SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2005; **35**:1409–1439
Published online 29 July 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.676

Visual programming support for graph-oriented parallel/distributed processing

Fan Chan[1], Jiannong Cao[1,*,†], Alvin T. S. Chan[1] and Kang Zhang[2]

[1]*Software Management and Development Laboratory, Department of Computing, Hong Kong Polytechnic University, Hung Hom, KLW, Hong Kong*
[2]*Department of Computer Science, P.O. Box 830688, EC31, The University of Texas at Dallas, Richardson, TX 75083-0688, U.S.A.*

SUMMARY

GOP is a graph-oriented programming model which aims at providing high-level abstractions for configuring and programming cooperative parallel processes. With GOP, the programmer can configure the logical structure of a parallel/distributed program by constructing a logical graph to represent the communication and synchronization between the local programs in a distributed processing environment. This paper describes a visual programming environment, called *VisualGOP*, for the design, coding, and execution of GOP programs. VisualGOP applies visual techniques to provide the programmer with automated and intelligent assistance throughout the program design and construction process. It provides a graphical interface with support for interactive graph drawing and editing, visual programming functions and automation facilities for program mapping and execution. VisualGOP is a generic programming environment independent of programming languages and platforms. GOP programs constructed under VisualGOP can run in heterogeneous parallel/distributed systems. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: parallel and distributed processing; programming environment; graph-oriented model; cluster computing

INTRODUCTION

Parallel and distributed computing technologies are becoming increasingly popular and mature. The expertise required to apply these technologies is, however, becoming highly specialized. This is because parallel/distributed programming is much more complex than programming in a sequential paradigm. Many functions such as parallel execution, task mapping, inter-process communication, synchronization and reconfiguration are inherently difficult to program [1,2].

*Correspondence to: Jiannong Cao, Software Management and Development Laboratory, Department of Computing, Hong Kong Polytechnic University, Hung Hom, KLW, Hong Kong.
†E-mail: csjcao@comp.polyu.edu.hk

Contract/grant sponsor: Hong Kong Polytechnic University; contract/grant number: H-ZJ80

Copyright © 2005 John Wiley & Sons, Ltd.

Received 11 November 2002
Revised 25 May 2004
Accepted 11 January 2005

Providing structured and high-level abstractions can greatly simplify the programming task and reduce the development time. We have proposed a graph-oriented programming model, called GOP, for configuration and high-level programming of modular parallel/distributed systems [3,4]. GOP provides high-level abstractions to designing and programming parallel and distributed systems by providing built-in support for a language-level construct and various operations on a high-level abstraction called the *logical graph*. With GOP, the configuration of the interacting processes of a parallel/distributed program can be represented as a user-specified logical graph that is mapped onto the physical network topology. The programming of inter-process communication and synchronization between local processors is supported by built-in primitives for graph-based operations [5].

This paper describes VisualGOP, a visual programming tool developed for supporting the design and coding of graph-oriented parallel/distributed programs. VisualGOP provides integrated graphical tools for creating, compiling, and executing programs. By its graphical nature, VisualGOP is at a higher level of abstraction than GOP. With the visual components provided by VisualGOP, a programmer creates a graph-oriented parallel/distributed program by visually constructing a graph, whose nodes represent local programs (LPs), and edges represent the communication and synchronization between the LPs. To run the constructed program, the programmer can compile the LPs and visually map them onto physical processors in a network environment. The VisualGOP library facilitates the inter-LP communication and synchronization. Through the visual facilities, the programmer does not need to code the textual specification of the graph construct and manage the mappings between logical processes and physical processors.

The characteristics that meet the design goals of the VisualGOP framework can be summarized as follows.

- *Visual programming*: the programmer interacts with VisualGOP through a visual representation of the design of a parallel/distributed program. The design is presented as a logical graph, where the nodes represent the LPs of the program and the edges represent the interactions between the LPs. VisualGOP provides a graph drawing and editing facility to support the creation and modification of logical graphs. It also includes a program editor for visually manipulating the textual source code within an LP.
- *Portability support*: LPs can be written in different languages based on the existing platforms, compilers and program libraries. The GOP model provides a set of high-level programming primitives for writing parallel/distributed programs. VisualGOP defines icons representing various GOP primitives and automatically translates them to their textual representations.
- *Binding of local programs to graph nodes*: VisualGOP provides a visual support for the programmer to specify the binding of LPs in a parallel/distributed program to the nodes of the constructed logical graph.
- *Mapping of graph nodes to processors*: a programmer has the freedom of control over the mapping of graph nodes to processors. Using a high-level graphical notation and its operations, the programmer can specify the processor assignments manually, or let VisualGOP assign processors automatically.
- *Execution*: the constructed GOP program can be distributed to, compiled and executed on the specified processors.

The main focus of this paper is to introduce the GOP model for high-level parallel/distributed programming, describe the VisualGOP architecture and framework, and present the major features of VisualGOP. These features provide the support for the program development lifecycle, from program

Copyright © 2005 John Wiley & Sons, Ltd.

construction to process mapping and compilation. VisualGOP also supports the interoperability based on XML. We present how to use VisualGOP to construct a GOP program and report the performance evaluation result.

THE VisualGOP CONCEPTS

The GOP model

In the GOP model [3,6], a parallel/distributed program is defined as a collection of LPs that may execute on several processors. Parallelism is expressed through explicit creation of LPs and communication between LPs is solely via message passing. The distinct feature of GOP is that it allows programmers to write distributed programs based on user-specified graphs, which serve the purpose of naming, grouping and configuring LPs. The graph construct is also used as the underlying structure for implementing uniform message passing and LP coordination mechanisms.

The key elements of GOP are a logical graph construct to be associated with LPs and their relationships, and a collection of functions defined in terms of the graph and invoked by messages traversing the graph. As shown in Figure 1, the GOP model consists of the following.

- A *logical graph* whose nodes are associated with LPs, and edges define the relationships between the LPs.
- An *LPs-to-nodes mapping*, which allows the programmer to bind LPs to specific nodes.
- An optional *nodes-to-processors mapping*, which allows the programmer to explicitly specify the mapping of the logical graph to the underlying network of processors. When the mapping specification is omitted, a default mapping will be performed.
- A *library of language-level graph-oriented programming primitives*. GOP provides a high-level abstraction with a library of primitives that can be implemented in a language such as C, C++ and Java. In other words, the implementation of GOP may be different on each platform, but the concept of using GOP to design the parallel/distributed program is the same.

Once the local context for the graph instance is set up, communication and coordination of LPs can be implemented by invoking operations defined on the specified graph. The operations on a user-specified graph can be categorized into several classes.

- *Communication and synchronization* operations provide various kinds of communication primitives that can be used to pass messages from one node to another, or to other nodes in the graph. These primitives can be used by LPs to communicate with each other and to synchronize their operations without knowing the low-level details such as name resolution, addressing and routing.
- *Subgraph generation* operations derive subgraphs such as a shortest path between two nodes and spanning trees of a graph. Many distributed algorithms rely on the construction of some forms of subgraphs of the underlying control graph.
- *Query* operations provide information about the graph, such as the number of nodes in the graph, current location of a LP, and whether an edge between two given nodes exists. This information is useful for many system control and configuration functions.

Copyright © 2005 John Wiley & Sons, Ltd.

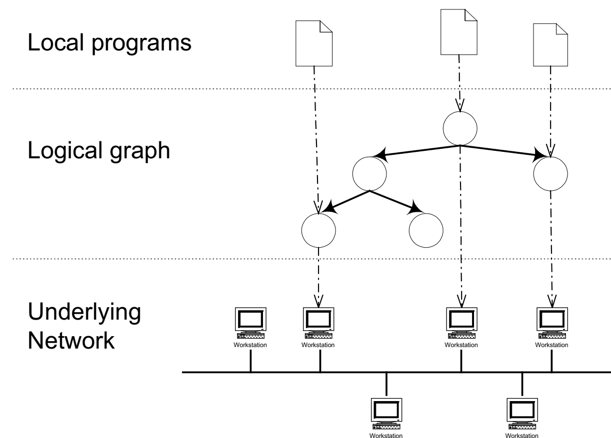

Figure 1. The GOP conceptual model.

- *Update* operations provide the programmer with the capability to dynamically insert into and/or delete edges and nodes from a graph. Also, mappings can be dynamically changed during the program execution. These primitives are very useful for dynamic control functions such as dynamic reconfiguration of distributed programs.

Therefore, GOP is a programming methodology that provides graph semantics that allow a programmer to exploit the relationships between the nodes, physical processors and the local programs during the program design. GOP programs are conceptually sequential but augmented with primitives for binding LPs to nodes in a graph. The programmer has a clear view of the overall communication pattern through the graph, while the operations on the graph and details of inter-node communications such as parameters and contents are completely hidden. The sequential code of LPs can be written using any programming language such as C, C++ or Java.

The architecture and framework

This section introduces the overall architecture and the visual programming framework of VisualGOP. A detailed description of VisualGOP's major components and their salient features will be discussed in the next section.

VisualGOP consists of the following major components.

- *Logical graph construction*. This component is provided for constructing a GOP program through visual programming by representing the logical program structure graphically. VisualGOP is capable of manipulating graph structures in both graphical and textual (XML-based) forms and can transform from one form to another.
- *LP editing tool*. This tool is used for editing the source code of LPs in a GOP program.

