Subgraph Identification In A Parallel Programming Environment*

Qing Yu

COINS Technical Report 89-30
July 1989

Department of Computer and Information Science

University of Massachusetts

Amherst, Massachusetts 01003

USA

* The Parallel Programming Environments Project at the University of Massachusetts is supported by the Office of Naval Research under contract N000014-84-K-0647 and by the National Science Foundation under grants DCR-8500332 and CCR-8712410.

Abstract

The specification of interprocess communication for massively parallel computation requires the support of sophisticated programming tools. As part of a parallel programming environment for MIMD, non-shared memory message-passing architectures, we are building a graph editor that supports the use of process structure graphs in programming and a graphical interface that supports the use of a new programming abstraction for interprocess communication. Both tools will require the automatic identification of subgraphs. We report here on heuristics that perform such identification.

1 Introduction

Massively parallel programming is much more difficult than sequential programming, and it will require the development of sophisticated programming tools. For MIMD non-shared memory message-passing architectures, these tools must support the specification of interprocess communication structures which is both tedious and error-prone.

The availability of low-cost workstations with a bit-mapped display and mouse has made the widespread use of graphics software possible. Visual display of communication structures has been adopted by several programming environments[1,2,3]. These environments allow the user to draw a graph. But because it is tedious to draw a large graph manually, and because parallel programs should be scalable, specifications of entire *graph families* are required. We are currently building a grammer-based graph editor[5,6,7], which allows the user to generate a large graph by applying transformations to a small graph in the same family. As part of this editor, we must identify subgraphs that form the domain of a transformation and we must identify subgraphs called *itineraries* that provide a path for abstract messages. This subgraph identification is the focus of this project.

In the next two sections, we describe the use of subgraph identification in specifying transformation domains and in specifying itineraries. We discuss possible extensions in the last section.

2 Subgraph Characterization and Recognition in the Graph Editor

The program we developed solves the problem of subgraph characterization and recognization. We will first explain its role in the graph editor, and then discuss some issues in its implementation.

2.1 Requirements from the Graph Editor

Nodes in a communication graph represent processes, and edges represent their intercommunications. Both nodes and edges can be labeled with a set of attributes. Attributes can be roughly classified as follows:

system-defined: Attributes in this class can be display related or graph-theoretic. Display related attributes — such as position coordinates — provide information about the display of the graph on the screen. Graph-theoretic attributes — such as the degree of a node — provide information on the structural characteristics of the graph.

user-defined: For a specific graph family, more attributes can be introduced by the user. Most of these are structure related. For a mesh, the row and column numbers are two obvious choices. For a tree, a node is often labeled with its level number. User-defined attributes can also be code related. They can be used for process labeling (assigning code to a process) or as parameters in a generic code body. There is a close relationship between structure related attributes and code related attributes. The same code is usually assigned to the nodes with the same structure features. For example, usually the same code is assigned to the leaf nodes in a tree which are labeled with the maximum level number. Code related attributes are essential for the processes to work correctly. Structure related attributes usually carry more specific information about a node. Their presence helps to characterize sets of related nodes.

The graph editor provides support for specifying labeled graph families based on *aggregate rewriting graph grammars* [5,6]. To describe a graph family, the user first constructs a *start graph* which is often the smallest instance of the family and then defines the transformations needed to modify that start graph into the next larger family member. The transformations are then automatically iterated to form all remaining family members.

In specifying a transformation, the user must be able to define its domain — that is, the set of nodes to which it should be applied. He can do this directly, by providing a predicate describing the domain, or he can do it by example: he selects a set of nodes from a sample graph and have the system automatically generates a closed-form expression that describes the selected subset. The predicates are over the set of user- and system-defined graph attributes. This problem is called *subgraph characterization.*

When a transformation is applied to a larger graph, the domain of the transformation must be identified. With the predicate describing this domain, the editor must be able to identify it in this larger graph. We refer to this problem as *subgraph recognization.*

We have developed an subgraph characterization and recognition package to solve these problems. It is described in the next subsection.

