VISUAL PROGRAMMING SYSTEM SPECIFIED FOR THE INTERPROCESSOR COMMUNICATIONS IN PARALLEL PROGRAMS

Yasutaka Sakayori, Motoki Miura and Jiro Tanaka

Institute of Information Sciences and Electronics

University of Tsukuba

Tennodai 1-1-1, Tsukuba, Ibaraki 305-8573 Japan

yasutaka@iplab.is.tsukuba.ac.jp, miuramo@iplab.is.tsukuba.ac.jp

and jiro@is.tsukuba.ac.jp

ABSTRACT

In this paper, we propose *GRIX* system, which is a visual programming system specified for interprocessor communications in parallel computing. We explain the system outline and its GUI based operations. The GUI based operations make users enable to input the structure of interprocessor communications with intuitive images. Code generation by this system omits the users' jobs of translation from the figured images of users into the textual program codes. The program codes generated by the system can be adopted for various parallel computing environments like LAN-connected workstations and massively parallel computers.

Keywords
Visual Programming System, Interprocessor Communications and Parallel Programs

INTRODUCTION

We have proposed GRIX system [1,2], which is a visual programming system specified for interprocessor communications appearing in programs for Message-Passing parallel computing. The aim of GRIX is to realize effective interprocessor communications and to reduce users' jobs by using visual input instead of complex input with textual specifications.

We must pay large attention for interprocessor communications when we construct parallel programs. The cost of interprocessor communications is the bottleneck of computing. Moreover, they bring barrier synchronizations in some cases. Inefficient coding brings frequent barrier synchronizations and the computing loss from the peak performance will be large.

There exists lots of means for coding the sentences of interprocessor communications, like *PVM* [3], *MPI* [4] and native functions implemented only for each computers. Those specifications show a tendency to be complex because we require high performance of execution. One of our emphasizing point is those complexities which bring confusion should be eliminated.

For instance, in order to realize the most simple communication, which is a pair of send and receive, by using PVM functions, programmers have to complete four procedures at least, packing data (ex. **pvm_pkint**), sending data (ex. **pvm_send**), receiving data (ex. **pvm_recv**) and unpacking data (ex. **pvm_upkint**). Moreover, some variables included in each function must be complex because of bringing high performance. This tendency must be strengthened at more native environments.

We worry about the above misfortune of programmers and insist on the following opinions.

- Forcing programmers to learn the meaning of difficult functions, especially unessential ones for interprocessor communications like memory mapping, is not desirable.
- The ideal method to materialize interprocessor communications is the programming with the most simple information, "who (which processing elements)", "where (which address space)", "what (which data)" and "how (like blocking and non-blocking)."

We have selected the realization by using Visual Programming method as the one of the most efficient solution for above insistence. Visual Programming method [10] is the programming form using the interface with visual images like icons, figures, animations and so on. Coding interprocessor communications intuitively with visual images and keeping high performance are the roots and the aim of GRIX.

VISUAL PROGRAMMING SYSTEM GRIX

GRIX is used in the case of appearing the interprocessor communications in SPMD programming. GRIX have been constructed the system from the following features.

- Input based on GUI (Graphical User Interface)
- Highly portable optimization technique
- GUI output of the optimization result
- Code generation corresponded to any parallel computing environments

It has become more intuitive for programmers to input communications because of GUI. The optimization which has high portability has made possible to generate the actual codes corresponded to any environments. The graphical output of the result of optimization enables users to realize the actual code intuitively and support in the program debugging.
GRIX system can be classified into three subsystems: GUI Engine, Optimization Engine and Code Generation Engine (Figure 1). GUI Engine changes the input information from the users into the file which can be executed in Optimization Engine. Optimization Engine changes the communications provided from the file into the one of the faster to communicate and easier to generate the actual code. The modified information is written into the two files, one is for the graphical output and the other is for code generation. GUI Engine shows the modified information graphically. Graphical output is for the purpose of intuitive realization of optimizations and can also help users during their debugging works. The file for code generation is used by Code Generation Engine in order to generate the actual code for each environment automatically. Code generation for each environment can be realized by preparing corresponding translator.

INTERFACE SPECIFIED FOR INTERPROCESSOR COMMUNICATIONS

We have paid attention to the viewpoint of the users, when they input the send-receive relationships. We explain the visual interface of GRIX with the most basic example assuming the input of 1-dimensional node ID (We call the physical processor represented on the GUI system "node").