- *Network resource management*. This component provides the programmer with control over networking resources. It facilitates the mapping of LPs to graph nodes and the mapping of graph nodes to processors. It also allows the programmer to access the information about the status of the network elements.
- *Compilation tool*. This tool is used for transforming the diagrammatic representations and GOP primitives into the target machine code for execution.

These components are organized to form the architecture of VisualGOP, as shown in Figure 2. They are divided into two levels: the visual level and the non-visual level. Each component has its own data storage and communicates with other components through a shared, common data structure.

The visual level components present design views, controlled by a visual graph editor and a text program editor. They also provide the mapping and deployment tools, which are accessed through a mapping panel and a processor panel. Figure 3 shows a screendump of the main user-interface of VisualGOP.

The non-visual level contains components responsible for maintaining a representation of the GOP program design and deployment information. This representation is kept in memory during the program design, and later stored in a file, in either a format internal to VisualGOP or an XML format. This level also contains the distributed Remote Execution Manager, which uses the stored information to execute the constructed GOP program on a given platform.

Using VisualGOP

With VisualGOP, the programmer starts program development by building a highly abstract design and then transforms it successively into an increasingly more detailed solution. More specifically, VisualGOP separates program design and configuration (i.e. the definition of a logical graph) from the implementation of program components (i.e. the coding of LPs). This principle of separation of concerns facilitates the reconfiguration of the program structure independent of LP coding.

The visual programming process under VisualGOP consists of the following iterative stages.

1. Specify the logical graph representing the configuration of a parallel/distributed program. The logical graph consists of a set of nodes representing LPs and a set of edges representing the interactions between the LPs in the program. The programmer uses the *graph design editor* to visually create the logical graph.
2. Create LPs and map them onto the nodes of the logical graph. There are two ways to do this. One way is to create the LPs first and then bind them to the graph nodes. Another is to combine the two steps into one—click on a node of the graph to open the text editor in which the code of the LP mapped to the node can be entered.
3. Map the nodes of the graph to a network of processors. The *Mapping Panel* of VisualGOP displays the GOP program elements (nodes, processors, LPs) in a hierarchical tree structure. LPs, nodes and processors can be added to and deleted from the panel. The programmer can use drag-and-drop to bind and unbind the LPs to graph nodes and the graph nodes to processors. The binding information concerning a node can be viewed by selecting and clicking on the node. The *Processor Panel* provides icons for displaying processors and their inter-connections.

Copyright © 2005 John Wiley & Sons, Ltd.

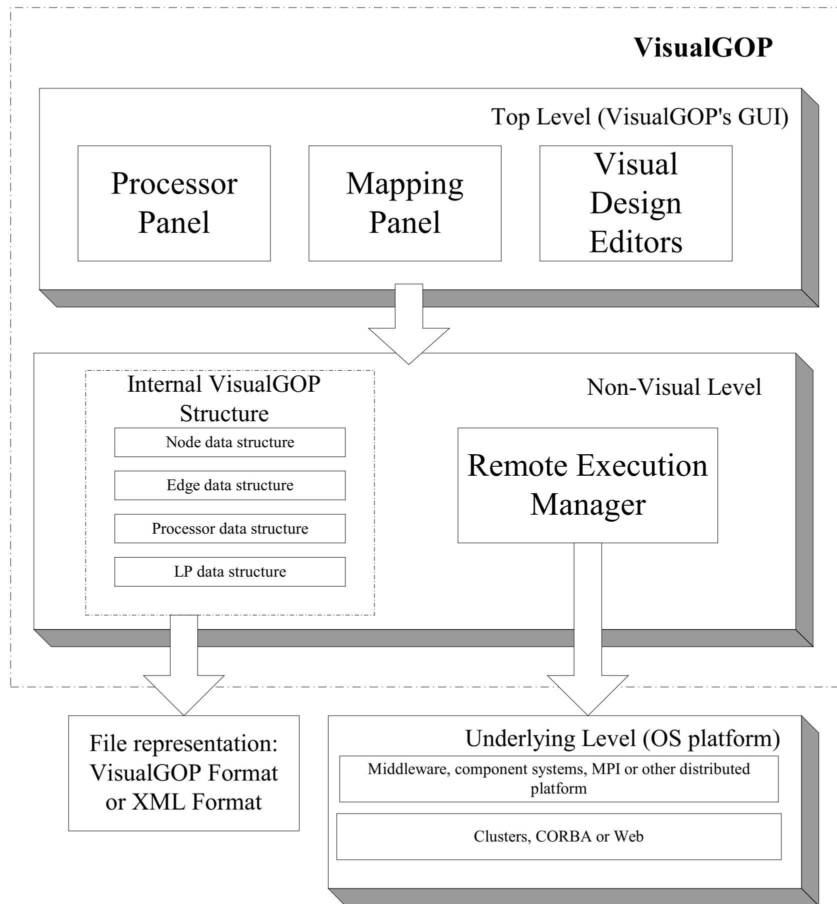

Figure 2. The VisualGOP architecture.

When a processor is added, a new processor icon will be shown on the panel. For node-to-processor mapping, the panel also provides the drag-and-drop function to bind and unbind graph nodes to one of the processors in the panel.

4. Compile the LPs and execute the application. When the programmer needs to deploy the program to other platforms, VisualGOP will first distribute the graph information, LP source files, and the network information to the target machines. Source files are then compiled on the target machines. Finally, the constructed GOP programs are executed on the specified processors. Outputs will be displayed on VisualGOP.

Copyright © 2005 John Wiley & Sons, Ltd.

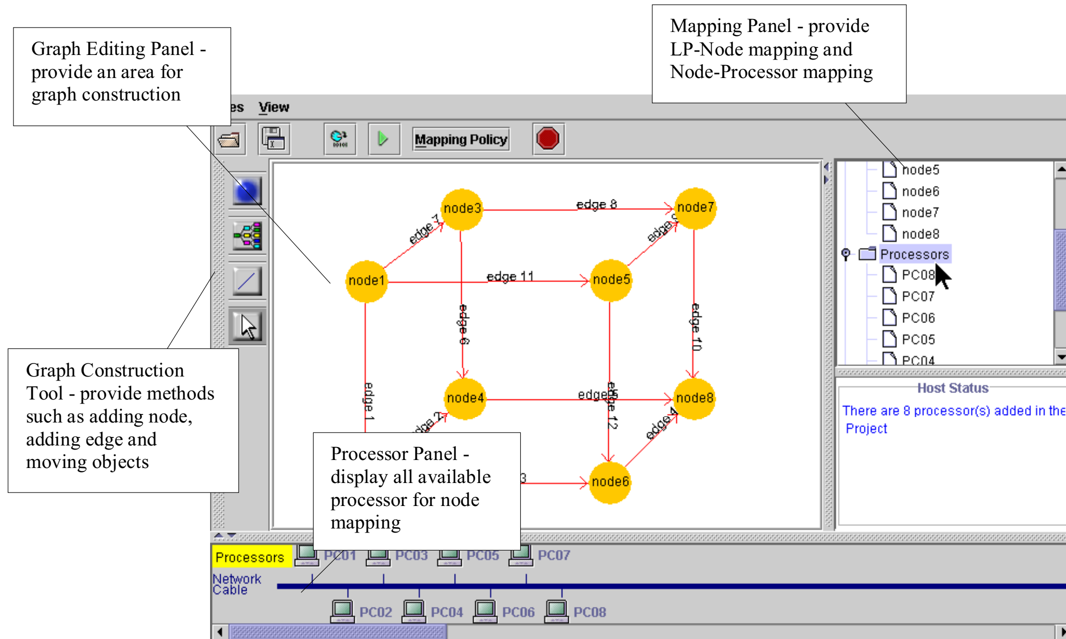

Figure 3. The main screen of VisualGOP.

The following sections describe in more details various features supported during the programming process. We will use an example, i.e. the finite differential method (FDM), to illustrate how VisualGOP supports high-level and visual parallel programming. FDM is an approach that obtains an approximate solution to a partial differential equation governing the behavior of a physical system. The method imposes a regular grid on the physical domain. It then approximates the derivative of an unknown value u at a grid point (x, y) by the values of adjacent grid points. Consider a specific partial differential equation—the Laplace equation. The approximation u_{ij} to the exact solution $u(x_i, y_j)$ on the grid satisfies the following equation

$$u_{ij} = \frac{1}{4}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}), \quad i, j = 0, \ldots, N-1 \tag{1}$$

Equation (1) shows that the value of u at any point is affected by four adjacent elements. Given the initial value, the value of u at any point can be approximated by iteration

$$u_{ij}^{(k)} = \frac{1}{4}(u_{i+1,j}^{(k-1)} + u_{i-1,j}^{(k-1)} + u_{i,j+1}^{(k-1)} + u_{i,j-1}^{(k-1)})$$

until a predefined accuracy level is reached.

Copyright © 2005 John Wiley & Sons, Ltd.

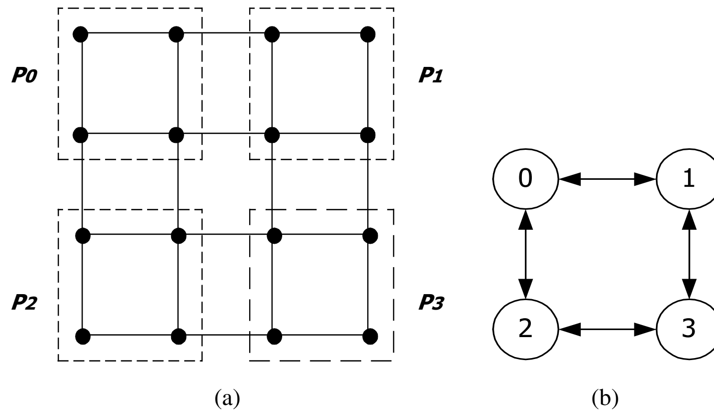

Figure 4. Grid partitions for four processors: (a) 2D grid partition for four processors; (b) program graph for four processors.

