Programming Environment Specified for Describing Interprocessor Communications based on the Operations on Graphical User Interface

Yasutaka Sakayori, Buntarou Shizuki and Jiro Tanaka
Institute of Information Sciences and Electronics
University of Tsukuba
Tennodai 1-1-1, Tsukuba, Ibaraki 305-8573, Japan.

Abstract *In this paper, we propose the GRIX system, which is a visual programming system specified for interprocessor communications in parallel computing. We will outline the system and explain its Graphical User Interface (GUI) based operations. GUI-based operations enable users to input the structure of interprocessor communications with intuitive images. Using this system to generate code eliminates user tasks of translation from images produced by users into textual program codes. Moreover, most programmers construct the pictorial images of communications' behavior on paper or in their minds, when faced with interprocessor communications during parallel programming. By adopting the GUI interaction with the sense of the programmers' pictorial tracing, this system can provide an effective programming environment for most programmers.*

Keywords: Interprocessor Communications, Visual Programming System

1 Introduction

In this paper, we propose the GRIX system [1], which is a visual programming system specified for interprocessor communications occurring in programs for Message-Passing parallel computing.

Occasionally, poor implementations of interprocessor communications cause frequent synchronizations and waste computing resources. Furthermore, there are many methods for coding the sentences of interprocessor communications, like PVM [2], MPI [3] and native functions implemented only for specific computers. A major point we emphasize is to eliminate complexities that cause bring confusion. For instance, in order to realize the simplest communication, which is a send-receive pair, by using PVM functions, programmers must complete at least four procedures, packing data, sending data, receiving data and unpacking data. Moreover, some variables included in each function must be complex to achieve high performance. This tendency is stronger in more native environments.

The ideal method of describing interprocessor communications is with simple information such as "who", "where" and "how." We can accomplish this by using the Visual Programming method [4]. The Visual Programming method is a programming form using visual images like icons, figures and animations, instead of textual specifications.

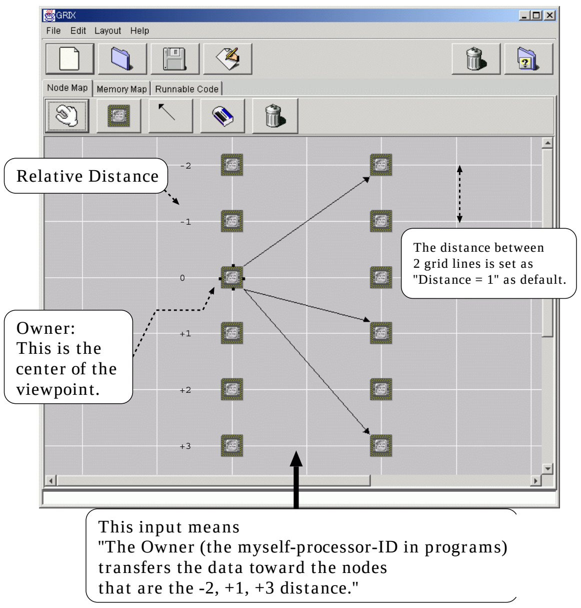
2 Visual Programming System GRIX

When programmers implement interprocessor communications based on the Single Program / Multiple Data (SPMD) paradigm or Master-Slave style, they describe the outlined movement of interprocessor communications on paper or in their minds. The GRIX system adopts a GUI that is consistent with programmers' pictorial plans for the input interface and generates the textual programs from their input.

GRIX is used if interprocessor communications occur in SPMD programming. The separation between interprocessor communications and other calculations can be clearer in SPMD than in other parallel programming styles. and the GRIX system can be expected to be more effective.

The GRIX system is implemented with Java and can be classified into two subsystems: the GUI Engine, for interactions between users and the system, and the Code Generation Engine, for generating textual codes. Though the current Code Generation Engine is produces PVM or MPI codes, it can be adopt for generating code in any environment by preparing a code generation module for each environment.