Input from Absolute Viewpoint

As the one of the viewpoint, we have thought about the case that users already know the number of processors and they input the relationships in advance. For instance, it is the case that the programmer uses the single computer including multiple processors, like massively parallel computers (Figure 2), and he wants to use the actual ID of processors which have been assigned by each environment, like *"The processor **2** sends the data toward the processor **5**."* We call this viewpoint **"Absolute Viewpoint."**
The initial window (left of Figure 3) appears by informing the system of the number of showing nodes. In this window, both of the send-node and the receive-node are represented as circles. The nodes lined up vertically mean the set of processors, the two nodes aligned on the horizontal line represent the same processor: the left one is a send-node and the right one is a receive-node. The number in the left of each send node shows the ID of its processor, which is 0-origin. The user describes the relationship by drawing arrows. The relationships between nodes are shown as arrows. The most basic operation is dragging a mouse from the send-node to the receive-node. The user can describe any relationship among nodes naturally (right of Figure 3).
Though the user can describe every type of relationship by using this operation, he has to draw the arrows one by one. Therefore, we implement some multiple-arrow drawing operations which help the users input three typical communications like Broadcast, All-to-All Broadcast and Shift. For instance, in the case of the input of 1-to-multiple communications like Broadcast and Scatter, the user selects 1 send node and multiple receive nodes at first (upper left of Figure 4: selected nodes change into emphasized color). Then, by selecting existence order from the menu bar or typing established shortcut key (right of Figure 4), the system describes all arrows for selected nodes (lower left of Figure 4). Those advanced operations bring more rapid input by omitting monotonous repetition.

Input from Relative Viewpoint

As the contrast of Absolute Viewpoint, the user frequently images with the relative relationships among processors. This is often happened when the programmer uses the multiple workstations connected by LAN. For instance, it is one of the Scatter image like *"the processors whose ID are **even** number send the data to the processors which have the **–2,+1,+3 distance** from them."* In this viewpoint, users construct the communications with the notice of the movement in existence processors. We call this viewpoint **"Relative Viewpoint"** (Figure 5).
Explaining how to input this example, at first, user must inform the system of the number of nodes which is needed to represent all the relative processors (in this example, it is 6). The upper left of Figure 6 is the initial window

with 6 nodes. In this window user selects one node which is the center of viewpoint, we call the node **Owner**, and the system paints it with emphasized color and shows the distance in the left of nodes (upper right of Figure 6). If the user changes the owner, the value of distance shown in the left of the nodes is rewritten into the new value. Afterward, user describes the arrows with the same operations of Absolute Viewpoint (lower left of Figure 6). There remains the job of defining the condition of Owner. In this example, the condition of the active nodes is "ID is even number." Defining the condition starts from selecting **'Edit → define active nodes'** from the menu. The choices prepared as the default are **'all'** (all nodes are active), **'1 node'** (existence 1 node is active), **'even'** (the nodes which have even ID are active), **'odd'** (the nodes which have odd ID are active) and **'customize'** (active nodes are defined by the user). The input of this example is completed by selecting 'even' (lower left of Figure 6).
The user who wants to define another condition can select 'customize' and input the sentence of the definition in the dialog (this dialog also appears when user selects '1 node'). The system allows users to input the constant, the variable, the four basic operations of arithmetic and logical operations (ex. **((id /2) % 2) == 0** : **id** is the variable of own node ID). Definition of the condition can be executed not only after the input of communication relationships but also before that. Without the definition of the condition, the system judges the condition as "all active."
In the case of Gather, it is possible to input it by selecting Owner from the receive nodes. In the case of the input with this viewpoint, the system does not allow users to describe the arrow which is not connected with Owner. The advanced operations like the case of Absolute Viewpoint can be used while keeping that restriction.

OPTIMIZATION

GRIX optimizes the inputted communication relationships in order to improve the efficiency. The concept of optimization is to make the communication pattern which is **"fast to communicate"** and **"easy to generate"** the actual code. The principal jobs of Optimization Engine are as follows.

- Removing communication conflicts
- Reducing the number of communication by memory copying
- Generating conditional sentences of active (or inactive) nodes

Optimization for Absolute Input

GRIX optimizes the inputted communication relationships in order to improve the efficiency. Especially, in order to realize the motivation that is to keep up with any environment, the optimization for the communications which are inputted with Absolute Viewpoint is important because random communications can be happened.
When user has inputted with Absolute Viewpoint, the optimization is performed with the following 5 steps using synchronous communications.