PROGRAM CONSTRUCTION

Graphical programming

When designing distributed programs, the programmer commonly draws an informal directed graph that shows the distributed structure. Such a graph may concentrate only on the interactions among the graph nodes while ignoring the inner workings of the nodes.

VisualGOP uses the graph abstraction method to represent distributed programs. It divides a distributed program into several LPs and defines their interactions. Each graph node can be allocated on a processor in a distributed system and then an LP can be assigned to the node. As LPs are finally located on distributed processors, their interactive behaviors describe the message-passing performed between distributed processors.

When solving the FDM problem on p processors, the grid will be partitioned into p sections. It can be decomposed in different manners. Here, a two-dimensional partition is used to generate a coarse grid. Figure 4 shows the grid partitions for four processors. Figure 4(b) depicts the program graph of the FDM derived from this partition. The graph has the coarse-grid topology corresponding to the physical topology of the grid. The entry and exit nodes are omitted in the program graph because they are not important in explaining the problem and solution.

First, an appropriate logical graph of the distributed application should be well designed. Based on the design, the programmer can use the Graph Editing Panel to draw the graph, which will be translated into a textual form. The Graph Editing Panel displays the graphical construction view, and allows the program's logical graph structure to be edited. Controls are provided for drawing the graph components within this panel. These controls include functions for adding nodes, moving nodes, and generating

Copyright © 2005 John Wiley & Sons, Ltd.

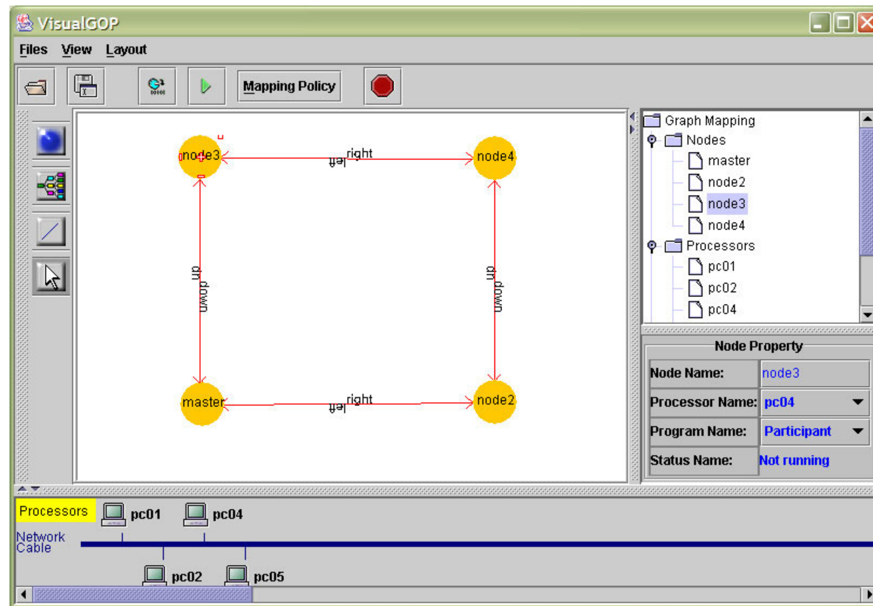

Figure 5. Logical graph for four processors in VisualGOP.

sub-graph nodes and edges. Figure 5 shows how nodes can be added to the graph by simply clicking on the add-node button in the Graph Construction Tool. Also, edges can be added between two nodes by joining the source and destination nodes.

A visual element (e.g. node or edge) can be manipulated directly on the screen when constructing a program. The screen has several display areas, each depicting certain properties of the visual element and its role in the program, and the mapping relationship that it belongs to. The properties are updated automatically whenever the program structure is modified in the Graph Editing Panel. The properties of a visual element can be presented visually, such as a button or a pull-down menu. The associated tree panel, the Mapping Panel, presents the element properties in a simple tree structure. It is useful not only as an indicator of the overall program structure, but also as a navigational aid for browsing different parts of the system. Navigation is assisted by highlighting the part of a tree diagram that corresponds to the element being displayed in the Graph Editing Panel. Due to the limited screen estate, only a limited number of visual elements can be viewed and operated on the screen. It is desirable for a programmer to hide the information that is not currently needed, and to show as many objects as a programmer wants to view and operate.

Copyright © 2005 John Wiley & Sons, Ltd.

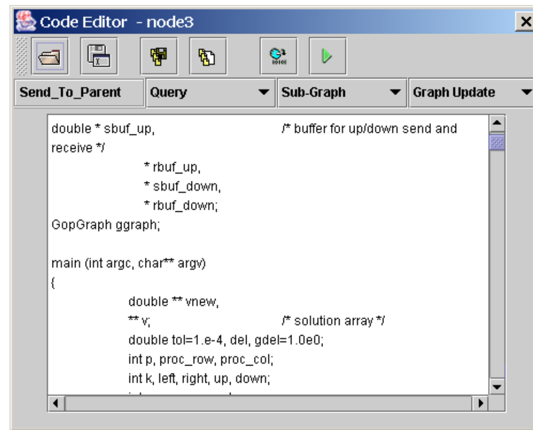

Figure 6. Editing a local program.

LP source code editing

The logical graph constructed on the Graph Editing Panel represents only a skeleton of the target topology without any semantics. It becomes meaningful only when the program code is provided as its contents through the Code Editing Panel (program editor). The program editor is one of the key components in the VisualGOP system and serves not only for source code editing for each graph element but also consistency checking as explained below.

The program editor performs consistency checking to prevent the creation of graph semantic errors. For example, the programmer is not allowed to create incorrect connections, such as connecting a node to a non-existent node. Such semantic errors can be detected only after the source codes have been attached to all the graph nodes. To assist the programmer with visual programming, the editor is syntax-directed such that newly created or modified node or edge names are correct within the logical graph structure. Further details are discussed in the next section.

Source code editing supports the mapping of a node to the program code. There are two methods for invoking the Code Editing Panel:

1. start editing a new source code program;
2. invoke the mapped LP by double-clicking on the node.

To edit a new program, the programmer starts the program editor from the Mapping Panel. In our FDM example, the programmer writes two LPs using the C language: the *Participant*, which calculates and submits the values and then collects the values from the neighboring nodes, and the *Coordinator*, which acts like a Participant node with extra functions for initializing the application and outputting the result. First, the programmer creates the *Participant* in the program editor (see Figure 6) and maps the LP to a node. When editing the master node, VisualGOP allows the programmer to load the *Participant* inside

Copyright © 2005 John Wiley & Sons, Ltd.

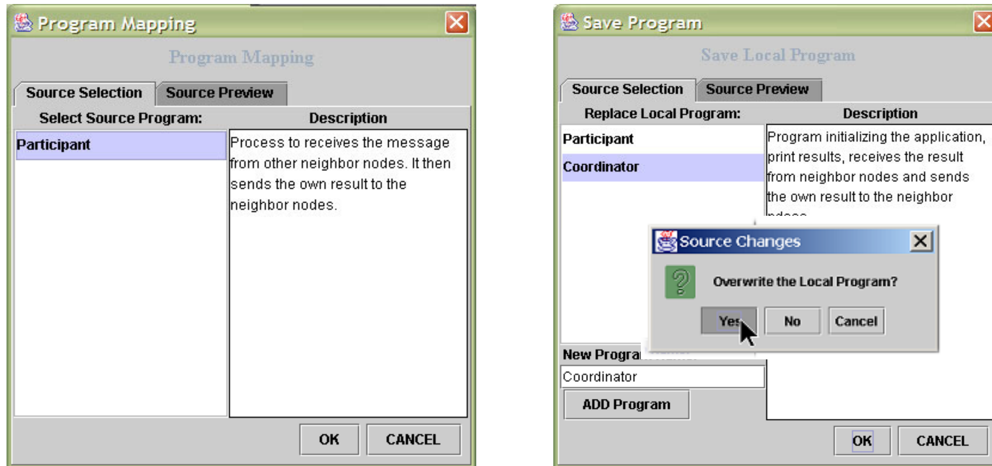

Figure 7. Program loading.

the program editor (see Figure 7). A LP can be made in the *Coordinator* by adding new functions. The created *Coordinator* will bind to the master node automatically.

Generation of GOP primitives

The GOP model provides high-level abstractions for programming distributed programs by directly supporting logical graph operations (i.e. GOP primitives). Using the basic GOP primitives, the programmer does not need to know the details of the low-level communication mechanism. He or she only needs to know how to use the selected GOP primitives. In the Code Editing Panel, GOP primitives can be automatically generated from visual icons. As described in the previous section, the primitives can be categorized into several groups: communication, query, sub-graph generation and graph update. A programmer can generate GOP primitives via two methods:

1. adding the GOP primitives from the editor's menu directly (see Figure 8);
2. dragging graph nodes directly into the Code Editing Panel (see Figure 9).

CONSISTENCY CHECK

The program editor can automatically check the consistency of a program code using the graph structure specified by the programmer. When the programmer opens the Code Editing Screen, the automatic consistency checking module is also started, as shown in the flowchart in Figure 10. While the programmer is editing the source code, the editor will submit the current code statements

Copyright © 2005 John Wiley & Sons, Ltd.

