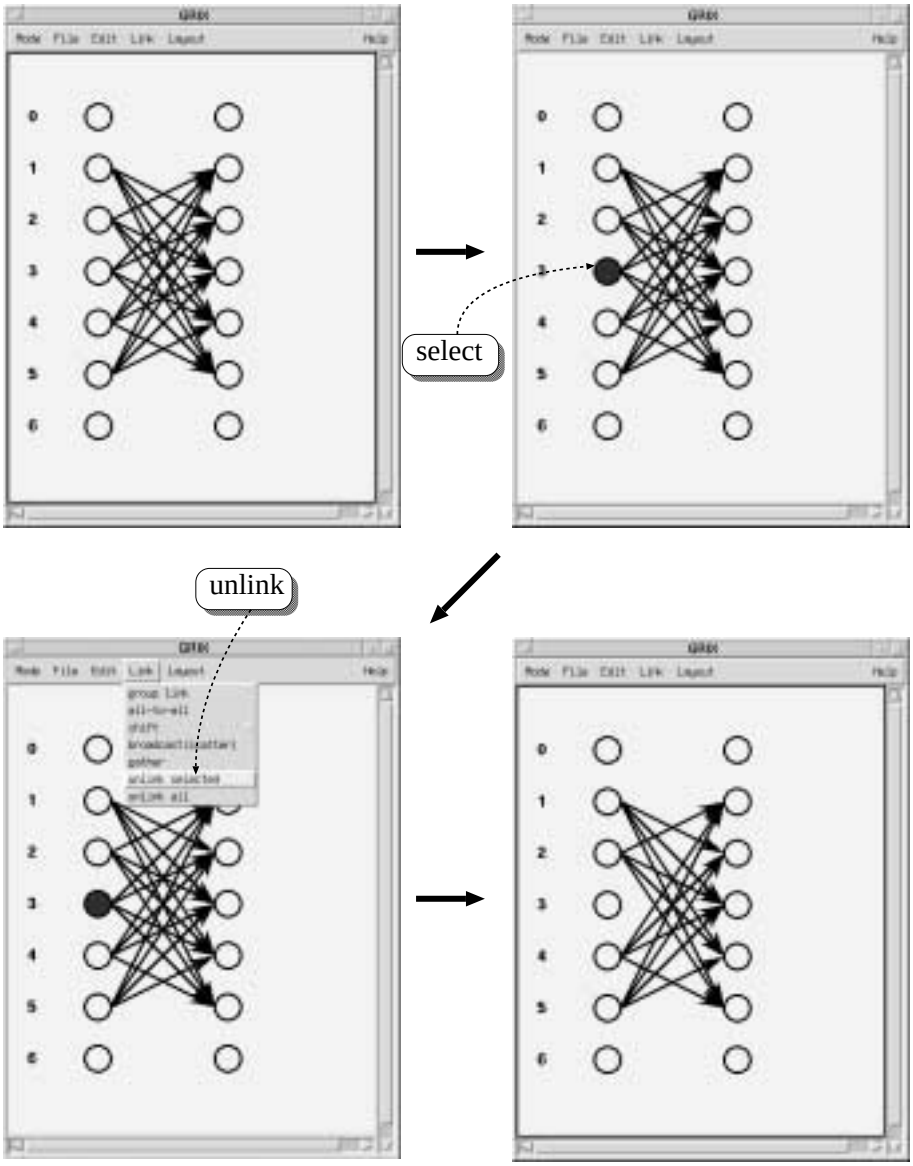

図 4.22: unlink オプション

図 4.23: ノード選択ダイアログの呼び出し

図 4.24: 通信パターン入力ダイアログの呼び出し

図 4.25: ノード選択ダイアログ

図 4.26: 通信パターン入力ダイアログ

- **+2 shift**: 選択ノード群でストライド 2 の Shift 転送
- **1 to 1 random**: 選択ノード群でランダムな 1 対 1 通信

主にテキスト入力に有効なものは, ランダムな選択とランダムな通信に対してであり, これはマクロ的な入力の際に効果的である. また, このテキスト入力とそれ以外のオプションの操作は併用することも可能である.