(1). **Grouping the communications** by shift transfers which have same-stride bandwidth.
(2). **Assigning each group** of (1) to each *"step"* of synchronous communications.
(3). **Overlapping the steps**, if the steps have no conflict in send-nodes and receive-nodes.
(4). **Transforming the communication patterns**, if it can reuse the received data.
(5). **Creating the Bit-Pattern** which has the information of active nodes (or inactive nodes) in each shift transfer.

It is possible to generate the code only using (1) and (2). We can reduce the execution steps of communications by applying (3). By (1)~(3) implementations, the input example in Figure 3 is changed as shown in Figure 7.
In the center and the right of Figure 7, the nodes lined up vertically mean the set of processors, and the nodes aligned on the horizontal line represent the same processor. This is same to the input window. But their horizontal lines also have the meaning of the notion of time. Therefore, the leftmost nodes aligned vertically are initial send-processors. The next nodes are initial receive-processors. The receive-processors become the send-processors in the next stage. This optimization are performed with the synchronous communication. Therefore, there exists a barrier synchronization in the point that each vertical set of node are drawn. We call the gap as *"step."*
The optimization (4) which reuses received data is applied in case that there are the partial Broadcast and the other communication patterns. The example in Figure 8 is the case that 6 processors perform Broadcast and 4 processors, All-to-All Broadcast.
The number of the communication steps is computed as the largest number of the arrows which each node has. By adding the (4), the Broadcast is transformed like Figure 9 and the number of communication steps is reduced. We represent the number of processors which perform Broadcast as N_b. (1)~(3) requires N_{b-1} steps because of the send-conflict of Broadcast. Then optimization added (4) reduces the communication steps into $log_2 N_b$ (N_b is n's power of 2) or $log_2 N_b + 1$ (N_b is not n's power of 2). In case of Figure 8, the optimization reduces the number of steps from 5 into 3.
Moreover, in the example of Figure 8, there are two irregular nodes, node 4 and 5, for the Broadcast transformation at the Step 3. In Step 3, each node of 0→3 can send the data to node 4 and 5. In this case, the stride must be +3 because there exists the communication by +3, node 6 to 9. So the stride computed by the Broadcast transformation at Step 3 is +3.
The purpose of (5) is to write the code of deciding the

active (or inactive) nodes by one conditional sentence. For example, if send-active nodes are *{0,3,8}* in *{0~9}*, its Bit-Pattern is $d_bit=2^0+2^3+2^8=1+8+256=265$. And its code is as follows.

```
d_bit = 265;
if (1 && (d_bit >>= myid))
        /*myid:node ID of myself*/
```

And the code of deciding the receive-active nodes includes the sentences which decide senders and use the *d_bit*. The code is as follows.

```
nprocs = 10;
if ((sender = (myid-stride)) < 0)
   sender += nprocs;
else if (sender >= nprocs)
   sender -= nprocs;
if (1 && (d_bit >>= sender))
      /*nprocs:the number of processors*/
      /*stride:the distance of shift transfer*/
```

In case of one step has very few inactive nodes and has no overlapped step, it is possible to change the step for All-Active. This implementation can take away the conditional sentence. If only unique communications like Broadcast and Summation are requested, it does not implement above optimization and uses the functions which each environment has.

Optimization for Relative Input

The optimization for Relative Input is simpler than the one of Absolute Input. The system does not need to prepare the communication steps and overlap them. Overlapping is impossible because there always exists the communication conflict in Owner node. Asynchronous communications can be executed by labeling on each stride communications. Of course, the system has to prepare each receiving memory space.
Now, we use **myid** as own ID and **nprocs** as the number of processors in the actual code. In the case of Figure 6, the distance from active nodes are -2, +1 and +3. If those strides stored in the array **stride[]** like **int stride[3] = "-2, 1, 3";**, the send and receive sentence forms in the actual code are like following code.

Send Sentence:

```
if ((myid % 2) == 0)+¥¥
    for (i = 0; i < 3; i++)-¥¥
        SEND to (myid + stride[i]), label(i);
/* Sending the data labeled i toward the node which has
the distance of stride[i] */
```

Receive Sentence:

```
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);
/* Receiving the data labeled i from Sender, the node
which has the -stride[i] distance, if Sender is active */
```

In the case that the input contains *all processors*, like *"The processors whose ID are odd number receive the data from all processors"*, the send and receive sentence forms are like following code using **nprocs** instead of **stride[]**.