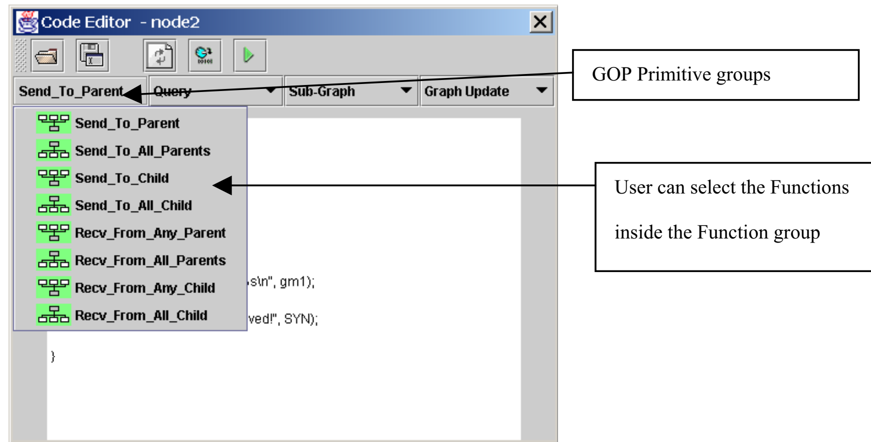

Figure 8. The GOP primitive menu.

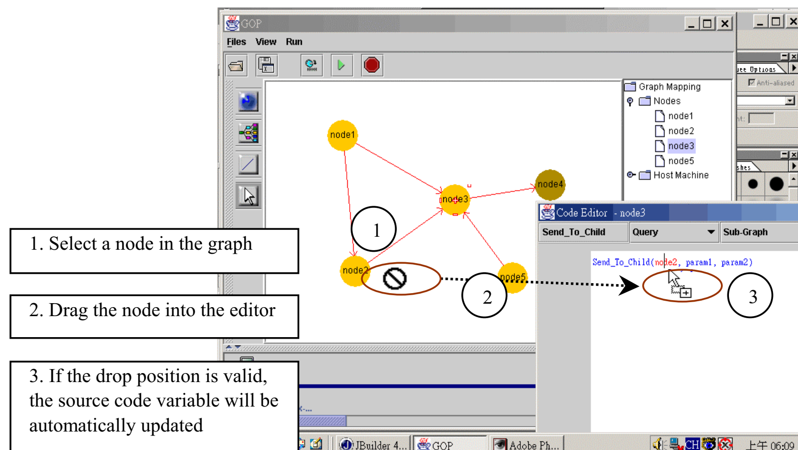

Figure 9. Manipulating graph nodes directly.

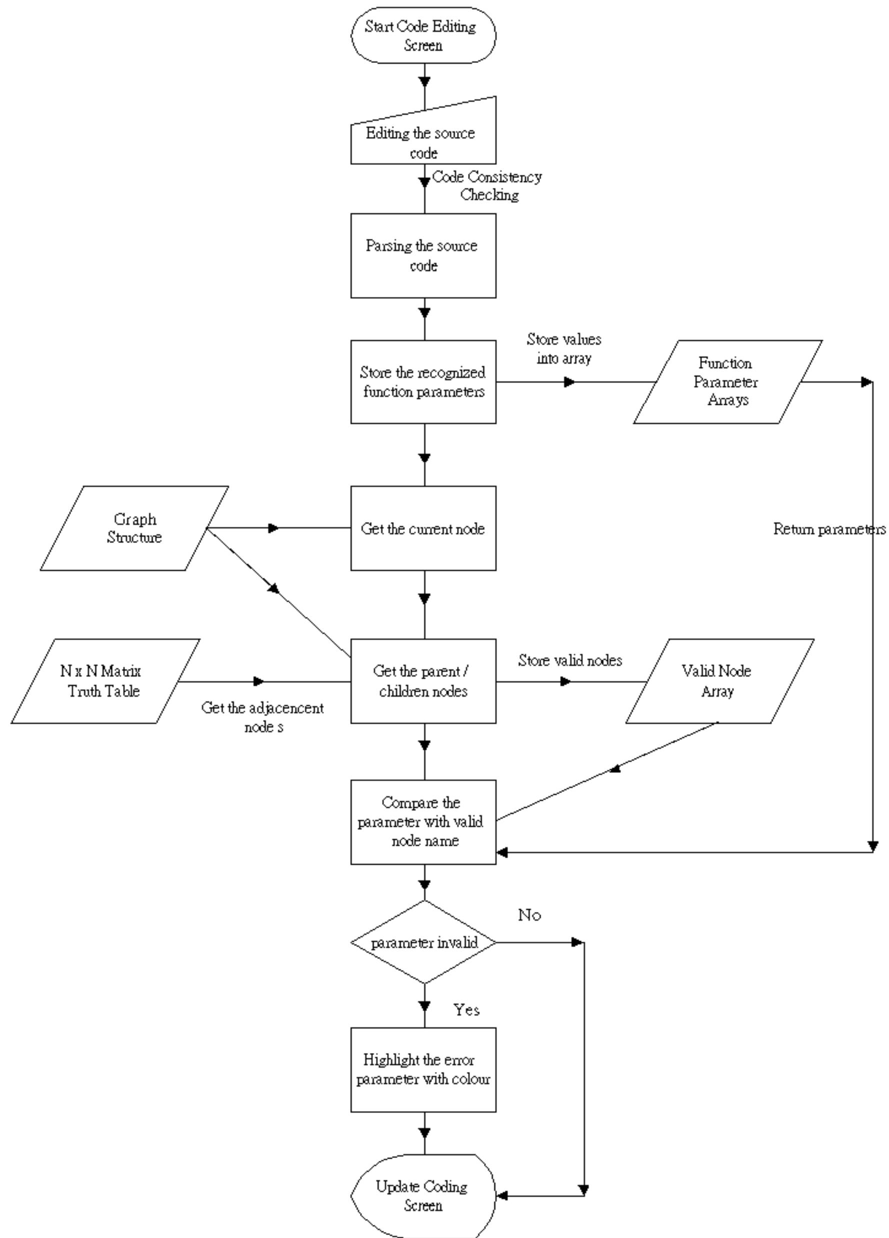

Figure 10. Flow chart of automatic consistency checking.

Copyright © 2005 John Wiley & Sons, Ltd.

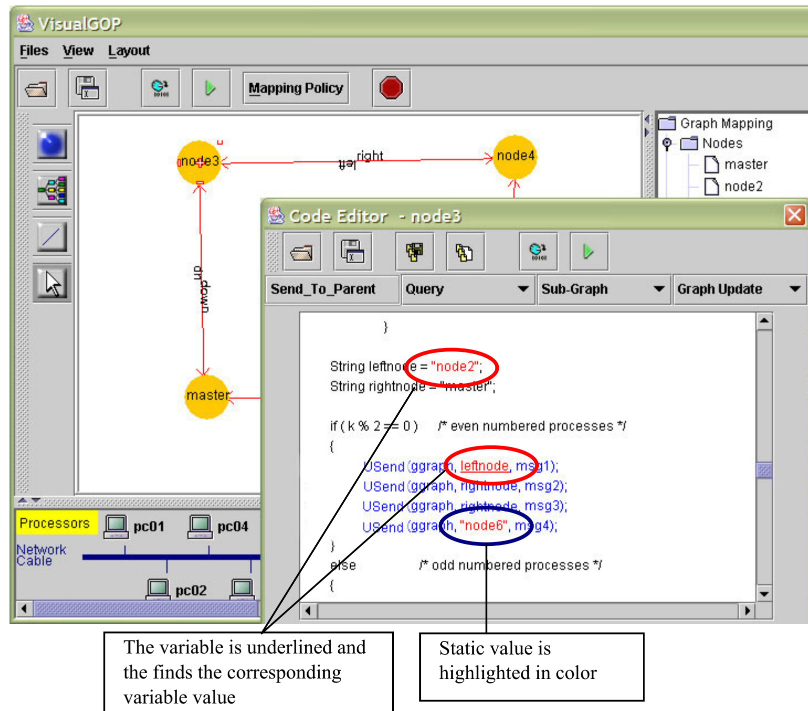

Figure 11. Participant program source.

to the source code parser. Then, the parser finds the supported GOP primitives and its parameters, stores them in a variable array called the Function Parameter Array. After that, the parser obtains the name of each node under editing from the Graph Structure (an internal VisualGOP structure). Based on the $N \times N$ matrix truth table, the parser can find the parent nodes of any child node and store the result in the Valid Node Array. By comparing the Function Parameter Array and the Valid Node Array, the consistency checking module can determine whether the parameters in the GOP primitives contain a valid node name. If a parameter is found to be invalid, the corresponding part of the source code will be highlighted to indicate that the parameter is inconsistent with the graph structure. Then, the programmer is expected to notice the problem and correct the parameter value. The current implementation of consistency checking is limited to static variables. A running program cannot be corrected for its errors. What is gained from the static implementation is the reusability of program codes. To reuse a code, the programmer can use relevant GOP primitives (e.g. getEdgeNode()) to generate the node name during runtime.

Figure 11 shows an active screen of a *Participant* in the program editor and the GOP primitive *USend* is used in this example. USend() defines a unicast message-passing primitive, delivering a message

Copyright © 2005 John Wiley & Sons, Ltd.