4.2 最適化

GRIX は入力された通信パターンに対し, その実行時の効率を向上させるための最適化処理を行う. 特にランダムな通信の入力が行われる可能性のある, マクロ的視点の入力の際の最適化は, あらゆる並列実行環境に対応するというモチベーションから, それに対応できるものを用意した. 最適化命令は図 4.27の様に, 入力画面の状態でメニューから「`File` → `optimize`」を選択することで可能である.

図 4.27: 最適化命令

4.2.1 マクロ的入力の際の最適化

マクロ的な入力が行われた際の最適化は, 同期通信を基本とした以下の 5 段階に進められる.

(1) 一定のストライドを持つ Shift 転送ごとにグループ分けをする.

(2)(1) の各グループを順番に各ステップに割り振る.

(3) 各ステップでセンドレシーブの各ノードが重ならないものどうしをオーバーラップさせる.

(4) 受信データの再利用が可能であれば、それにあわせて変換をする.

(5) 各ステップにおけるアクティブノードの情報をビットパターンに変換する.

最適化 (1),(2) で最低限のコード化は可能となる. 更に最適化 (3) を行うことにより, 実行ステップの減少がはかられる. この (1)〜(3) で図 4.1で示された入力例は, 以下の図 4.28で示される動作により出力画面の図 4.2を得る.

この図 4.2では入力画面同様に, 縦に並ぶノードがプロセッサの集合, 同じ列の横に並ぶノードは同じプロセッサを示すが, 横方向に右に向かった時間軸が存在する. まず最左のノード群が最初の送信プロセッサであり, その右のノード群がその際の受信プロセッサとなる. そしてその受信ノード群は新たな送信ノード群となりその右のノード群への送信を行うことを示す. またマクロ的視点の入力の際の最適化は, 同期

通信を前提としているので, この図の送受信の切り替わる際には同期が張られている. この際その同期で区切られた部分を「ステップ」と定義する.

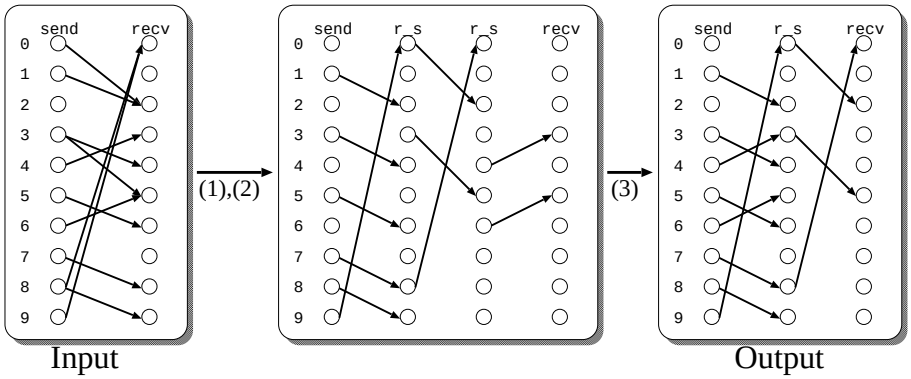

図 4.28: Optimizer の基本動作 (1)∼(3)

(4) で示した受信データの再利用は, 部分的な Broadcast と他の通信パターンが排他的に混在しているときに用いられる. 図 4.29で示す例は, 10 台のプロセッサのうち 6 台が Broadcast, 4 台が All-to-All の Broadcast を行っている場合の, 受信データの再利用を用いた最適化である.

最適化 (1)∼(3) では, 計算される必要な通信ステップ数は, 各ノードの持つ矢印の最大本数分だけになる. この方法の場合, Broadcast の動作部分では配信されたデータを受信したプロセッサは各ステップで 1 個ずつしか増えていかない. しかしここでデータを受信したプロセッサが, 受信したデータをまだ受信していないプロセッサに対し送信するようにすると, 受信済みのプロセッサは 2 の巾乗で増えていくことになる (図 4.30).

よって最適化 (4) では, 図 4.30のように Broadcast の動作が変換され, 以下の表 4.1に示す Broadcast 部分の通信ステップの減少が可能となる.