Figure 1 provides an overview of the GRIX system.



The distance between 2 grid lines is set as "Distance = 1" as default.

Owner: This is the center of the viewpoint.

This input means "The Owner (the myself-processor-ID in programs) transfers the data toward the nodes that are the -2, +1, +3 distance."

Figure 1: Overview of GRIX system

The basic operation in a GRIX system is "drawing the figure" representing the flow of interprocessor communications. In practical terms, users place the nodes that represent PE and draw edges that represent data flows between two PEs on the canvas by mouse operations.

The GRIX system also enables grid information on the canvas of Fig. 1 to be used effectively by both users and the system. Users can employ the grid for beautiful and reasonable node placing by automatic snapping to the nearest grid point. The system uses the grid information to infer the users' intention to some extent.

3 Interface to Describe Interprocessor Communications

In this section, we explain the visual interface of the GRIX system that is specified for interprocessor communications.

3.1 Input of Send-Receive Relationship among Processing Elements

We now explain how to input the send-receive relationship with the following Scatter example: "the processors with even-number IDs send the data to the processors that are -2,-1 or +3 distance from them." Figure 1 is an appropriate input for the example. Users perform four tasks to reach this input.

1. Map 6×2 nodes represented by CPU icons on the grid points on the canvas.

A new node is created by selecting the toggle button with the CPU icon's image, holding this state, and clicking the left button of the mouse on the canvas. The node is then automatically snapped to the nearest grid point of the coordinate clicked by the user. At first, user must create six nodes in a line to show the distance between -2 to +3 by those operations. As shown in Fig. 1, the user must create two lines of nodes to represent time flow from left to right by adding the meaning of "send nodes" for the left nodes and the meaning of "receive nodes" for the right ones. Therefore, the two nodes aligned on the horizontal line represent the same processor.

2. Select a certain node that is the center of viewpoint from the send nodes.

One node is the center of the viewpoint, in other words, the processor node referred to as "myself" in SPMD programs. We call it the node *Owner*. Selecting the third (from top) node from the left nodes as the Owner node adds the meaning of relative distance as shown in Fig. 1. Basically, the width between two grid points represents the distance of nodes as 1.

3. Draw data-flow edges from Owner to the receive nodes.

Users describe the data flow edges from the send node to the receive node by mouse-dragging operations. They can describe any relationship among nodes naturally by these operations.

4. Input the condition that activates Owner node.

The condition of Owner must now be defined. In this example, the condition of the active nodes is *"ID is an even number."* Users have to inform the system of this condition with a representation that can insert the condition sentence. For instance, if the variable for referring to "myself" is set as `SPMD_procnum`, the input must be `SPMD_procnum % 2 == 0`. If the condition is not defined, the system judges the condition to be "all nodes are active."

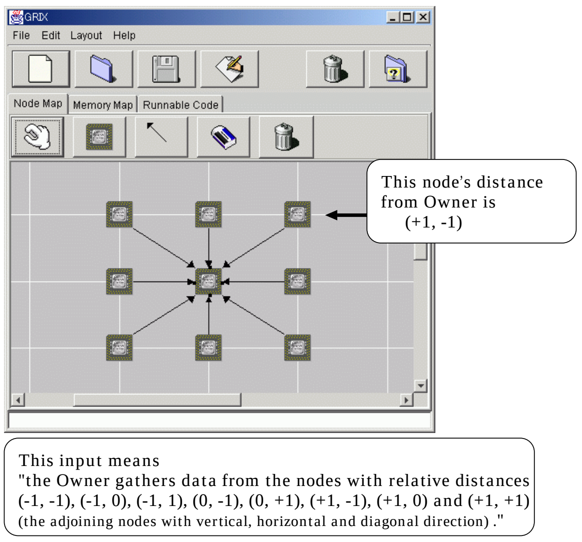
Gather is input by selecting Owner from the receive nodes and drawing edges from the send nodes to one Owner.