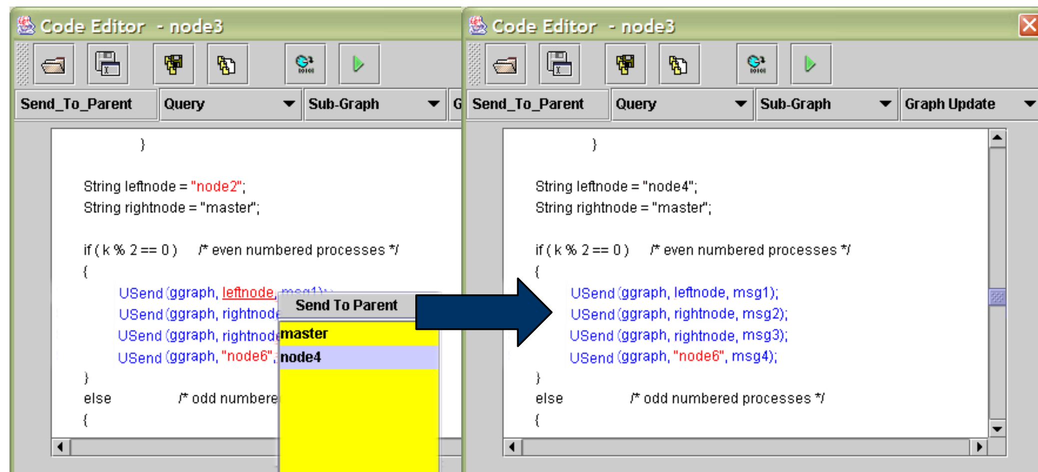

Figure 12. Validation in a participant program.

from the current node to another node. The second parameter of *USend* accepts a node variable or value. When the parameter is detected as invalid by the program editor, it will be highlighted in color. Also, if the parameter is a variable (i.e. a reference value), it will be underlined, and the corresponding variable value will be searched and retrieved automatically from the source code.

When the programmer clicks on an invalid parameter value, a list of valid values will pop up for the programmer to select the correct node name (see Figure 12). If the selected parameter is a static string, the program editor updates the string directly. Otherwise, the program editor will find out the actual value of the selected variable, and update it with the valid node value.

VisualGOP supports the interaction of visual components with LPs. The programmer can also drag and drop nodes into the GOP primitives, so that parameter values will be updated to be valid. In Figure 13, the programmer has two options (node4 or master) for updating a parameter. By providing the programmer with automated, intelligent assistance throughout the software design process, VisualGOP creates a flexible programming environment and eliminates most of the mundane clerical tasks.

SUPPORT FOR SCALABILITY

VisualGOP allows the programmer to create graphical representations for realistic parallel/distributed applications. It should be highly scalable and adaptable to the parameters such as the problem size and number of processors. Our graph scaling algorithms and graph mapping strategy are based on the Task Interaction Graph (TIG) [7,8]. TIG provides a concise topology to describe process-level computation and communication. It has a flexible structure for graph scaling. In graph scaling, the nodes in the graph

Copyright © 2005 John Wiley & Sons, Ltd.

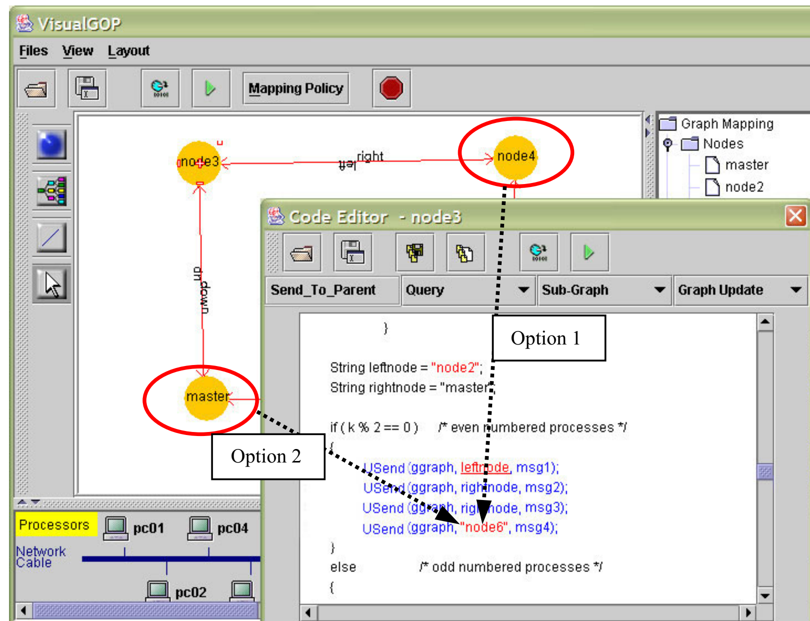

Figure 13. Updating GOP primitive parameter through the Graph Editing Panel.

can be decomposed or merged, and the edges are reconstructed based on the original graph structure to produce a new task graph to match the parameters.

Graph scaling can be made in two modes. If the number of parallel tasks is less than the available number of processors, graph expansion will be performed to generate more tasks. On the other hand, if the number of parallel tasks is greater than the available processors, the graph may be compressed to include fewer nodes. This approach has been implemented and reported in an earlier paper [9].

The programmer can inform VisualGOP of a graph pattern that is particularly suited to the domain application under consideration. For example, to implement FDM, we use a mesh structure (see Figure 14) to model the basic pattern of the task graph, so that the solution can be scalable to the application size.

We choose the mesh tree template for our application as shown in Figure 15. All the LPs can use one copy of the source program, which is thus easy to design and maintain when the application grows large. When editing the LPs, we use the GOP primitive that can generate the node name from its neighboring edges, so that the static node name is not required by the system. VisualGOP then expects the programmer to input the processor number, and performs graph expansion if the available processor number is larger than the number of graph nodes, or graph compression otherwise. The programmer can preview the expanded or compressed graph and accept or reject the graph update.

Copyright © 2005 John Wiley & Sons, Ltd.

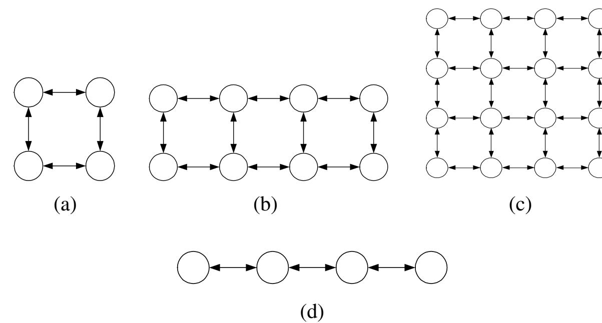

Figure 14. Graph expansion for mesh structure: (a) original 2×2 mesh; (b) expanded 2×4 mesh; (c) expanded 4×4 mesh; (d) 1×4 mesh.

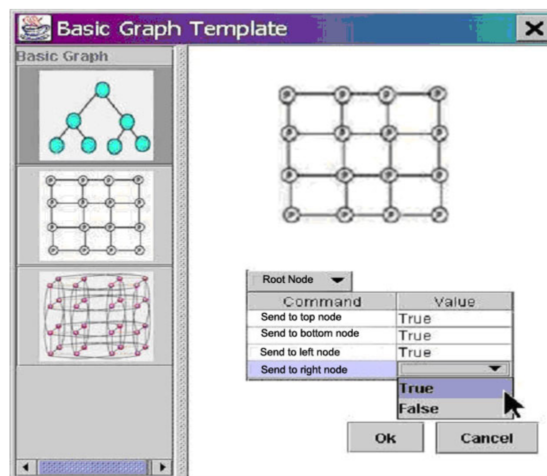

Figure 15. The graph template for the basic graph diagram.

Another example is the tree pattern for graph expansion. The tree in Figure 16(a) can be expanded in two directions. One is breadth-oriented expansion as shown in Figure 16(b), in which all the expanded nodes are attached to the root. The other is depth-oriented expansion in which functional nodes spawn children to be connected beneath as shown in Figure 16(c).

In this tree example, the master task initiates N worker tasks and distributes an equal portion of an array to each worker task. When a worker task receives its portion of the array, it performs a simple value assignment to each of its elements. The value assigned to each element is simply that element's index in the array plus one. Each worker task then sends its portion of the array back to the master task.

Copyright © 2005 John Wiley & Sons, Ltd.

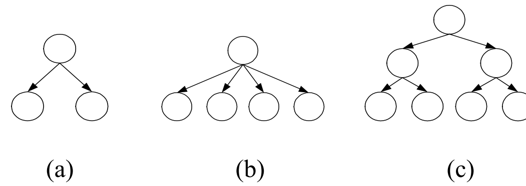

Figure 16. Graph expansion applied on a tree structure: (a) original tree; (b) breadth-oriented expansion; (c) depth-oriented expansion.