また図 4.29の例では, 通信ステップの 3 番目において Broadcast 部分のストライド幅の変換も行われている. 図 4.30の動作では, ステップ 3 番目のノード 4,5 への送信

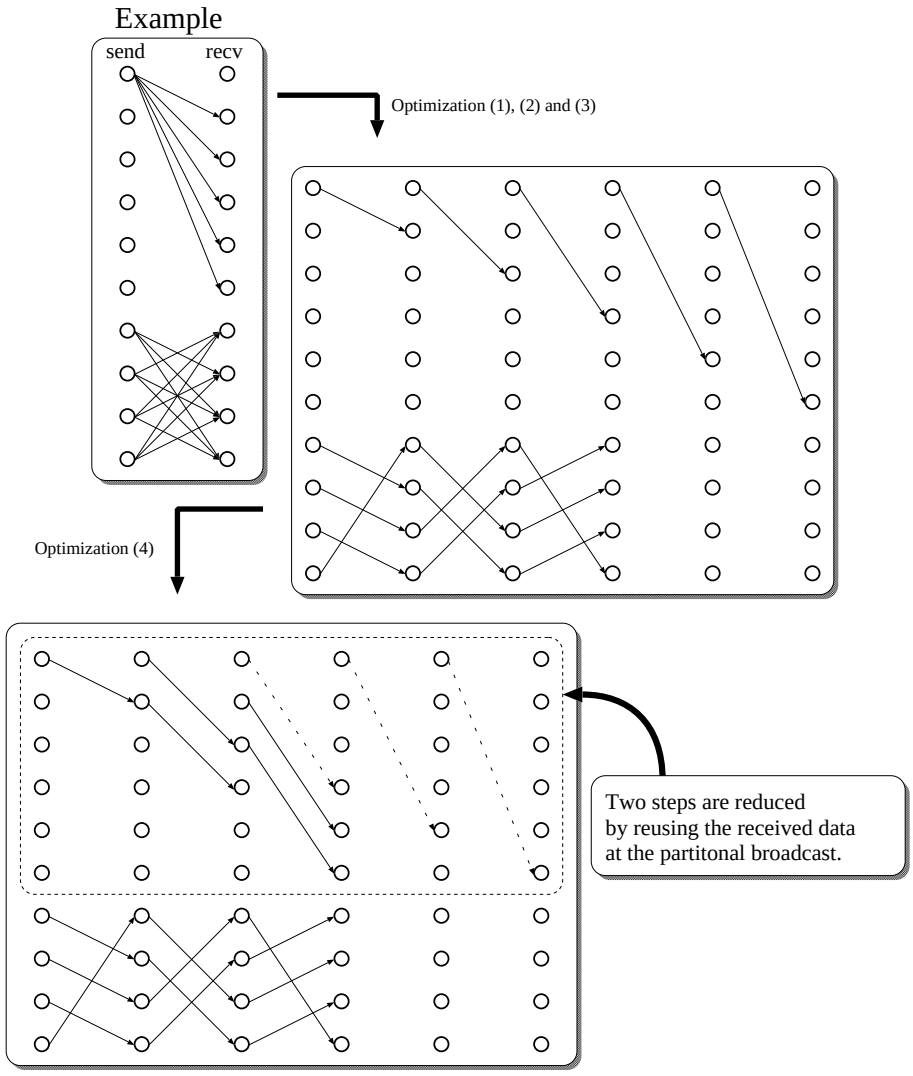

図 4.29: 受信データの再利用

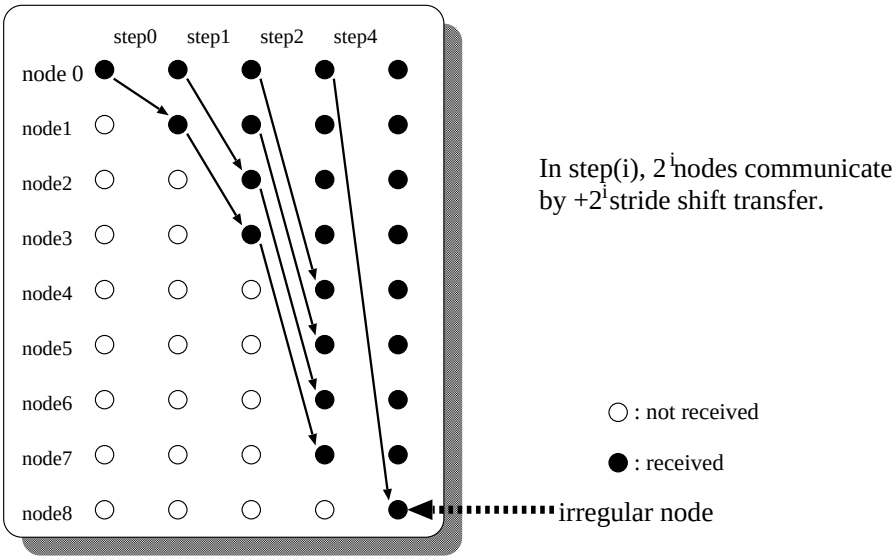

図 4.30: 受信データの再利用を用いた Broadcast の変換

表 4.1: 最適化動作 (4) による通信ステップの変化

最適化動作	通信ステップ数
(1)∼(3)	$N_b - 1$
(4)	$log_2 N_b, log_2 N_b + 1$

N_b:Broadcast 実行プロセッサ数

はノード 0,1 から行われることになる．この場合, ステップ 3 における通信ストライドの種類は,

- ノード 0,1 からノード 4,5 へのストライド幅 +4
- ノード 7 からノード 10 へのストライド幅 +3
- ノード 8,9,10 からノード 7,8,9 へのストライド幅 -1(+9)

の 3 種類となる． しかし, ノード 4,5 への送信をするデータはノード 0∼3 全てが持っているので, この場合ノード 2,3 からノード 4,5 への送信に変更すると, ノード 7 からノード 10 への送信とストライド幅が一致するので, 条件文によって分けられる通信の種類が 1 つ減り効率的である．この様に図 4.30の変換における最終ステップがイレギュラーノードに対する通信である場合, 他の通信のストライド幅と同じになりうるものを探し, 可能であれば先の変換を行う．

最適化 (5) により, ターゲットコードのセンドレシーブのアクティブノードの判定が, 1 回の条件文で済む． 例えば $\{0 \sim 9\}$ のノードの内, 送信アクティブノードが $\{0, 3, 8\}$ である場合, そのビットパターンは $d\_bit = 2^0 + 2^3 + 2^8 = 1 + 8 + 256 = 265$ となり, コードの中でアクティブノードの判定は

```
d_bit = 265;
if (1 && (d_bit >>= myid))  /*myid: 自ノード ID*/
```

また受信アクティブノードの判定は, 送信者を判定し, 同じように d_bit を用いて

```
nprocs = 10;                /*nprocs: ノード数 */
if ((sender = (myid-stride)) < 0)