For Scatter or Gather, the system does not allow users to describe an arrow that is not connected with Owner. With this restriction, however, it is possible to input data assuming a 2-dimensional(2D) node ID such as shown in Fig. 2.

Figure 2 shows communication in which the Owner node gathers data from nodes with relative distances of (-1, -1), (-1, 0), (-1, +1), (0, -1), (0, +1), (+1, -1), (+1, 0) and (+1, +1). In this case, the height between two nodes means the column-direction distance and the width means the row-direction distance.

3.2 Inferring from Grid Information

Nodes and edges are placed on the canvas following the grid points by automatic snapping.



This node's distance from Owner is (+1, -1)

This input means "the Owner gathers data from the nodes with relative distances (-1, -1), (-1, 0), (-1, 1), (0, -1), (0, +1), (+1, -1), (+1, 0) and (+1, +1) (the adjoining nodes with vertical, horizontal and diagonal direction)."

Figure 2: Input with 2D node ID

If those nodes and edges have certain characteristics, the system infers the users' intention from them.

For example, the statements to infer "the input with 1-dimensional(1D) ID" are "two lines of the node group have the same number" and "it can define the edges' direction uniquely, vertically or horizontally."

The upper-most left grid point on the canvas has this attribute, the origin (0, 0). The grid point x to the right of the origin and y below the origin has the attribute (x, y). In Fig. 3, the existence nodes have the attributes (1, 0∼5) and (3, 0∼5). The system infers the existence of two lines of the node from these attributes. The five existing edges start at (1, 2) and end at (3, 0), (3, 1), (3, 3), (3, 4) and (3, 5). The edges thus have the vector attribute, (2, -2), (2, -1), (2, 1), (2, 2) and (2, 3). The system can infer that all the edges are directed to the right because the first elements of all vectors have the same value.

The system can judge the left-lined nodes to be send nodes and the right-lined ones to be receive nodes. By using the grid attribute, the system detects the same result for some kinds of symmetric inputs.

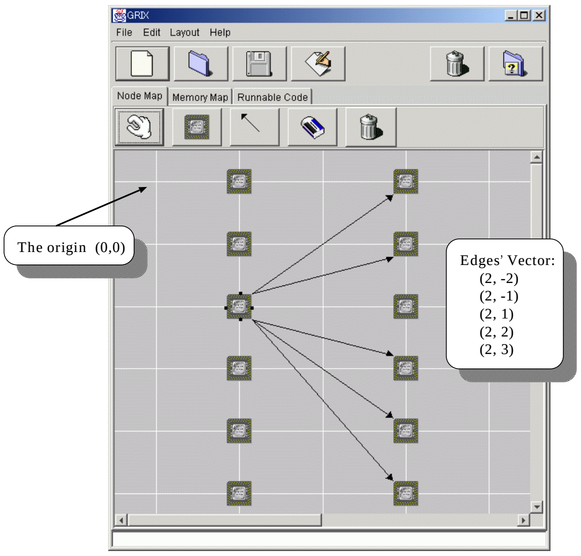


Figure 3: Inference from objects' attributes

4 Code Generation

The GRIX system can automatically generate the actual code with its simple and powerful mechanism. SPMD programs can use the variables for referring to "myself," "the number of all nodes" and "the other existing node." Assuming variables SPMD_procnum, SPMD_nprocs and SPMD_tids[], the generated PVM code for the input of Fig. 1 is as follows.

```
 1: int stride[3] = {-2, 1, 3};
 2:
 3: /*---------- SEND ----------*/
 4: if ((SPMD_procnum % 2) == 0){
 5:   for (i = 0; i < 3; i++){
 6:      pvm_pkbyte(&send_area, datasize, 1);
 7:      pvm_send(SPMD_tid[(SPMD_procnum
 8:               + stride[i] + SPMD_nprocs)
 9:               % SPMD_nprocs], i);
10:   }
12: }
```

In the first line, stride[] is an array having the relative distances to the receive nodes. This is generated by the values from the y elements of the edges' vector (x, y).