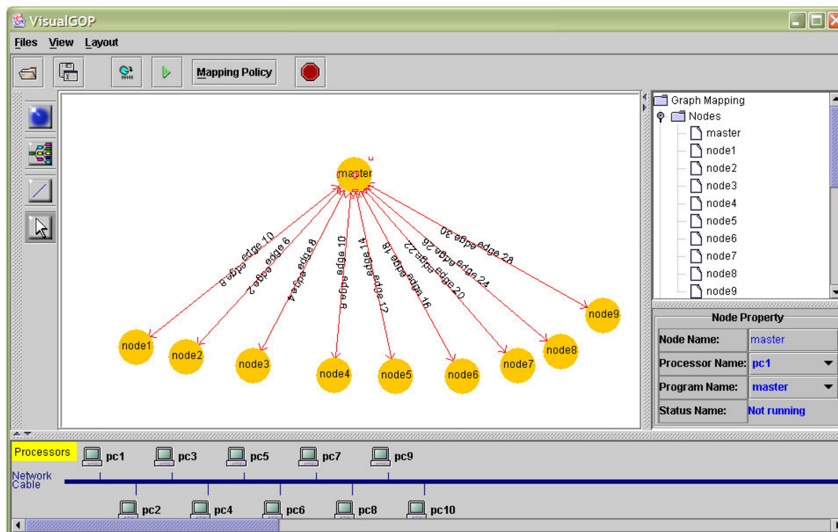

Figure 17. Array assignment program in VisualGOP.

VisualGOP helps the programmer to expand the graph by breadth-oriented expansion (see Figure 17). The array distribution is not affected since the programmer can arrange the array assignment according to the number of processors.

TWO-STEP MAPPINGS

An important task in implementing GOP is to manage the mapping of LPs to the nodes of a logical graph (LP-node mapping), and the mapping of graph nodes to the underlying network processors (node-processor mapping). LPs need to be bound to the nodes of a logical graph for naming,

Copyright © 2005 John Wiley & Sons, Ltd.

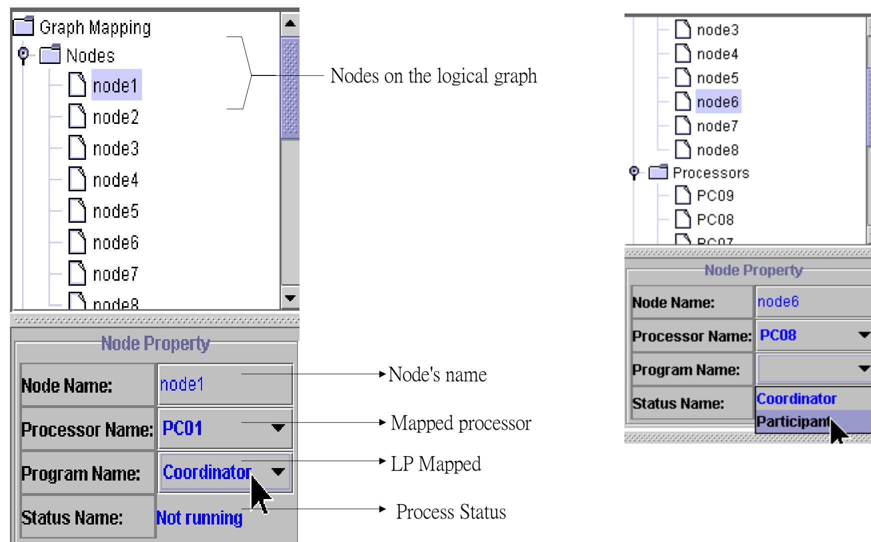

Figure 18. LP-node mapping.

configuration, and communication purposes. For node-processor mapping, if the programmer specifies the mapping, the solution to the problem becomes straightforward. Otherwise, the GOP system could provide support for task allocation in order to make efficient use of system resources and/or to speed up the computations. Once mapped, the graph node has all this required information (by node-processor mapping) such as IP address, compiler path, etc.

A mapping example

When the programmer selects the status of a specific node, a LP can be chosen from the node property's pull-down menu. After the selection, the LP is mapped to that node (see Figure 18). Also, the mapping can be made by dragging and dropping the node into one of the processors in the Processor Panel.

Let us define a mapping, named M1, for the relationships between graph nodes and LPs. In the mapping, there are two types of LPs: the Coordinator LP, which receives and distributes the data, and the Participant LPs, which calculate and submit their own partial data and receive data from the Coordinator. Our definition of M1 is (given in the C language, where LV-MAP is the corresponding mapping data type):

LV-MAP M1 =
{ {0, "Coordinator"}, {1, "Participant"}, {2, "Participant"}, {3, "Participant"},
. . . , {N, "Participant"} }

Copyright © 2005 John Wiley & Sons, Ltd.

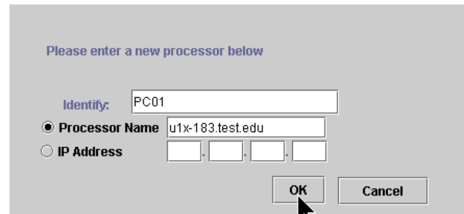

Figure 19. Processor configuration.

Also, the programmer can specify another mapping M2 of the graph nodes onto a network of processors.

Distributed processors are used to run the whole application. In VisualGOP, each node is assumed to be executing on a different processor. For example, an LP on processor A sends a message to another LP on processor B. The message passes from one platform to another, so the node-processor mapping is required to perform such a task.

The Processor Panel displays the information about processors, including whether a processor is currently available, and if it is assigned to a node the node is highlighted. Programmers can manually assign processors to LPs or let VisualGOP assign automatically as explained in the next section.

With the aid of the Processor Panel, a node can be mapped onto a target processor. The mapping is done in two steps. First, the programmer has to configure the target processor in the processor configuration dialog (see Figure 19). Inside the dialog, the programmer can specify the new identity name, IP address or hostname (current implementation allows only one processor per machine). In the second step, the node is mapped onto the configured processor. The mapping can be done manually or automatically. In the manual mode, the programmer clicks on a node in the node property (see Figure 20) and specifies the name of the target processor. The automatic mode will be discussed next.

Automatic mapping

After the program design, the programmer can perform the LPs-to-nodes mapping manually or automatically. In the Single Program Multiple Data (SPMD) model, all the nodes share the same copy of the program, so the mapping is simple. In the Multiple Program Multiple Data (MPMD) model, a node may work on different tasks. The programmer can choose a set of rules to perform the LPs-to-nodes mapping automatically. There are rules for classifying the LP into different groups, e.g. a range of node IDs, similar node names, and node types. Finally, a task graph will be mapped to the network of processors. Each processor is responsible for executing one node in the task graph; i.e. there is a one-to-one correspondence between a processor and a node.

Programmers can also choose a mapping policy for the nodes-to-processors mapping. Using the mapping policy, the system binds nodes to the processors automatically. It simplifies the mapping process and helps programmers to assign the most useful/powerful processors to the more time-critical LPs.

Copyright © 2005 John Wiley & Sons, Ltd.

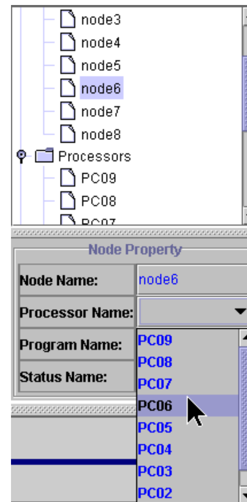

Figure 20. Node-processor mapping.

Map processors sequentially

This simple mapping algorithm maps process nodes to processors in their listed order. The programmer can take advantage of this algorithm by writing programs in the order of their performance priorities and listing the available processors from the most powerful to the least powerful.

Map processors by priority

The programmer specifies the power indices of available processors and weightings of the nodes as illustrated in Figure 21. This mapping algorithm will automatically assign the node of the highest weighting to the most powerful processor, and so on.

REMOTE COMPILATION AND EXECUTION

VisualGOP supports heterogeneous code compilation and remote execution. The GOP library for various graph operations has been implemented in C. Figure 22 shows the process of automatic compilation and execution for C and Java languages that have been implemented. The Code Detector recognizes each LPs implementation language and assigns the corresponding remote compilation and execution procedures to the LP, which will be compiled, linked to the GOP library, and executed automatically. Using this technique, remote compilation and execution are simple and at a high level—a single programming environment could target distributed codes to run on a variety of systems.

Copyright © 2005 John Wiley & Sons, Ltd.

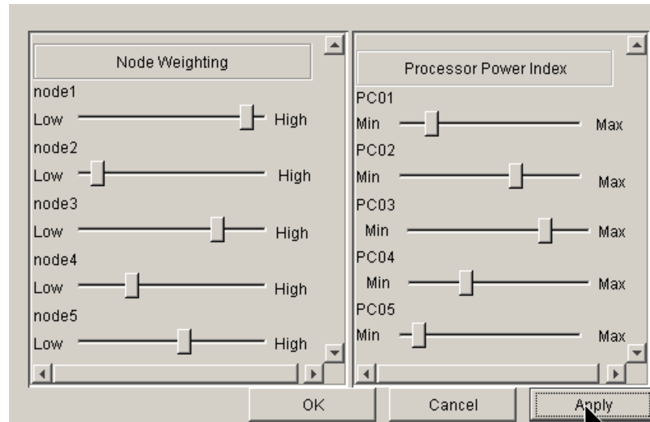

Figure 21. The mapping index table.

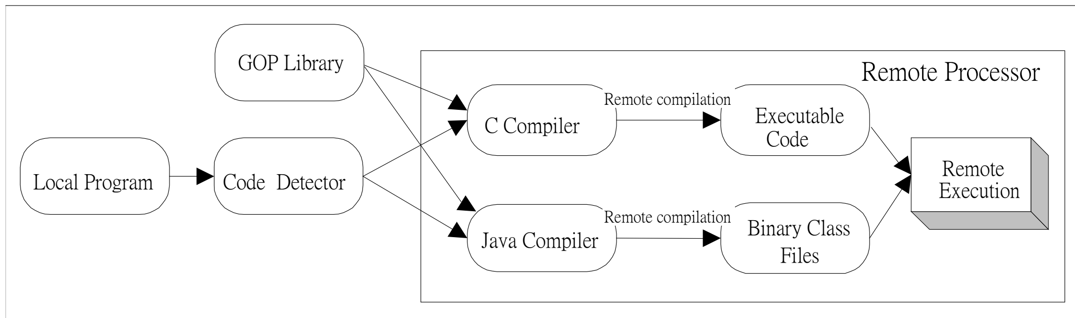

Figure 22. Automatic code generation framework.