2.2 Subgraph Package

There are several issues involved in the implementation of the subgraph package. First, the format of predicates must be determined and the algorithm for predicate deduction must be designed. Second, the predicate deduced from a specific (usually small) graph must be generalized to be applied to larger graphs in the family. Third, if there are multiple results of the characterization, they should be ranked according to generality and simplicity.

The format of predicates is defined as follows:

- A *base predicate* is a positive predicate of the form $name = value$ or a negative predicate of the form $name \neq value$, where *name* is an attribute name and *value* is a value in the domain of that attribute.
- A *predicate* is a base predicate or a conjunctive or disjunctive composition of two base predicates.

Our experience indicates that this simple format is sufficient for most of the useful subgraphs.

All the predicates applicable to a subgraph are generated. In order to avoid duplication, we order the attributes and then order the two base predicates in a composite predicate by their attributes. The algorithm has 4 phases:

1. Get all the positive base predicates for the subgraph.
2. Get all the predicates which are compositions of two positive base predicates.
3. Get all the predicates which are compositions of two base predicates where the first is positive and the second is negative.
4. Complement the subgraph; repeat the three phases above; negate the predicates; and transform them into standard format.

The last phase reduces the complexity of the algorithm and increases the number of applicable predicates generated.

Example 2.1 Let's use the mesh shown in Figure 1 to illustrate how predicates are generated. The nodes in the mesh are labeled with two user-defined attributes *row* and *column*.

- For subgraph {(0,0), (0,1), (0,2)}, the predicate ($row = 0$) is generated in phase 1.
- For subgraph {(0,0), (0,1), (0,2), (1,2), (2,2)}, the predicate ($row = 0$ *or* $column = 2$) is generated in phase 2.
- For subgraph {(0,0), (0,1)}, the predicate ($row = 0$ *and* $column \neq 2$) is generated in phase 3.
- For subgraph {(0,0), (0,1), (1,0), (1,1)}, nothing is generated in phase 1–3. In phase 4 the predicate ($row = 2$ *or* $column = 2$) is generated for the complement of the subgraph. Then the predicate is negated, and we have ($row \neq 2$ *and* $column \neq 2$) which describes the original subgraph.

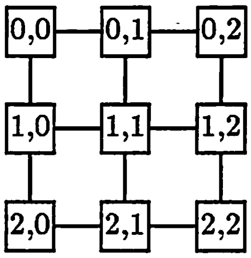


Figure 1: an annotated 3×3 mesh

After we get the list of predicates, we have to generalize them so that they can be applied to other graphs in the same family. We need to replace specific values with general values like MAX or MIN where MAX (MIN) is the maximum (minimum) value for a given attribute in the graph. We use $\max(attname, graphname)$ and $\min(attname, graphname)$ to represent the maximum and minimum values of the attribute *attname* in the graph *graphname*. The generalization process is as follows for a graph named *graphname*:

- Transform base predicates: Suppose the predicate has a base predicate $attname = value$ or $attname \neq value$.
  Replace *value* by MAX if $value = \max(attname, graphname)$.
  Replace *value* by MIN if $value = \min(attname, graphname)$.

- Coordinate two base predicates in a predicate: Suppose the predicate has two base predicates and both have *attname* as the attribute portion.
  If after the first step one value portion is MAX, another is *val*, and
  $\max(attname, graphname) - val \leq val - \min(attname, graphname)$,
  replace *val* by $MAX - (\max(attname, graphname) - val)$.
  If after the first step one value portion is MIN, another is *val*, and
  $val - \min(attname, graphname) \leq \max(attname, graphname) - val$,
  replace *val* by $MIN + (val - \min(attname, graphname))$.

Example 2.2 Here we generalize the predicates given in Example 2.1. Let's call the 3×3 mesh shown in Figure 1 $mesh_3$. $\min(row, mesh_3) = 0$, $\max(row, mesh_3) = 2$, $\min(column, mesh_3) = 0$, $\min(column, mesh_3) = 2$. The four predicates are generalized as follows:

- $row = 0 \Rightarrow row = MIN$
- $row = 0$ *or* $column = 2 \Rightarrow row = MIN$ *or* $column = MAX$
- $row = 0$ *and* $column \neq 2 \Rightarrow row = MIN$ *and* $column \neq MAX$
- $row \neq 2$ *and* $column \neq 2 \Rightarrow row \neq MAX$ *and* $column \neq MAX$

There are often several predicates applicable to one subgraph. The predicates are ranked based on the simplicity first and then the generality. A predicate with one attribute is simpler than a predicate with two attributes. A base predicate is simpler than a composition of two base predicates. If two predicates have the same number of base predicates, the one with less negations is the simpler. The generality increases whenever a specific attribute value is replaced by a general value. The predicates are presented to the user in the order of the rank. Our experience shows that this ranking works well.