Send Sentence:

```
for (i = 1; i < nprocs; i++){
    if ((receiver = myid + i) < 0)
        receiver += nprocs;
    else if (receiver >= nprocs)
        receiver -= nprocs;
    if ((receiver % 2) == 1)
        SEND to receiver, label(i);
/* Sending the data labeled i toward Receiver, the node
which has the i distance, if Receiver is active */
```

Receive Sentence:

```
if ((myid % 2) == 1)
    for (i = 1; i < nprocs; i++)
        RECV from (myid - i), label(i);
/* Receiving the data labeled i from the node which has
the distance of -i */
```

In order to determine the sentence forms, Optimizer introduces following 4 information into Code Generator.

- Which is Owner, send node or receive node?
  → Definition of the conditional sentence form
- How many nodes does Owner communicate to or communicate from?
  → Definition of the number of the element of **stride[]**, or using **nprocs**
- How long distance between send node (or send nodes) and receive nodes (or receive node)?
  → Definition of each value of **stride[]**
- What condition does Owner have?

FEEDBACK FOR USER

After the optimizations, user knows the result by the graphical output windows as shown in Figures 10 and 11. These graphical output windows bring users more intuitive realization about how each node communicates in each step. Figure 10 is the result of the optimization for Figure 3 which is the input with Absolute Viewpoint.
In the case of the output for Relative Input, the system does not know the number of the actual processors and asynchronous communications can be happened. Moreover, the result of optimization is mainly the information of active nodes. So, the system describes the output window with assuming existence number of processors. Figure 11 is the output window of Figure 6 assuming 10 processors.
Though this output can be seem the one with synchronous communications, all the communications can be executed

asynchronously. Users can be realized the information that *"the nodes whose ID are even number are active"* intuitively by this output window. These graphical outputs can help users when they debug the program.

CODE GENERATION

GRIX involves the subsystem which generates the actual code automatically in order to deal with the recent various environments, like PVM and MPI. This code generation removes coding with the knowledge of those specifications and enables the programmer to write the code with the intuitive manner. Moreover it is possible to acquire the code for various environments with their corresponding translation engines. We have prepared the code generation subsystem for PVM at first. For instance, the actual code of Figure 6 which is the input with Relative Viewpoint is following code.

```
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++;
    }
}
```

The followings are simple explanations of variable names written in the program.

- **SPMD_nprocs**: the number of Virtual Processors (VP)
- **SPMD_procnum**: the VP number of myself (zero origin)
- **SPMD_tids**: the array of tid those indexes are VP numbers

RELATED WORKS

GRIX is a new GUI tool for parallel computing because we have implemented it specified for the input of interprocessor communications.

There exists some GUI systems for parallel computing. P. Newton and J. C. Browne propose *CODE* [5], which is a visual parallel programming language. G. A. Geist proposes *HeNCE* [7], it is also the visual parallel programming language. Newton has compared their characteristics in the bibliography [6]. Those visual programming languages force users to write the program on their GUI windows. The visual input for the sentences which is not redundant, like the normal sentence of basic operations *C[i,j] = C[i,j] + A[i,k] * B[k,j];* , has some waists. We insist that the visual input for the redundant points is most effective and the redundant point in the parallel program is the sentence for interprocessor communications.

XPVM [8] is the GUI system in order to control and visualize the PVM programs. GRIX and XPVM have same characteristics on the point of controlling interprocessor communications. But GRIX is not aiming only on PVM but also the multiple plat home for executing interprocessor communications. Performance monitors which visualize the load performance of each processor, are also the GUI systems for parallel computing. *ParaGraph* [9] is the one of performance monitors. Though performance monitors are only for output, GRIX has the functions of both input and output.

CONCLUSIONS

There exists lots of specifications to code interprocessor communications and they shows a tendency to be complex. Visual programming method has large possibility to make such a difficult programming more smooth. We have proposed the GRIX system, the visual programming system for making the programming of interprocessor communications less difficulty. Because of the GUI system, GRIX is the efficient environment to realize the faster interprocessor communications and allows programmers the easier and more intuitive inputs of interprocessor communications. GRIX can set programmers free from complicated works being familiar with a lot of complex specifications about interprocessor communications. GRIX will be more effective and more interactive system by jointing the existing textual editors.