Each target (remote) machine has a daemon process installed which listens to and processes any requests from VisualGOP. When VisualGOP submits a request for remote compilation or execution, a connection is immediately established between VisualGOP and the remote machine.

When the programmer selects the remote compile option within a node, a selection dialog (see Figure 23) is provided for the programmer to manually choose a processor as the target machine for compilation and execution. The machines' names are specified by the programmer on the Graph Panel. After a processor is selected, the code is ready for compilation. If the process/processor mapping is performed automatically, the above manual assignment process is skipped. For a Java LP, upon the programmer's input of the command-line arguments needed for compiling or executing the LP on the

Copyright © 2005 John Wiley & Sons, Ltd.

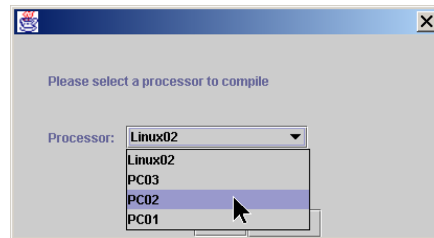

Figure 23. Dialog for choosing the host machine to compile.

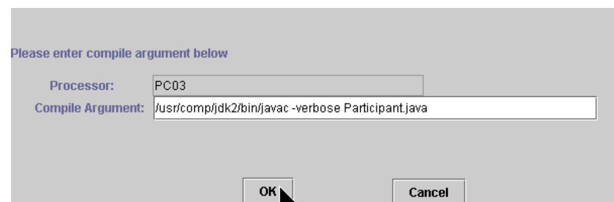

Figure 24. Dialog for entering compilation arguments.

target machine (see Figure 24), the GOP library is invoked through Java's native code interface, and remote compilation and execution are performed automatically. For a C LP, each different type of machine is equipped with a pre-configured makefile which is invoked by VisualGOP to run on the remote machine for compilation and execution.

The execution option is similar to the compilation option. After compilation and execution, the results are sent back to the VisualGOP console to be displayed on an output window (see Figure 25).

INTEROPERABILITY

Graphical data represented in the GOP system sometimes need to be exchanged between VisualGOP and other graph-based parallel/distributed programming systems. To enable interoperability between different tools performing various tasks on heterogeneous platforms, a standard representation is needed for the exchange of the GOP logical graphs between different systems. Other benefits of the standard textual representation of logical graphs include the fact that structured or semi-structured graphical data may be transformed into any other data representation schemes automatically, and also are easily understood by a human reader.

XML [10] becomes our natural choice for defining the VisualGOP data formats, due to its increasing popularity and wide acceptance as a standard data interchange description language. One of the main

Copyright © 2005 John Wiley & Sons, Ltd.

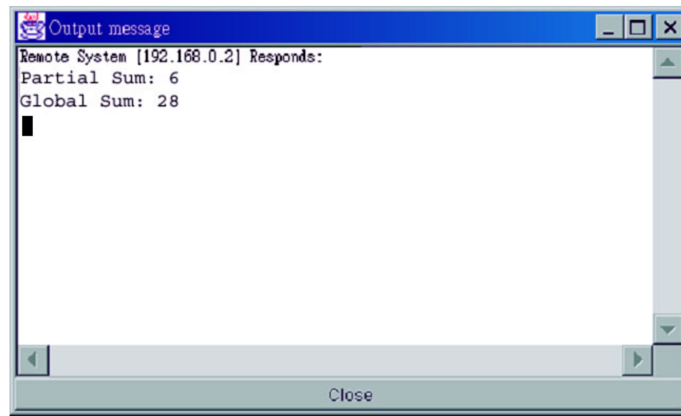

Figure 25. Dialog for returning the remote command result.

motivations of using XML to describe graphical data textually is to allow the data to be exchanged between clusters of CORBA (Common Object Request Broker Architecture) and Web-distributed applications using VisualGOP. Programmers can also reuse graphs in XML formats by importing them into VisualGOP. Additionally, the standard textual form can be used for system analysis and reuse of graphs by different applications.

An XML document may include features in a graph that are not directly relevant to the graphical representation seen by the programmer. For example, processor configurations, LP definitions, and other additional graph descriptions may be included in an XML description. For data integrity, the XML Schema is used for checking the textual XML graph and maintaining correct graphical data types and structure.

The graph description languages based on XML, such as GraphXML [11] and GXL [12], provide interchange formats for graph drawing and information visualization applications, with provisions for extension. We propose an extension to GraphXML, called *GOP-XML*, to suit the systems like VisualGOP for graph-based parallel and distributed programming. GOP-XML supports the specification of processor configuration, LP definition, various edge and node attributes, etc. Each graph in GOP-XML can be represented as an XML document.

The example of GOP-XML code in Figure 26 shows the basic style of the graph description. The first line describes the file as being in XML format. The second line indicates the title and the type of the graph. The third section (between the <GraphTopology> and </GraphTopology> tags) describes the logical graph in GOP. The fourth section (<Processors> tag) provides details of the processors used. The fifth section (<LocalPrograms> tag) defines the local programs, followed by the sixth section (<API> tag) specifying the GOP primitives used in the application. Finally, the last section (<Additional> tag) adds the supplementary information about the graph, such as graph type, graph element descriptions. Attributes can also be defined as value pairs for each of the elements.

Copyright © 2005 John Wiley & Sons, Ltd.

```
<?xml version= "1.0" encoding="utf-8"? >
<Graph Title="Sample GOP Graph" Type="Tree">
  <GraphTopology>
    <GraphLayout Number="0">
      <Node Name="master" IsGroup="false" Interface="Coordinator">
      </Node>
      <Edge Name="edge1">
      </Edge>
    </GraphLayout>
  </GraphTopology>
  <Processors>
      <Processor Name="pc01" Type="IP/Hostname">
      </Processor>
  </Processors>
  <LocalPrograms>
      <LocalProgram Name="lp1" LangType="c">
      </LocalProgram>
  </LocalPrograms>
```

Figure 26. Basic structure of GOP-XML.

The GOP-XML schema defines a set of rules to check the allowable elements, attributes and structures in the XML. A parser has been developed to validate the syntax of any GOP-XML document.

When the programmer needs to compile and execute the application, a GOP-XML document will be transferred to the corresponding processors. As shown in Figure 27, when VisualGOP is required to start the remote compilation and execution, its XML conversion engine will collect the graph, the mapping and network information, and translate them into the GOP-XML format. The GOP-XML file and the LPs are then deployed to the processors of the target platform. Independent of any particular language and platform, VisualGOP can be implemented as a library in familiar sequential languages and integrated with programming platforms such as clusters, CORBA, and the Web [13,14].

PERFORMANCE EVALUATION USING GOP

This section presents the evaluation results using VisualGOP as a high-level graph-oriented programming model [13]. Our preliminary experiments involve communications between processes in a parallel application. We first compare the performance of our proposed GOP application library with those of the MPI library, and then identify the overhead involved in the GOP system.

The experiments used a SGI Origin 2000 machine with 20 processors, running on IRIX64 6.5. The release of the IRIX implements the MPI 1.2 standard and all the testing programs are written in C. We used the FDM example for the evaluation. The program input is provided with problem sizes of 256×256 and 512×512.

Execution times were measured in seconds using the routine MPI Wtime. The timing measurements were made by inserting instructions to start and stop the timers in the program code.

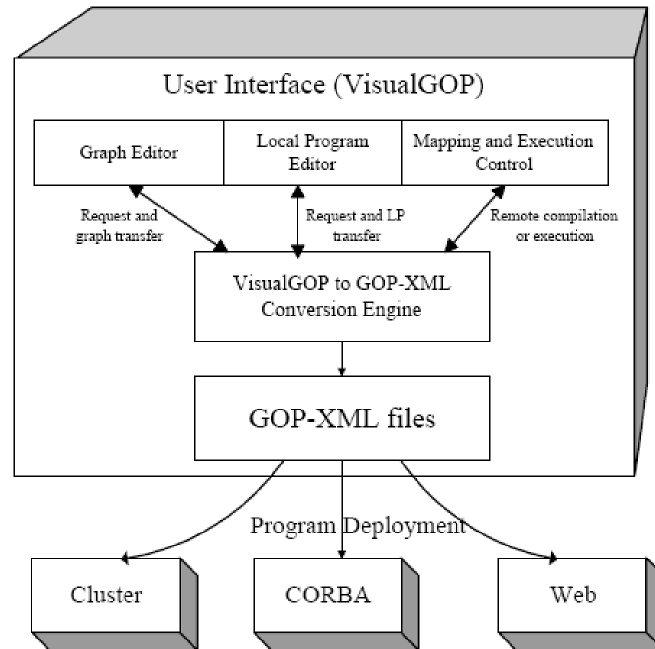

Figure 27. Program deployment process using GOP-XML.

To obtain more accurate results, we chose the lowest bound from 10 measurements. The major differences between the MPI and GOP implementations are that the former needs an extra routine to calculate the runtime processor ID for each node, while the latter resolves the node names into process IDs before invoking the communications based on MPI.