Example 2.3 A binary tree is shown in Figure 2. Each node is labeled with an attribute *level*. The root is on level 0, and the leaf nodes are on level 2. A node also has an attribute *degree* whose value is automatically derived by the system. If the nodes on level 1 are selected, six predicates will be presented for this subgraph in the following order:

1. $degree = MAX$
2. $level = 1$
3. $level \neq MAX$ *and* $level \neq MIN$
4. $degree \neq MIN$ *and* $degree \neq MIN + 1$

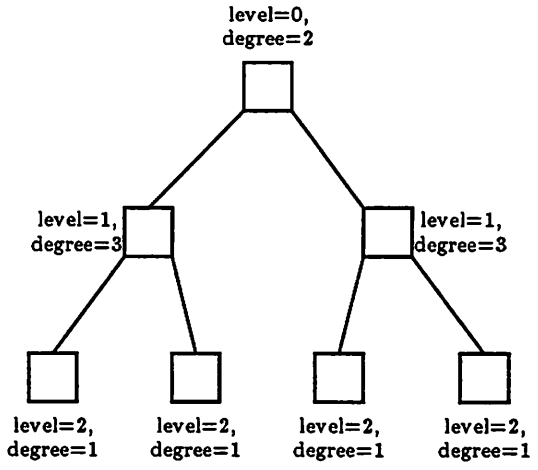


Figure 2: a 3 level binary tree

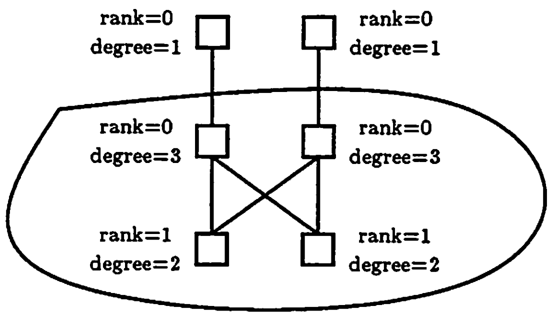
5. $level \neq MIN$ *and* $degree \neq MIN$

6. $level \neq MAX$ *and* $degree \neq 2$

Predicate 1 is simplest and most general. Predicate 2 is simple but less general. All the other predicates are compositions of two base predicates and are more complex than 1 and 2. Predicates 3 and 4 are ranked the same, their order depends on the order of the attributes. If *level* is in front of *degree*, the predicates in terms of *level* will be generated first. The predicates are sorted by a stable sorting algorithm, so predicate 3 remains to be ahead of predicate 4. Predicates 4 and 5 are both general, but predicate 4 has only one attribute (*degree*) involved while predicate 5 has two attributes ($level, degree$) involved. Predicate 6 also has two attributes involved and is less general than predicate 5, so it is the last in the list.

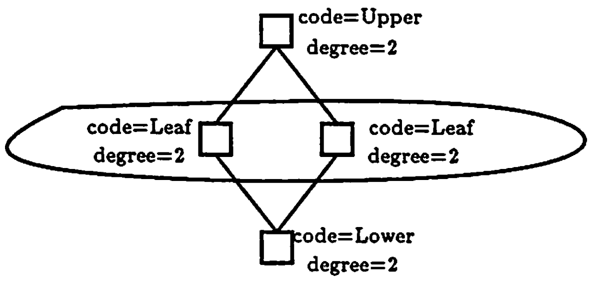
Next we give two examples to illustrate how subgraph characterization is used in defining the domain of a transformation.