  sender += nprocs;     /*stride: シフト転送幅 */
else if (sender >= nprocs)
  sender -= nprocs;
if (1 && (d_bit >>= sender))
```

となる．それ以外にも, もし各ステップでインアクティブノードが極めて少数で, かつどのステップもオーバーラップしていなければ, 全てをアクティブノードとしてしまう最適化も行い, 上のような条件文を除くこともできる．また, Broadcast や Summation といった特殊な通信のみが発生する場合は, 上の最適化を行わずに環境に用意された関数を用いるようにする．

4.2.2 ミクロ的入力の際の最適化

ミクロ的な入力の際に行われる最適化は, 送信, または受信ノードの条件を判定することが主な作業となる.

図 4.8の様な Scatter の入力の際には, 送信側のアクティブノード情報はあらかじめユーザによって決定されているので, 受信側のどのノードがどの種類の送信を受け取るかを判別する. 図 4.8の場合, 送信する相手はアクティブノードから -2,+1,+3 の距離を持つノードである. 送信側のアクティブノードの判定は, 実行時の自プロセッサの ID をmyid, 全プロセッサ台数をnprocs とし, 各々の通信の距離を

```
int stride[3] = "-2, 1, 3";
```

という様にstride という変数に与えておく. そうすると送信側は

```
if ((myid % 2) == 0)
   for (i = 0; i < 3; i++)
       SEND to (myid + stride[i]), label(i);
/* 値 i のラベルをつけて +stride[i] 離れたノードに送信 */
```

という様な実行コードになる. 一方受信側は

```
for (i = 0; i < 3; i++){
    if ((sender = myid - stride[i]) < 0)
        sender += nprocs;
    else if (sender >= nprocs)
        sender -= nprocs;
    if ((sender % 2) == 0)
        RECV from sender, label(i);
/* 値 i のラベルのついた送信を -stride[i] 離れたノードから受信 */
}
```

という実行コードになる. この条件文は, マクロ的入力の際の最適化 (5) と同様に, 受信側で送信ノードを判定し, それがアクティブであるかを判定するものである.