REFERENCES

[1] Yasutaka Sakayori, Motoki Miura and Jiro Tanaka: GRIX: *Visual Programming System for Interprocessor Communications*, Proceedings of the 10th IASTED International Conference PDCS '98, pp.503-508, Oct. 1998

[2] Yasutaka Sakayori: *GRIX: The Proposal of Visual Programming System Specified for Interprocessor*

Communications, Master Thesis, University of Tsukuba, Feb. 1999 (in Japanese, English shorter version is available)

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

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

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

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

[7] 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

[8] 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

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

[10] B. A. Mayers: *Taxonomies of Visual Programming and Programming Visualization*, Journal of Visual Language and Computing, 1(1):97-123, 1990

FIGURES

Figure 1: System Structure of GRIX

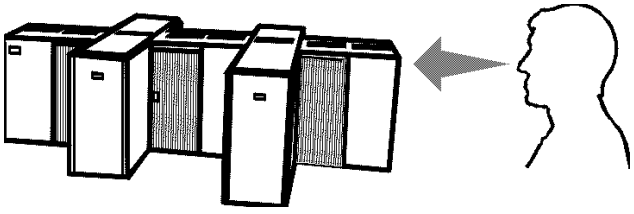

Figure 2: Absolute Viewpint

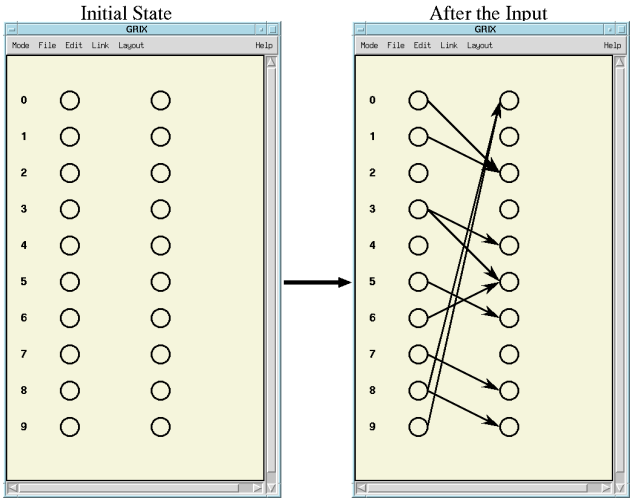

Figure 3: Input from Absolute Viewpoint

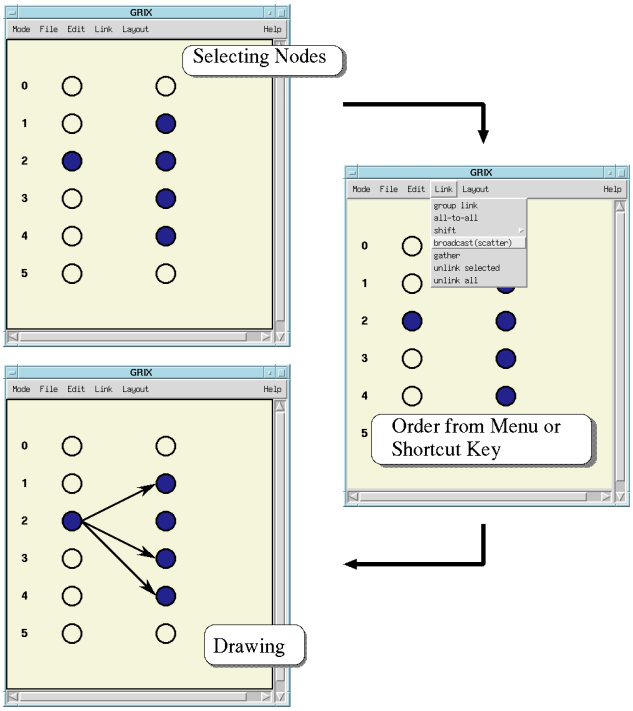