Example 2.4 To describe the family of butterfly graphs, we begin with a four node graph annotated by a user-defined attribute *rank* and system-defined attribute *degree*. Three transformations are needed to generate the next butterfly: the first begins a new rank by making a copy of the nodes along the top rank; the second copies the original butterfly, and the third completes the graph by adding a copy of the top rank [7]. The second transformation must be applied only to nodes of the original butterfly; the user indicates this by circling the desired domain in the first instance of the graph:



Our heuristics then generate the description $degree \neq MIN$. This predicate can be used to identify the domain of the transformation when it is later applied to other graphs.

Example 2.5 Another example is given in [7] where domain restrictions are used. A tree resulted from the combination of two binary trees at the leaves is called a *database tree.* To specify the family of database trees we start with a four node graph and a transformation is applied on the leaf nodes. The user can simply circle the leaf nodes as shown in the graph. The description of the domain of this transformation is $code = Leaf$.



The problem of subgraph recognition is straightforward once the internal representation for predicates has been determined. Given a graph and a predicate, each node in the graph is tested. The subgraph consists of the nodes satisfying the predicate.

3 A Graphical Interface for Canister Communication

In addition to process graph specifications, the graph editor supports the specification of global communication patterns. In this section, we discuss the role of subgraph identification in a programming abstraction, called *canister communication* [8]. We first introduce some basic concepts about canister communication and then discuss the implementation of a graphical interface.

3.1 Canister Communication

Massively parallel algorithms usually have regular communication patterns. Canister communication supports the recognition of these patterns. A *canister* is a logical container of a message which traverses a directed subgraph called an *itinerary.* On the itinerary nodes represent processes and edges represent communication channels. The type of data a canister can carry is declared and each node on the itinerary has a set of access rights to that data. Message passing primitives are replaced by logical operations on canisters which are used in a manner consistent with the itinerary. A canister is created in association with an itinerary. It is filled with data and transmitted along the itinerary. Eventually it is emptied and destroyed. Detailed definitions of itineraries and canister operations are given in [8]. Here we will present some examples of meaningful itineraries which are supported by the graphic interface to be discussed in the next subsection.

Example 3.1(a) In the band `matrix multiplication` algorithm[10], data from the two input matrices A and B enter a processor along two paths. The path for matrix A is from left to right. The path for matrix B is from top to bottom. When the data along path A and path B are available, a process reads the a and b values and computes $c = c + a \times b$. The c value moves along path C, that is, diagonally from bottom right to top left. There are three itineraries corresponding to these three paths. A node can only have read access to data in canisters traversing on itineraries A and B. It has read and write access to data in canisters traversing on itinerary C.

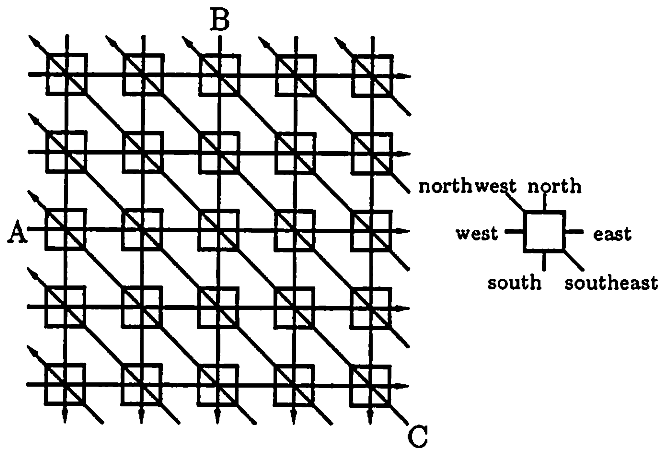


Figure 3: communication paths in band `matrix multiplication`

In this example an itinerary is simply a subgraph consisting of straight paths which do not intersect. The next example we will present has a more complex itinerary where paths are merged at some points.