Te data should be sent on the condition that "myself is active." Therefore, the conditional sentence in the fourth line are generated.

The fifth line means to send data three times; three is the number of elements in stride[]. The receive node ID which has the distance "*distance*" can be calculated by ("*myself ID*" + "*distance*" + "*the number of nodes*")%"*the number of nodes*." Therefore, the index of SPMD_tids[] in the seventh line is as follows.

```
(SPMD_procnum + stride[i] +
          SPMD_nprocs) % SPMD_nprocs
```

The loop index i is used as the description that correspond with the one of the receive execution (in the ninth line).

In some reception cases, it can be difficult to define the active condition using only "myself ID," such as sending cases, especially if the active send nodes have complicated conditions. Therefore, we selected a strategy that calculates the sender, the node ID that must send to "me," and adds the conditional sentence as in send execution using the sender ID.

The sender ID from the distance stride[i] can be calculated from the node ID that has the distance -stride[i] from "me." Therefore, the code for receiving is as follows.

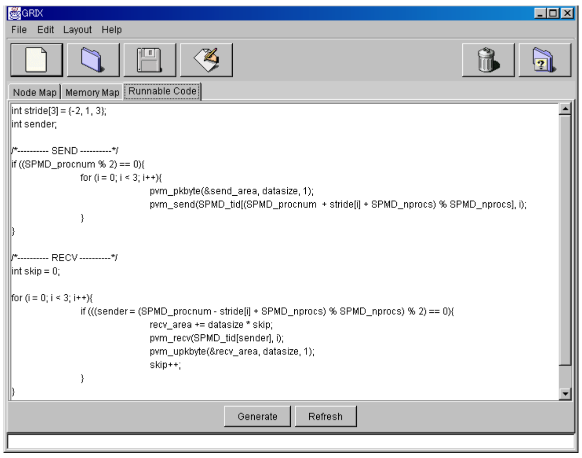
```
/*---------- RECV ----------*/
int skip = 0;

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

The system shows the generated code as shown in Fig. 4.

5 Example Using Jacobi Algorithm

The Jacobi algorithm [5] is one of the most efficient algorithms for parallel computing. The following pseudo code presents one Jacobi algorithm implementation.



```
int stride[3] = {-2, 1, 3};
int sender;

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

/*---------- RECV ----------*/
int skip = 0;

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

Figure 4: Code Viewer

```
 1: begin
 2:   {Boundary Conditions}
 3:   for $i := 1$ to $n$ do
 4:     begin
 5:       $x[0][i] := north[i]$;
 6:       $x[n+1][i] := south[i]$;
 7:       $x[i][0] := west[i]$;
 8:       $x[i][n+1] := east[i]$;
 9:     end
10:   {Initialize values of elements of x}
11:   for $i := 1$ to $n$ do
12:     $x[i] := 50$;
13:   {Refine estimates of x until values converge}
14:   repeat
15:     $diff := 0$;
16:     for $i := 1$ to $n$ do
17:       for $j := 1$ to $n$ do
18:         $newx[i][j] := (x[i-1][j] + x[i][j-1]+$
19:               $x[i+1][j] + x[i][j+1])/4$;
20:     for $i := 1$ to $n$ do
21:       for $j := 1$ to $n$ do
22:         begin
23:           $diff := max(diff, |newx[i][j] - x[i][j]|)$;
24:           $x[i][j] := newx[i][j]$;
25:         end
26:   until $diff < \epsilon$
27: end
```

Parameters n, ϵ and $diff$ represent the data size, the convergence criterion and the maximum change. The values of arrays $north$, $south$, $east$ and $west$ are the boundary conditions. x is the parameter for the solution, and $newx$ is the parameter for the new estimated solution.

One of the parallelizing blocks incorporated into the SPMD program in the pseudo code is the block of lines 16 to 19. We can find the following two strategies for parallelizing that block.

