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

GRIX:Visual Programming System for Interprocessor Communications

酒寄 保隆[†] Yasutaka SAKAYORI
三浦 元基[††] Motoki MIURA
田中 二郎[†††] Jiro TANAKA

†筑波大学博士課程工学研究科
Doctoral Program in Engineering, University of Tsukuba
††筑波大学修士課程理工学研究科
Master's Program in Science and Engineering, University of Tsukuba
†††筑波大学電子情報工学系
Information Science and Electronics, University of Tsukuba

概 要

本論文では、並列計算機特有のプロセッサ間通信に対するビジュアルプログラミングシステム **GRIX** (GRICSS:**GR**aphical **I**nterprocessor **C**ommunication **S**upport **S**ystem) を提案する。本システムは、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。

1 はじめに

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^* [1]やその拡張である NCX [2]といったものがある。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 Pictorial 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 GRIX を提案する。

2 システム構成

GRIX のシステム構成を図 1に示す。 GRIX 処理系は Graphical Input/Output System, Optimizer, Code Generator の 4 つの処理系と、その 4 つの処理系で入出力される 3 つの中間ファイルから構成される。

2.1 Graphical Input/Output System

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

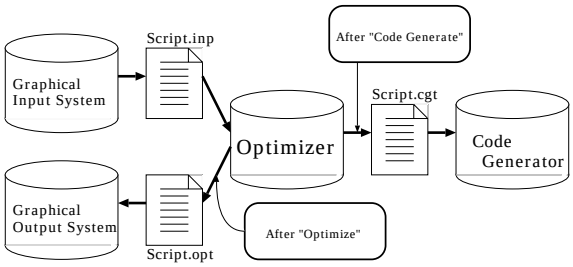

図 1　GRIX のシステム構成

解を可能としている。次に示す図 2,3が、 GRIX の入出力画面である。

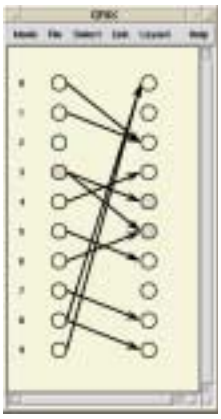

図 2　input view

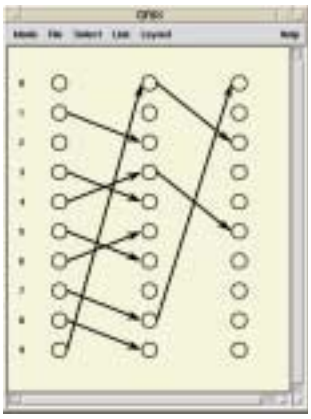

図 3　output view

2.1.1　入力システム

Graphical Input System は、ユーザがプロセッサ間の静的な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 2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。

送受信関係を記述することは、ユーザが矢印を入力画面に描いていく GUI 入力、または後述のテキスト入力を行うことで可能とさせる。まず、入力画面に直接矢印を描いていく方法について述べる。矢印を描くための最も基本となる操作は、送信ノードから受信ノードへマウスドラッグすることである。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。さらに我々は、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 (図 4) を用意した。この図 4に示されるように、マウスドラッグによるノードの選択とキーボードからの「e」 (Edit の頭文字) の入力により、 Broadcast, All-to-All Broadcast, Shift の通信パターンの入力が容易に行える。これ以外にも、ノードを選択した後でメニューから unlink を選択することにより、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している。

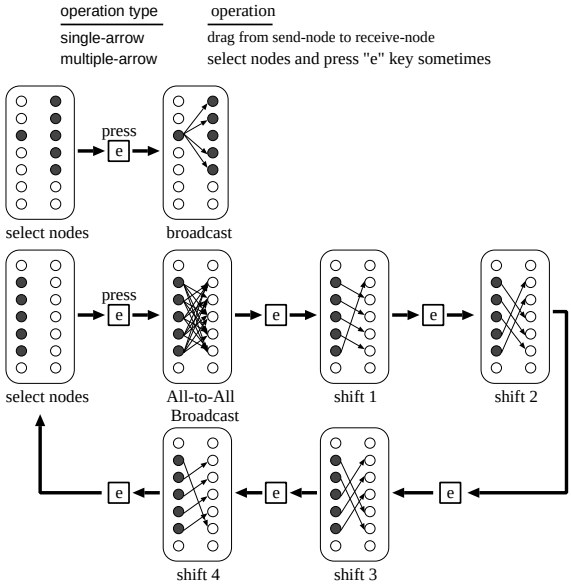

図 4　入力オプション

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。それを用いたノード選択と、通信パターンの入力形式の例を以下に示す。

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
- **7/10 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
- **+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

また、これら 2 種類の操作は併用することも可能である。

2.1.2 出力システム

Optimizer は、通信プログラムの実行状況をグラフで表すための、中間ファイルを生成する。そのファイルを読み込み、その結果の表示 (図 3) を行うシステムが Graphical Output System である。この図 3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。また後述の 2.2で示すように、 Optimizer は同期通信を前提とした最適化を行っているので、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。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。