Example 3.2(a) A pipelined `tree summation` algorithm can be used to perform parallel summation of vectors. Each process gets the values from the child processes, performs the summation, and passes the value to the parent process. All the communications support the same logical operation. The resultant itinerary is the whole tree. Let's call this itinerary `TreePath` and the canister `SumCan`. Code for the processes can be written as follows[4]:

```
int sumfun(a, b) { return a+b; }

tree() {
    CanTypeDecl(int, SumCan) ;

    sumfun(CanGet(TreePath),SumCan) ;
    CanPut(SumCan) ;
}
```

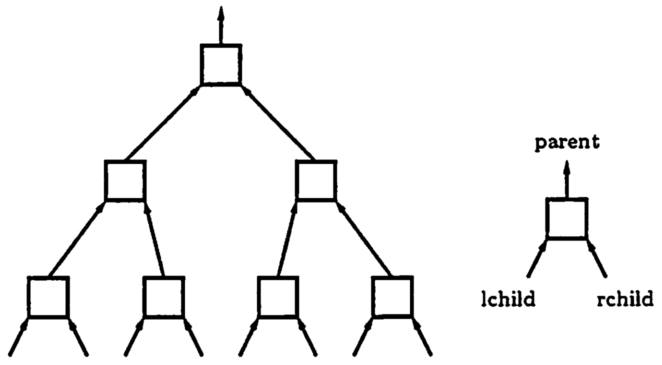


Figure 4: communication paths in `tree summation`

Fan-in results when two or more paths join at a node; we propose two functions for merging values at such nodes: `CanSelectAny` which returns message chosen at random from those channels with messages available; and `CanSelectAll` which returns a set of messages, one from each incoming channel. `sumfun` is a user-supplied merge function which sums the messages in such a set and returns the value in its second parameter.

In the `tree summation` algorithm, both inbound channels of a process are involved in the same logical operation. They are logically undistinguishable. In the next example processes appear in different places on the itinerary. The inbound channels associated with a process are involved in different logical operations and should be distinguished.

Example 3.3(a) In a `LU-decomposition` algorithm implemented on a mesh [11,12], data needs to be routed diagonally through a vertical and a horizontal channel as shown in Figure 5. There are three processes on a path. A process can be in three different positions on three different paths. *Aliases* are introduced to distinguish the positions of the process on the itinerary. The alias `Shift` is used for the first and the third positions, `Pass` is used for the second position. `CanGet( Pass)` then returns the canister sent by the neighbor below, and `CanGet(Shift)` returns the canister sent by the neighbor to the right. The two outbound channels are distinguished by aliases `Pass` and `Shift` in a similar way.

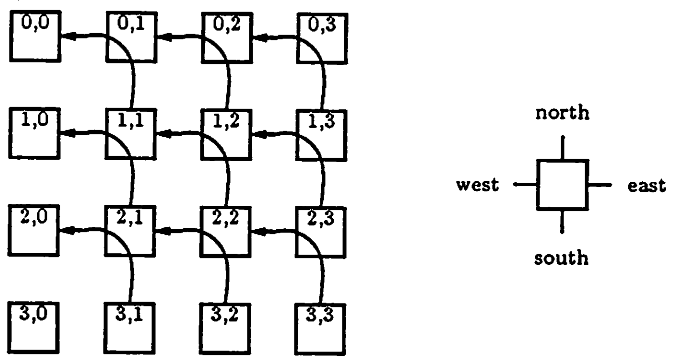


Figure 5: communication paths in `LU-decomposition`

3.2 Implementation of the Graphic Interface

Itineraries can be specified textually by a programmer, but it will be easier for him to use a graphic interface. We will describe the work we have done on such an interface in this subsection. Currently, the itineraries that the interface supports can not be cyclic. Fan-in or fan-out in the itineraries is supported separately. It is not allowed that both fan-in and fan-out exist in the same itinerary.

Again, the user specifies his itinerary on a small graph and our system generalizes the description to all graphs within the family. The user specifies a set of start nodes for the paths on his itinerary and gives a sample list of nodes for a complete path within a small graph. The system will generate an expression describing the itinerary which can be applied to the set of start nodes to identify the complete itinerary. During this process the user may be prompted for aliases when needed. The relevant information about the itinerary will be stored so a program using canister operations can be translated into one using standard message-passing mechanisms by the compiler.