「ID が奇数のノードが全プロセッサに送信する」場合は,

送信側:

```
if ((myid % 2) == 1)
    for (i = 1; i < nprocs; i++)
        SEND to (myid + i), label(i);
/* 値 i のラベルをつけて +i 離れたノードに送信 */
```

受信側:

```
for (i = 1; i < nprocs; i++){
    if ((sender = myid - i) < 0)
        sender += nprocs;
    else if (sender >= nprocs)
        sender -= nprocs;
    if ((sender % 2) == 1)
        RECV from sender, label(i);
/* 値 i のラベルのついた送信を -i 離れたノードから受信 */
}
```

という様になる.

以上のような送信側主体のものではなく,「ID が偶数のプロセッサが距離 -2,+1,+3 のプロセッサからデータを受信する」といった Gather のような受信側主体の入力の場合は, 逆に送信側で受信ノードを判定する. この場合のコードは Scatter 同様に

```
int stride[3] = "-2, 1, 3";
```

を用い,

送信側:

```
for (i = 0; i < 3; i++){
    if ((receiver = myid - stride[i]) < 0)
        receiver += nprocs;
    else if (receiver >= nprocs)
        receiver -= nprocs;
    if ((receiver % 2) == 0)
        SEND to receiver, label(i);
}
```

受信側:

```
if ((myid % 2) == 0)
    for (i = 0; i < 3; i++)
        RECV from (myid + stride[i]), label(i);
```

となる. つまり条件文の表記が, 送信側主体のものと逆になる.

以上のミクロ的視点の際の最適化時には, マクロ的なものと違い, 同期ステップを設けたりそれをオーバラップさせたりはしない. これは, 必ず送受信いずれかのノードで衝突が起きているため, 同期ステップに振り分けてもオーバラップは不可能となることと, ラベルによって通信の種類を分けることにより, 非同期に送受信が可能なためである. もちろんそのためには, メモリ空間に各々の送信データに対する受信領域を確保する必要がある.

整理すると, ミクロ的な入力の際に Optimizer が Code Generator に渡す情報は,

- owner は送受信どちらのノードか
  → 条件文生成のため
- 送信 (または受信) する相手のノードは全てなのか, またそうでなければいくつなのか
  → `nprocs` を用いる, もしくは配列`stride[]` の要素数を求める
- 全ノード選択でなければ相手のノードの距離はいくつか
  → `stride[]` の各々の値
- owner の条件は何か

の 4 つとなる.

4.3 ユーザへのフィードバック

図 4.27での最適化命令の後, ユーザはその結果を図 4.2の様な出力画面で知ることができる. この表示で, それぞれのノードがそれぞれのステップ内で, どのように通信をしているかが直観的に理解できる. 図 4.2はマクロ的視点の入力に対する最適化結果であるが, ミクロ的な入力に関しては, 実際の実行プロセッサ台数が不定で, 更に非

同期実行である．また最適化によって計算されるのは, 主にアクティブノードの情報であるため, ユーザへは適当な実行台数の場合の実行状況を表示する．図 4.31は, 図 4.8の入力例を 10 台の実行プロセッサを想定したときの動作である．