- Node mapping with 1D node ID and column-oriented or row-oriented data distribution.
- Node mapping with 2D node ID and block-oriented data distribution.

Figure 5 presents the communication behavior under 1D node mapping and row-oriented data mapping, using a 16×16 matrix on four nodes.

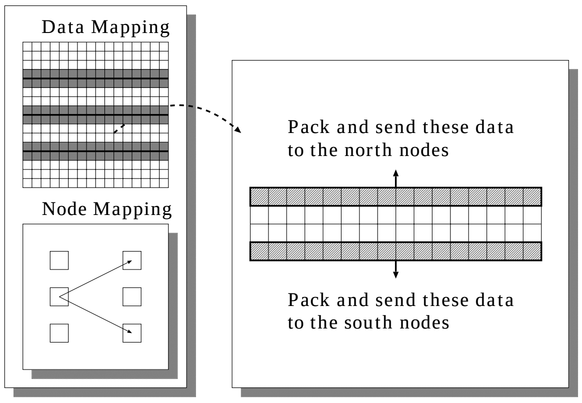

Data Mapping
Node Mapping
Pack and send these data to the north nodes
Pack and send these data to the south nodes

Figure 5: Communication behavior under 1D node mapping

Let us determine the communication cost assuming the Parallel Random Access Machine (PRAM) model [5]. Using the parameters λ as the start up cost, n as the data size (the number of points in each dimension of the data matrix), β as the proportional constant multiplying n, and p as the number of processors, the total communication cost is $4(\lambda + n\beta)$ because the cost of single transfer is $\lambda + n\beta$ and there are two pairs of send and receive transfers on one node. We find that there is no relationship with p under this strategy.

Figure 6 shows the communication behavior under 2D node mapping and block-oriented data distribution, using a 16×16 matrix on 16 nodes.

In this case, the total communication cost is $8(\lambda + n\beta/\sqrt{p})$ because the cost of a single

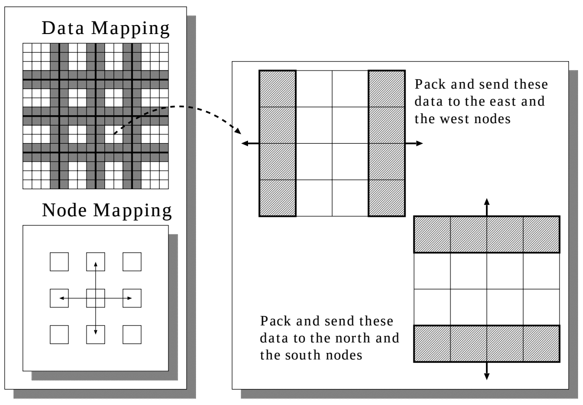

Data Mapping
Node Mapping
Pack and send these data to the east and the west nodes
Pack and send these data to the north and the south nodes

Figure 6: Communication behavior under 2D mesh node mapping

transfer is $\lambda + n\beta/\sqrt{p}$ and there are four pairs of send and receive transfers. Comparing those costs, we find the following trade-off.

$$\begin{aligned} 8(\lambda + n\beta/\sqrt{p}) &< 4(\lambda + n\beta) \Rightarrow \\ n &> \frac{\lambda}{(1-2/\sqrt{p})\beta} \end{aligned}$$

If there are 64 processors and $\lambda = 50\beta$, the trade-off point at which the 2D node mapping and block-oriented data distribution should be superior appears to be a 70×70 matrix because $n > 66.\dot{6}$.

If users do not know the exact data size, they may change the communication behavior from Fig. 5 to Fig. 6. This change can occur when they know the exact data size and do not know the environment parameter β and λ. Moreover, there exists some environments are difficult for showing the formulated performance.

The textual gap caused by this change is not so small. In particular, increasing the number of node ID dimensions generates large textual changes. However, the GRIX system only requires users to perform mouse operations on the canvas.