Our approach is based on the use of local names for communication channels which may be thought of as *ports*. The graph editor supports the labeling of ports,

Given a path on an itinerary, the system will generate an expression for the names of the exit ports on the path. Two types of expressions are currently used

- *simple path expression:* $p_1, p_2, ...p_n$
- *transitive path expression:* $(p_1, p_2, ...p_n)^*$

where p_i is the name of an exit port. To identify the path expression, the list of exit ports is first extracted from the path. The result can be considered as a simple path expression. This expression will be contracted into a transitive path expression if the list is a nontrivial repetition of a sublist. For example, suppose a list of exit ports *east*, *east*, *east*, *east* is extracted from a path. This list is a repetition of a sublist *east*, and is contracted into a transitive path expression $(east)^*$. To be more precise, we define $(p_1, p_2, ...p_n)^k$ $(k > 1)$ to be the list gained by repeating $p_1, p_2, ...p_n$ k times, and call $p_1, p_2, ...p_n$ a *base list*. The result of the contraction of a simple path expression is $(p_1, p_2, ...p_n)^*$ where $p_1, p_2, ...p_n$ is the shortest base list.

Once the path expression is identified, it will be used with the set of start nodes to identify the itinerary. For each start node we identify a path by following the exit ports in the path expression and add the path into the itinerary. If the path expression is transitive, the base list is used repeatedly until it is not applicable.

Example 3.1(b) Consider the band `matrix multiplication` algorithm. A node in Figure 3 has six ports: `north`, `south`, `east`, `west`, `northwest`, and `southeast`. There are three itineraries in this communication graph. For itinerary A, the user selects the first column to be the set of start nodes. He can specify one of the five rows to be a sample path on the itinerary. The list of exit ports *east*, *east*, *east*, *east* is extracted from the path, and it is transformed into a transitive path expression $(east)^*$. If we start from a node in the first column and follow the exit port *east* repeatedly, a complete row can be identified and added to the itinerary. The resultant itinerary is a directed subgraph consisting of all the nodes and all the horizontal edges. For itinerary C, a node on the last row or column is a start node, and a diagonal path is a sample path on the itinerary. The path expression is $(northwest)^*$. With this path expression, all the diagonal paths can be identified even though they have different lengths.

After the itinerary is identified the nodes will be classified according to its list of ports. The nodes in the same class have the same exit ports and entry ports. Associated with a class is a predicate describing it. The function in the subgraph package is used to identify the predicate for a class. The information is needed to translate a program using canister operations into one using standard message-passing mechanisms. For itinerary A in the example above, the following information is stored:

```
class1:  predicate:     column = MIN
         exit ports:    east
         entry ports:   west(dangling)
class2:  predicate:     column ≠ MIN and column ≠ MAX
         exit ports:    east
         entry ports:   lchild, rchild
class3:  predicate:     column = MAX
         exit ports:    east(dangling)
         entry ports:   west
```

In the example above the itinerary consists of straight lines which do not intersect. When they do intersect, the user is prompted for aliases. Aliases are needed to distinguish the different local roles a node plays. When aliases are used a port is virtually a pair of a port name and an alias. The user does not have to provide aliases if the node supports the same logical operation on different paths.

Example 3.2(b) To specify the itinerary for the `tree summation algorithm` shown in Figure 4, the user selects the leaf nodes to be the start nodes and specify a path from a leaf to the root. Each node has three ports: `lchild`, `rchild`, and `parent`. A sample path is described by the transitive path expression $(parent)^*$. A node may be on several paths from a leaf to the root, but it supports the same logical operation "summation". No aliases are needed.

The information stored by the system is:

```
class1:  predicate:   level=MIN (root)
         exit ports:  parent(dangling)
         entry ports: lchild, rchild
class2:  predicate:   degree=MAX (interior)
         exit ports:  parent
         entry ports: lchild, rchild
class3:  predicate:   level=MAX (leaf)
         exit ports:  parent
         entry ports: lchild(dangling), rchild(dangling)
```

For a interior process, a `CanGet` operation is translated into getting data from ports `lchild` and `rchild`, performing the computation specified by the merge function. A `CanPut` operation is translated into sending data through port `parent`. The leaf and root nodes have dangling ports. If the external I/O interface supports canister communication, the canister operations can be translated into I/O operations.