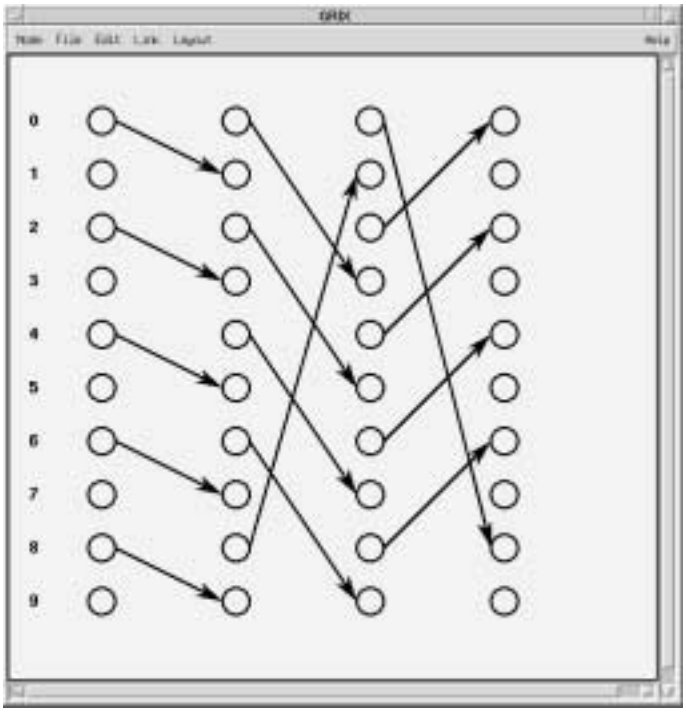

図 4.31: プロセッサ台数 10 のときの実行状況

この図 4.31の場合, 同期実行の様な表示になっているが, 実際は非同期に次々に送信し, 次々に受信する動作を行っている．またこの出力画面の主な目的の 1 つに, テキストで入力されたアクティブノードの情報を図解的に表示することもある． この様に, ユーザに対して最適化結果を, 入力画面同様に GUI 表示することで, 実際の実行コードの書かれかたが直観的に理解でき, またデバッグなどのサポートにもなりうる． またこの状態から入力画面に戻るには, 図 4.32の様に, メニューから「`File` → `re-edit`」を選択することで可能である．

図 4.32: 入力画面へ戻る

4.4 実行可能コードへの変換

GRIX には, PVM や MPI などのような並列実行環境が多種多様化している現状に対応するための, 自動コード生成を行う処理系が含まれる. それにより, 各種環境の仕様を理解したうえでのプロセッサ間通信のコーディングではなく, 実行イメージだけでのコーディングが可能となる. さらに各種環境用の自動コード生成の処理系を用意することにより, あらゆる環境に対応したコード生成が可能となる. 現在このコード生成を行う処理系は, PVM のコード生成を行うものを用意している. 実行コードを生成するには, 最適化出力画面で図 4.33の様に, メニューから「`File` → `generate` → `PVM`」を選択することで可能である.

図 4.33: コード生成命令

マクロ的視点の入力から得られた図 4.2の場合, 生成される PVM の実行コードは以下のようになる.

```
int sender;
int d_bit[2];
```

```
/*---------- STEP1 ----------*/
d_bit[0] = 85;
d_bit[1] = 80;
/*---------- SEND ----------*/
if (!(1 && (d_bit[0] >>= SPMD_procnum))){
  pvm_pkbyte(&send_area, datasize, 1);
  pvm_send(SPMD_tids[(SPMD_procnum + 1)% SPMD_nprocs], 0);
}
if (1 && (d_bit[1] >>= SPMD_procnum)){
  pvm_pkbyte(&send_area, datasize, 1);
  pvm_send(SPMD_tids[(SPMD_procnum + SPMD_nprocs - 1) % SPMD_nprocs], 1);
}
/*---------- RECV ----------*/
if ((sender = SPMD_procnum - 1) < 0)
  sender += SPMD_nprocs;
else if (sender >= SPMD_nprocs)
  sender -= SPMD_nprocs;
if (!(1 && (d_bit[0] >>= sender))){
  pvm_recv(SPMD_tid[sender], 0);
  pvm_upkbyte(&recv_area, datasize, 1);

  memcpy(&store_area, &recv_area, datasize);
  store_area += datasize;
}
if ((sender = SPMD_procnum + 1) < 0)
  sender += SPMD_nprocs;
else if (sender >= SPMD_nprocs)
  sender -= SPMD_nprocs;
if (1 && (d_bit[1] >>= sender)){
  pvm_recv(SPMD_tids[sender], 1);
  pvm_upkbyte(&recv_area, datasize, 1);

  memcpy(&store_area, &recv_area, datasize);
```

```
  store_area += datasize;
}

pvm_barrier(SPMD_GROUP, SPMD_nprocs);

/*---------- STEP2 ----------*/
d_bit[0] = 265;
/*---------- SEND ----------*/
if (1 && (d_bit[0] >>= SPMD_procnum)){
  pvm_pkbyte(&send_area, datasize, 1);
  pvm_send(SPMD_tids[(SPMD_procnum + 2) % SPMD_nprocs], 2);
}
/*---------- RECV ----------*/
if ((sender = SPMD_procnum - 2) < 0)
  sender += SPMD_nprocs;
else if (sender >= SPMD_nprocs)
  sender -= SPMD_nprocs;
if (1 && (d_bit[0] >>= sender)){
  pvm_recv(SPMD_tids[sender], 2);
  pvm_upkbyte(&recv_area, datasize, 1);

  memcpy(&store_area, &recv_area, datasize);
  store_area += datasize;
}
```