Figure 4: Advanced Operation

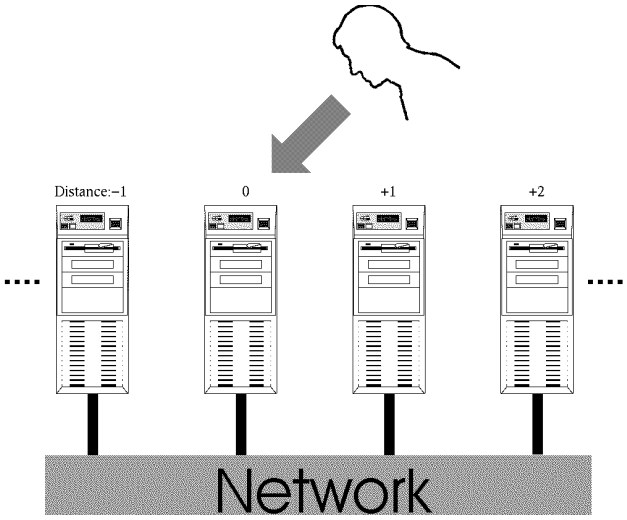

Figure 5: Relative Viewpoint

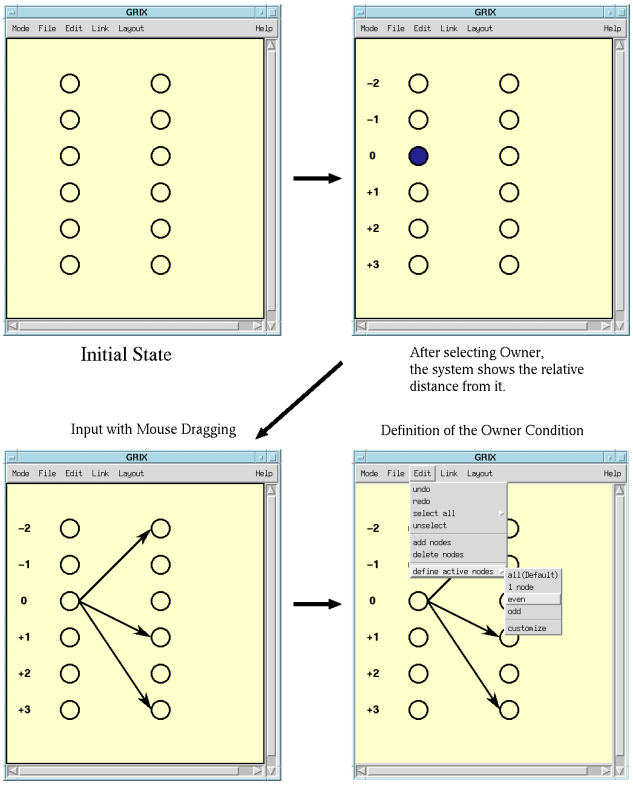

Figure 6: Input from Relative Viewpoint

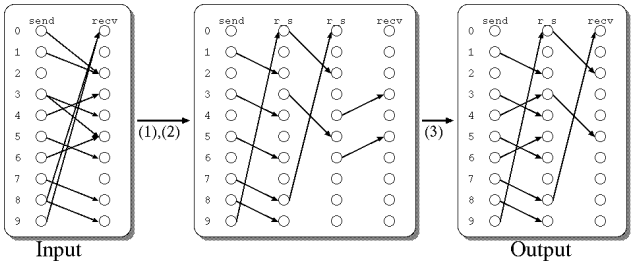

Figure 7: Optimization

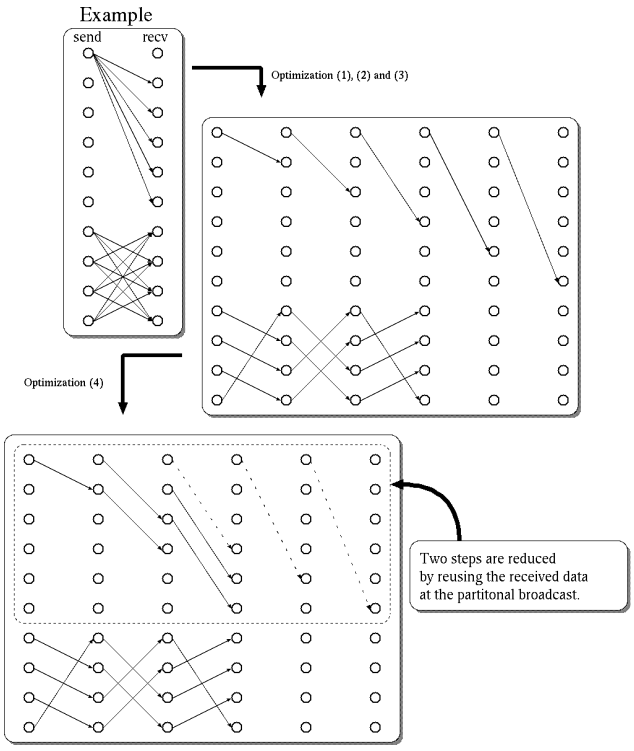

Figure 8: Reusing Received Data

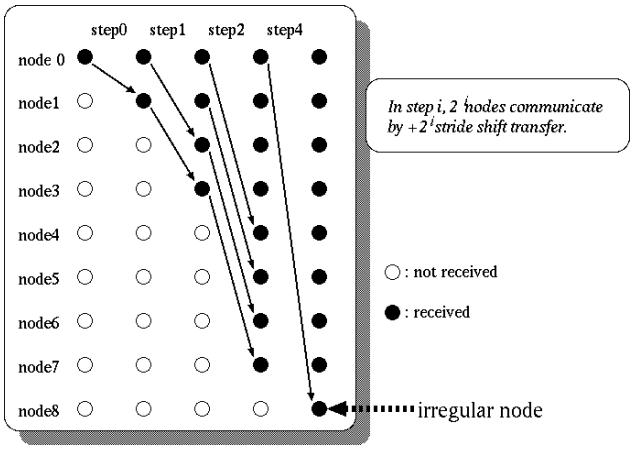

Figure 9: Broadcast Transformation

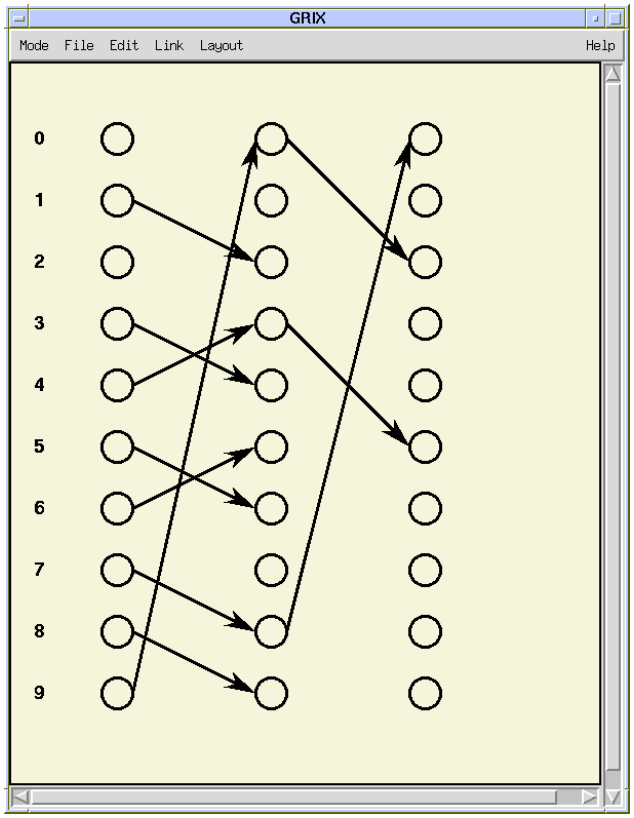

Figure 10: Graphical Output with Absolute Viewpoint

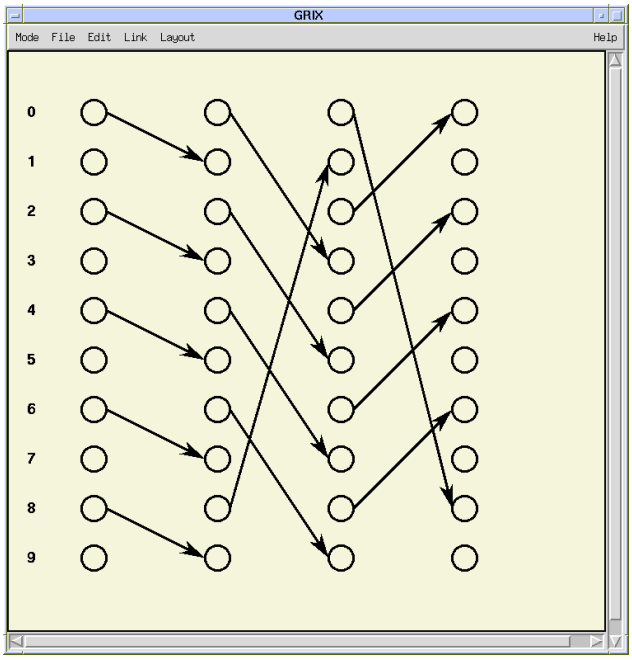

Figure 11: Graphical Output with Relative Viewpoint