Example 3.3(b) Consider the itinerary for the `LU-decomposition` algorithm shown in Figure 5. Each node has four ports: `north`, `south`, `east`, and `west`. The predicate describing the set of start nodes is ($row \neq MIN$ *and* $column \neq MIN$). If we use (i, j) to denote the node on row i and column j as shown, a sample path on the itinerary is (i, j), $(i-1, j)$, $(i-1, j-1)$. The path expression is *north*, *west*. A node can be at different points on the itinerary. For example, node $(2,2)$ is the first on path $(2,2)$, $(1,2)$, $(1,1)$, the second on path $(3,2)$, $(2,2)$, $(2,1)$, and the third on path $(3,3)$, $(2,3)$, $(2,2)$. When a node is encountered twice the user is requested to provide aliases. In this case he can provide a list of aliases ($Shift$, $Pass$, $Shift$) to distinguish the three positions. The usage of these aliases in a code body has been discussed in Example 3.3(a).

The nodes on the itinerary are divided into several classes based on the ports and aliases. Here we give the description of one class(the interior nodes in the mesh) to demonstrate the information the system stores.

```
predicate:    degree=MAX
exit ports:   (north, Shift), (west, Pass)
entry ports:  (south, Pass), (east, Shift)
```

For a node in this class, a CanGet operation is translated into a read on port south if the alias is Pass, or a read on port east if the alias is Shift. CanPut is translated similarly.

4 Future Work

In some communication graphs such as n-cubes and butterfly networks, the patterns of the connections are described in terms of the bits of the id numbers of the nodes. The format of the predicates used in the subgraph package is being extended so that bit patterns can be described. We are also considering using display related attributes such as position coordinates to characterize subgraphs. New requirements may appear as the graph editor becomes sophisticated and new functions have to be provided by the subgraph package correspondingly.

The types of itineraries supported by the graphic interface is currently limited. Basically it supports the specification of simple communication paths with no cycle and broadcast. We are considering extensions to support more complex itineraries.

References

[1] Lawrence Snyder, "Parallel Programming and the Poker Programming Environment," *Computer* 17(7), pp. 27-37 (1984).

[2] James Purtilo, Daniel A. Reed and Dirk C. Grunwald, "Environments for Prototyping Parallel Algorithms," *Proceedings of the 1987 International Conference on Parallel*

Processing, pp. 431-438 (August 1987).

[3] Kathleen M. Nichols and John T. Edmark, "Modeling Multicomputer Systems with PARET," *Computer* 21(5), pp. 39-48 (May 1988).

[4] Duane A. Bailey, Specifying Communication for Massively Parallel Ensemble Machines, Ph.D. Thesis, COINS Department, University of Massachusetts (1988).

[5] Duane A. Bailey and Janice E. Cuny, "Graph Grammar Based Specification of Interconnection Structures for Massively Parallel Computation," *Graph Grammars and Their Application to Computer Science, Lecture Notes on Computer Science* 291, pp. 73-85 (1987).

[6] Duane A. Bailey and Janice E. Cuny, "An Approach to Programming Process Interconnection Structures: Aggregate Rewriting Graph Grammars," *Parallel Architectures and Languages Europe, Lecture Notes in Computer Science 259,* J.W. de Bakker, A.J. Nijman and P.C. Treleaven (eds.), Springer-Verlag, pp.112-123 (June 1987).

[7] Duane A. Bailey and Janice E. Cuny, "ParaGraph: Graph Editor Support in Parallel Programming Environments," COINS Technical Report 89-53

[8] Duane A. Bailey and Janice E. Cuny, "Canister Communication in Parallel Programs," COINS Technical Report 88-42 (October 1988).

[9] Alfred A. Hough and Janice E. Cuny, "Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation," *Proceedings of the 1987 International Conference on Parallel Processing,* pp. 735-738 (1987).

[10] H.T.Kung *Let's Design Algorithms for VLSI Systems* CalTech Conference on VLSI, pp.65-90 (1979)

[11] Michael J.Quinn *Designing Efficient Algorithms for Parallel Computers.* McGrawHill, New York, 1987.

[12] Sun-Yuan Kung, K.S. Arun, Ron J. Gal-Ezer, and Bhaskar Rao. Wavefront Array Processor: Language, Architecture, and Applications. *IEEE Transactions on Computers*, C-31(11):1054-1066, November 1982.