また, ミクロ的視点の入力である図 4.8の場合の PVM コードは以下のようになる.

```
int stride[3] = "-2, 1, 3";
int skip = 0;

/*---------- SEND ----------*/
if ((SPMD_procnum % 2) == 0){
   for (i = 0; i < 3; i++){
       pvm_pkbyte(&send_area, datasize, 1);
       pvm_send(SPMD_tid[(SPMD_procnum + SPMD_nprocs + stride[i])
```

```
                        % SPMD_nprocs], i);
    }
}
/*---------- RECV ----------*/
for (i = 0; i < 3; i++){
    if ((sender = SPMD_procnum - stride[i]) < 0)
        sender += SPMD_nprocs;
    else if (sender >= SPMD_nprocs)
        sender -= SPMD_nprocs;
    if ((sender % 2) == 0){
        recv_area += datasize * skip;
        pvm_recv(SPMD_tid[sender], i);
        pvm_upkbyte(&recv_area, datasize, 1);
        skip++;
    }
}
```

このプログラムには, PVM における SPMD プログラミング用の SPMD ライブラリ[2]で用意されている変数を用いている． SPMD ライブラリは簡単に PVM 環境で SPMD プログラミングをするユーティリティプログラムである．以下に示す 3 つの変数が, このプログラム中に用いられている SPMD ライブラリで用意された変数である.

- `SPMD_nprocs`: プロセッサ数
- `SPMD_procnum`: プロセッサ番号 (0 origin)
- `SPMD_tids`: プロセッサ番号 (0 から`SPMD_nprocs-1`) をインデックスとして, task id を値とする配列

SPMD ライブラリのより詳細な説明は, 参考文献 [14] に記してあるので, そちらを参照されたい.

[2]参考文献 [15] の著者, 田井秀樹氏の製作

第 5 章

結論

5.1 まとめ

本論文ではユーザフレンドリで, かつ高速なプロセッサ間通信を実現するための環境として, プロセッサ間通信に特化したビジュアルプログラミングシステム GRIX を提案した. GRIX では図解的に最適化処理を構築したことにより, 複雑なテキストを用いた処理を行わずに, グラフ上での直観的な処理が行える. 入力機構に GUI を用いたことにより, より簡単で直観的なプロセッサ間通信の入力が可能になる. また最適化結果の GUI 出力によりフィードバックが, ユーザの実行コードに対する直観的な理解, 更にはデバッグなどの補助を行う. 更に, 自動コード生成を行う処理系によって, ユーザが並列実行環境の仕様を理解した上でのコーディングをする必要もなくなる. 今後, コード生成機構の複数環境への対応を施すことにより, 環境の多様化 · 複雑化を意識せずにユーザはプロセッサ間通信の記述が可能となるであろう.

5.2 今後の課題

今後の課題として, 今回試作したシステムの評価が挙げられる. 評価のポイントは, 最適化による通信実行時間の定量的な評価と, 複数のユーザから得る使用感についての定性的な評価の 2 つが挙げられる. この評価の結果により, 今後の処理系の実装が改良されていくことになる. また, 現在の GRIX システムはプロセッサ間通信の記述用に完全に独立した構成を持っている. 今後はプロセッサ間通信以外のプログラム編集と連動させるために, 既存の emacs や vi といったエディターと連動させて, よりインタラクティブなプログラミング環境を構築していく必要がある.

謝辞

本研究を行うにあたり, 筑波大学電子 · 情報工学系田中二郎教授には, 多くの御指導, 助言を頂き, 心より深く感謝します.

また, 本研究の礎となる知識を得るにあたって御世話になった, 図書館情報大学図書館情報学部中田育男教授と筑波大学電子 · 情報工学系山下義行助教授にも深く感謝します. そして, 共に研究生活を送ってきた三浦元基氏, 宮城幸司氏, 遠藤浩通氏, 飯塚和久氏をはじめとする IPLAB の各員にも深く感謝します. 特に三浦元基氏には, システムの実装から貴重な協力を頂き, 深く感謝します.