2.2 Optimizer

Optimizer は、 [6]を参考にした最適化理論に基づき、入力情報の中間ファイルを変換し、 Graphical Output System と Code Generator に読ませる中間ファイルを生成する処理系である。

2.2.1 Optimizer の基本動作

Optimizer の基本的な動作は、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。

(1),(2) で最低限のコード化は可能となる。 (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 2で示された入力例は、以下の図 5で示される動作により出力画面の図 3を得る。

この後次節 2.2.2で述べる、受信データの再利用が可能であればそれを行い、 (5) の動作に移る。 (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

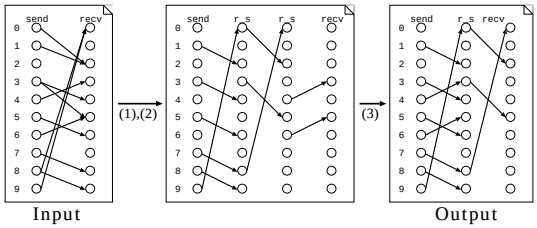

図 5 Optimizer の基本動作 (1)∼(3)

```
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。

2.2.2 受信データの再利用

上の (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混在しているときに用いられる。図 6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。 Optimizer の基本動作 (1)∼(3) では、この最適化による通信ステップ数は、各ノードの持つ矢印の最大本数分だけ必要になる。ここで (4) の最適化を加えることにより、図 7のように Broadcast の動作が変換され、以下の表 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 6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。

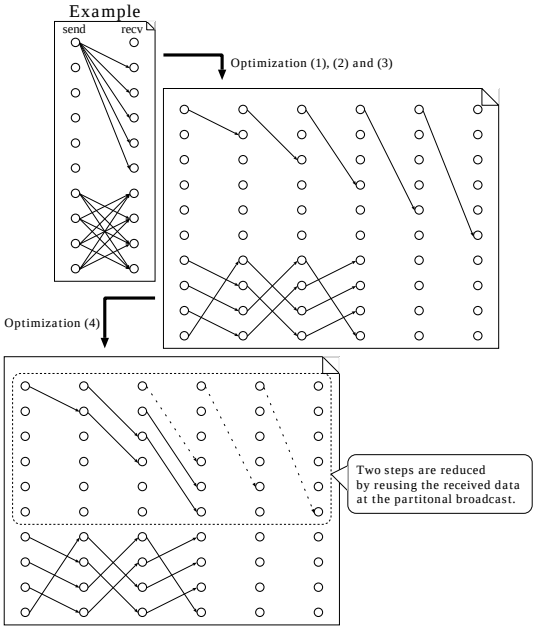

図 6　受信データの再利用

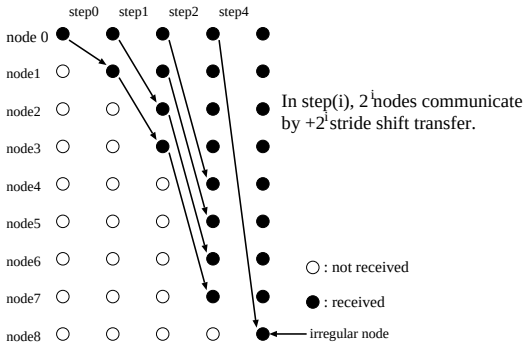

図 7　受信データの再利用を用いた Broadcast の変換

表 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

2.3　Code Generator

GRIX には、PVM [4]や MPI [5]などのような並列実行環境が多種多様化している現状 [3]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 Code Generator は、PVM のコード生成を行うものを用意している。この Code Generator によって、Optimizer によって生成されたコード生成用の中間ファイル`Script.cgt` が、実行可能なコードに変換される。

3　結　　論

本論文では、並列プログラミングにおけるプロセッサ間通信についての GUI を用いたコーディング支援システム GRIX の概要について述べた。
GRIX では Pictorial な処理を採用したことにより、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。また最適化を施した結果を GUI とコードの両方で出力することで、ユーザに対する最適化結果のフィードバックが直観的に示され、また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。また今後 Code　Generator を複数用意することにより、コード生成用の中間ファイルを保存しておくことで、あらゆる環境のコード生成が可能になると考えられる。今後は各処理系のカスタマイズ、特に Optimizer に関して非同期通信のサポートが必要である。それにともない、$LogP$ [7]などのモデル化理論を応用した Optimizer の構築も考えている。

参考文献

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

[2]　超並列 C 言語 NCX 言語仕様書 (Version 3)

[3]　IPCA-parallel:environments: *http://www.hensa.ac.uk/parallel/environments/index.html*

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

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

[6]　S.D.Kaushik, C.-H.Huang, J.Ramanujam and P.Sadayappan: *Multi-Phase Redistribution: A Communication-Efficient Approach to Array Redistribution*, submitted for publication

[7]　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