In Fig. 5, generated parameter stride[] is

```
int stride[2] = {-1, 1};
```

and the node to send is as follows.

```
(SPMD_nprocs + stride[i] + SPMD_nprocs)
                          % SPMD_nprocs
```

The system provides the following advanced functions for a 2D node ID such as shown in Fig. 6. The system provides a 2D array in order to access the nodes mapped with the 2D ID. Most parallel computing environments provide scalar variables for processors' ID. The correspondence between the 2D array and the scalar variables is provided by the following code, assigning 16 nodes (the scalar variables are 0 to 15) on a 4×4 mesh map.

```
int origin = 0
int ncolumns = 4, nrows = 4;
int procID[ncolumns][nrows];

for (i = 0; i < ncolumns; i++)
  for (j = 0; j < nrows; j++)
    procID[i][j] = origin + i * nrows + j;
```

Current indexes of array procID[][] can be accessed by the following code.

```
int x_procID = SPMD_procnum / ncolumns;
int y_procID = SPMD_procnum % nrows;
```

Because of the change of distance from scalar variables to vector variables, the array stride[] has one more dimension. In Fig. 5, stride[] is as follows.

```
int stride[2] = {-1, 1};
```

In Fig. 6, the distance array stride[][] is as follows.

```
int stride[4][2] = {{-1, 0}, {0, -1}, {0, 1},
                    {1, 0}};
```

Furthermore, the send node ID can be found by determining the distance in each index of procID[][] such as following.

```
procID[(x_procID + stride[i][0] + ncolumns)
       % ncolumns][(y_procID + stride[i][1]
                   + nrows) % nrows]
```

The receive node ID is determined in the same way as follows.

```
procID[(x_procID - stride[i][0] + ncolumns)
       % ncolumns][(y_procID - stride[i][1]
                   + nrows) % nrows]
```

The system provides textual changes and reduces users' trial and error work.

6 Related Work

There are some GUI systems for parallel computing. P. Newton and J. C. Browne proposed CODE [6], which is a visual parallel-programming language. G. A. Geist proposes HeNCE [7], which is also a visual parallel-programming language. Newton compared their characteristics in reference [8]. We focus on interprocessor communications in the GRIX system because it represents the most characteristic and difficult point of message-passing parallel programs.

XPVM [9] is a GUI system for controlling and visualizing the PVM programs. GRIX and XPVM have the same characteristics for controlling interprocessor communications, but XPVM is designed for run-time environments, not for editing programs.

7 Conclusion

There are many implementation methods for coding interprocessor communications, but they tend to be complex. Visual programming has great potential for simplifying such difficult programming. We proposed the GRIX system, a visual programming system for simplifying the programming of interprocessor communications. The visual interface specified for the input of interprocessor communications allows users to input such communications intuitively. Also, GRIX can free programmers from complicated tasks and the need to be familiar with many complex interprocessor communications procedures. Moreover, it can help them in both one-time coding and performance tuning trial and error.

GRIX will become a more effective system when combined with existing textual editors.

References

[1] Y. Sakayori, M. Miura, and J. Tanaka. Programming environment specified for interprocessor communications based on graphical user interface. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 2779–2785, June 2000.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM.* The MIT Press, 1994.

[3] W. Gropp, E. Lusk, and A. Skjellum. *USING MPI.* The MIT Press, 1994.

[4] B. A. Mayers. Taxonomies of visual programming and program visualization. In *Journal of Visual Language and Computing, 1(1):*, pages 97–123, 1990.

[5] M. J. Quinn. *PARALLEL COMPUTING Theory and Practice.* MacGraw-Hill, 1994.

[6] P. Newton and J. C. Browne. The code 2.0 graphical parallel programming language. In *Proceedings of ACM International Conference on Supercomputing*, July 1992.

[7] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing 91*, pages 435–444, 1991.

[8] P. Newton. Visual programming and parallel computing. Delivered at Workshop on Environments and Tools for Parallel ScientificComputing, May 1994.

[9] 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, Oak Ridge National Laboratory, 1993. ORNL/TM-12187.