加えて, 日頃の生活で特に親交が深かった貴重な友人である高級言語研究室の立堀道昭氏と宮元秀典氏に感謝します. またソフトウェア研究室の皆様, 初の国際会議で海外での道標となってくれた筑波大学電子 · 情報工学系千葉滋講師にも, ここで感謝の意を示します.

そして何よりも両親に深く感謝します.

参考文献

[1] 中澤 喜三郎, 中村 宏, 朴 泰祐:
超並列計算機 CP-PACS のアーキテクチャ, 情報処理, Vol.37, No.1, pp.18-28, Jan. 1996

[2] 中田 育男, 山下 義行, 小柳 義夫:
超並列計算機 CP-PACS のソフトウェア, 情報処理, Vol.37, No.1, pp.29-37, Jan. 1996

[3] リモート DMA 転送 使用の手引 -C-, HITAC

[4] リモート DMA 転送 使用の手引 -FORTRAN-, HITAC

[5] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck and Vaidy Sunderam:
PVM, The MIT Press, 1994

[6] William Gropp, Ewing Lusk and Anthony Skjellum:
USING MPI, The MIT Press, 1994

[7] IPCA-parallel:environments:
http://www.hensa.ac.uk/parallel/environments

[8] James R. Mason and Philip J. Hatcher and Steve Chappelow:
*Optimizing Irregular Communication Patterns in UNH C**, University of New Hampshire, May. 1994

[9] A. Lapadula and K. Herold:
A retargetable C compiler and run-time library for mesh-connected MIMD multicomputers*, University of New Hampshire, 1992

[10] 超並列 C 言語 NCX 言語仕様書 Version 3

[11] 過 敏意:
並列化コンパイラにおけるデータ分割と再分割のアルゴリズムの研究, 筑波大学大学院博士課程工学研究科博士論文, July, 1998

[12] S. D. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan:
Multi-Phase Redistribution: A Communication-Efficient Approach to Array Redistribution, A longer version is available as an Ohio State University CIS Technical Report.

[13] 酒寄 保隆, 三浦 元基, 田中 二郎:
GRIX: 並列プログラミングにおけるプロセッサ間通信のコーディング支援システム, 日本ソフトウェア科学会第 15 回大会論文集, pp.381-384, Sep. 1998

[14] Yasutaka Sakayori, Motoki Miura, and Jiro Tanaka:
GRIX:Visual Programming System for Interprocessor Communications, Proceedings of the 10th IASTED International Conference PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS (PDCS'98), pp.503-508, Oct. 1998

[15] 田井 秀樹:
超並列計算機用 NCX 言語処理系の試作, 筑波大学大学院修士課程理工学研究科修士論文, Mar. 1997

[16] 酒寄 保隆:
並列コンパイラ用のランタイムライブラリの実装, 筑波大学情報学類卒業論文, Mar. 1997

[17] 酒寄 保隆, 田中 二郎:
並列コンパイラ用のランタイムライブラリの実装法, 日本ソフトウェア科学会第 14 回大会論文集, pp.313-316, Oct. 1997

[18] S. Fortune and J. Wyllie:
Parallelism in Random Access Machines, In Proceedings of the 10th Annual Symposium on Theory of Computing, pp.114-118, 1978

[19] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken: *LogP: A Practical Model of Parallel Computation*, CACM, Vol.39, No.11, November 1996

[20] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken: *LogP: Towards a Realistic Model of Parallel Computation* 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, May 1993

[21] Peter Newton and James C. Browne:
The CODE 2.0 Graphical Parallel Programming Language, Proceedings of ACM International Conference on Supercomputing, July, 1992

[22] P.Newton:
Visual Programming and Parallel Computing, Delivered at Workshop on Environments and Tools for Parallel Scientific Computing, May, 1994.

[23] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam:
Graphical Development Tools for Network-Based Concurrent Supercomputing, Proceedings of Supercomputing 91, pp.435-444, 1991

[24] G. A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam:
PVM 3 Users Guide and Reference Manual, Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1993

[25] Michael T.Heath:
ParaGraph: A Tool for Visualizing Paformance of Parallel Programs, Univ of Illinois, Jennifer Etheridge Finger, Oak Ridge National Laboratory, June 1994

[26] 品野 竜太:
プロセッサのグループ化による超並列プログラムの可視化, 筑波大学大学院修士課程理工学研究科修士論文, March 1998