Figure 28 shows the execution times for the core-code of the FDM in the MPI and GOP systems, without considering other factors such as program initialization and finalization. The MPI program marginally outperforms GOP, as GOP needs to spend more time in manipulating the graph topology and preparing message data for communications. However, the difference is small and the speedups achieved by the MPI and GOP programs are almost the same, as depicted in Figure 29.

Figure 30 shows the program initialization time. Both GOP and MPI need to be set up before their library routines can be called. MPI includes an initialization routine MPI_INIT. GOP performs the same step as MPI during the graph initialization. In the graph initialization, each running node reads the graph structure and converts it into its programming structure, and then loads the graph into memory. We can see that the initialization time for both MPI and GOP increases linearly with the increasing number of processes (nodes). The difference between them is very small.

Copyright © 2005 John Wiley & Sons, Ltd.

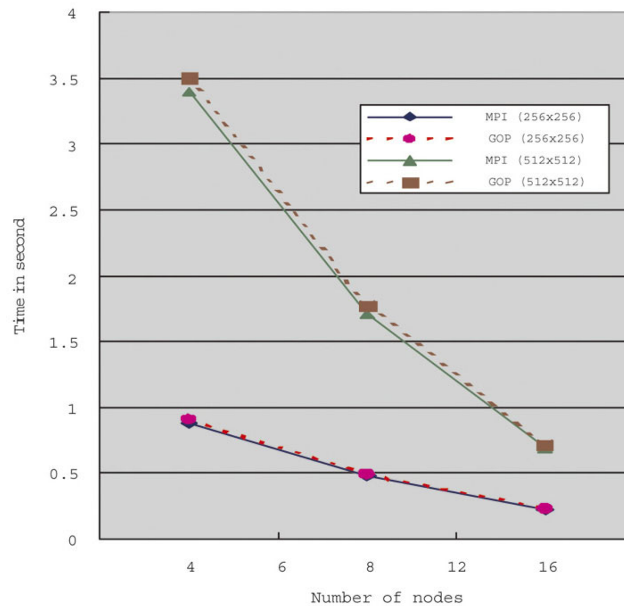

Figure 28. The total execution times of MPI and GOP implementations.

RELATED WORK

A parallel/distributed program is a collection of local programs distributed over various processors of a distributed network environment, running simultaneously, and communicating and synchronizing via message passing. Writing parallel/distributed programs overwhelms many programmers due to the difficulty of explicitly expressing communication and synchronization. Many efforts have been made to ease the programming tasks by providing high-level abstractions and programming tools. Support for visual programming of parallel and distributed systems is typically provided by graph-based programming environments. A program is defined as a directed graph where nodes denote computations and links denote communication and synchronization between nodes [15]. Directed-graph program representations are used to specify multiple threads of control, to display and expose parallel structure, and to express communication and synchronization, which are separated from the specification of sequential computations in individual local programs. In the past decade, a number of visual programming environments have been developed [2,16–22].

HeNCE [22,23] is an X-windows based software environment for developing parallel programs that run on a network of computers. To develop a HeNCE application, a programmer first expresses sequential computations in a standard language and then specifies how they are to be decomposed into a parallel program. Based on a parallel programming paradigm that describes an application program as a graph, HeNCE provides the programmer with a high-level abstraction for specifying parallelism.

Copyright © 2005 John Wiley & Sons, Ltd.

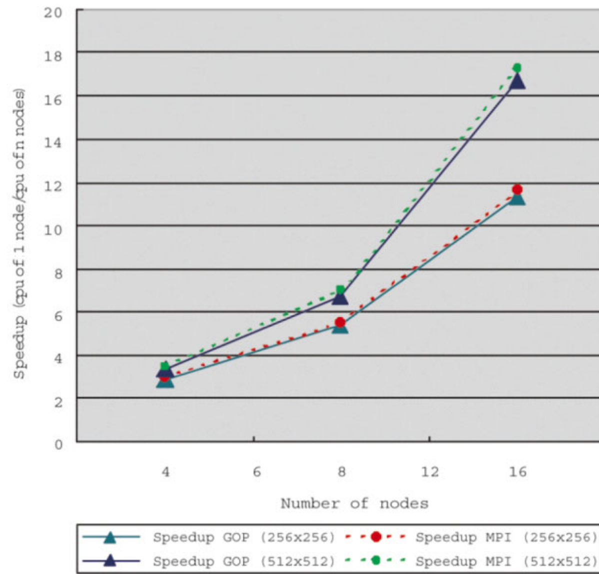

Figure 29. Speedups achieved by MPI and GOP programs.

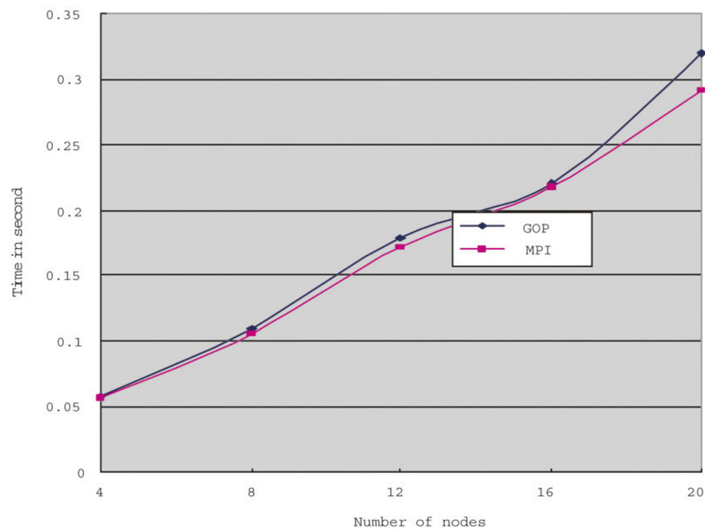

Figure 30. Graph initialization time.

Copyright © 2005 John Wiley & Sons, Ltd.

CODE [24,25] is a visual environment similar to HeNCE, as a graphical, re-targetable parallel programming system. The programmer writes a parallel program by drawing a graph which represents the relationships between various units of the program. The structure of the graph captures the major elements of a parallel program. The graph serves as a template for creating dynamic structures at runtime.

Both HeNCE and CODE facilitate a compositional approach to programming, in which dependencies are specified by means of arcs in a directed graph [1]. Sequential units of computation are composed to coordinate in forming a parallel program.

VDCE [21] is based on the dataflow programming paradigm. The software architecture consists of three parts: Application Editor, Application Scheduler, and VDCE Runtime System. VDCE provides an efficient mechanism to execute large-scale applications on distributed and diverse platforms.

Common to HeNCE, CODE and VDCE, nodes represent sequential computations in which communications with other nodes occur only at the beginning and end of a computation [15]. The dependencies are mainly based on data flow, so that computations have to be split into separate processes when communications occur. Overcoming this problem, Phred [26] attempts to combine data flow and control flow at the same level of abstraction as HeNCE and CODE. Neither of these formalisms provides sufficient information on the spatial distribution of processes.

By explicitly specifying processes with both dataflow and control flow, the Process Communication Graph (PCG) [27] used in the Visper distributed programming environment [28] adapts the Space–Time Diagram to visualize the design phase of message-passing programs. Its levels of abstraction range from groups of processes, processes, communication and synchronization, to sequential program blocks. The PCG environment, however, lacks the support for graph-based operations and validation capability.

In terms of the programming model and functionality, systems closer to VisualGOP include VERDI [20], the Software Architect's Assistant [19], and GRADE [17]. VERDI is a visual environment for Raddle [29]. It provides a visual language for describing the system control flow. VERDI has a graphical editor that allows a designer to represent the design graphically using icons. It can also execute a design using data that are either supplied by the programmer or generated internally. The designer can examine the functionality of the design during the execution through VERDI's animation feature or through the data produced, or even through a combination of both features.

The Software Architect's Assistant is a visual programming environment for the design and development of Regis [24]. Facilities provided include the integrated graphical and textual views, a flexible mechanism for recording design information and the automatic generation of program code and formatted reports from design diagrams. The Assistant focuses on program design and construction. There is no management of network resources such as processor mapping.

GRADE is a high-level, integrated programming environment for PVM-based program development. It provides a graphical user interface through which the programmer can access tools to construct, execute, debug, monitor, and visualize PVM parallel programs. GRADE also provides high-level graphical programming abstraction mechanisms to construct parallel applications. Like the Assistant, GRADE does not support the mapping of parallel processes to processors.

VisualGOP follows in many aspects the philosophy and techniques used in the existing systems. However, it provides several distinct features. First, existing systems are often specific to some programming languages and libraries, thus the program design and source code produced are only intended for specific platforms. VisualGOP, on the other hand, generates an XML-based standard

Copyright © 2005 John Wiley & Sons, Ltd.

representation of the logical graph and graph operations, and allows the LPs to be written in different languages and to execute on heterogeneous platforms. Second, VisualGOP provides visual instructions for GOP-based communication primitives and checks the consistency between the generated operations and the corresponding logical graph. Third, VisualGOP allows the programmer to visually define the mapping between local processes and graph nodes, and the mapping between graph nodes and the underlying physical processors.

CONCLUSIONS AND FUTURE WORK

Building a distributed application is not an easy task. The complexity and large scale of multiple processors in a heterogeneous network make the development process a difficult endeavor. This paper has described a novel graph-oriented approach to configuring and programming parallel/distributed software and a visual programming interface for the programmer to manage LPs and networked processing nodes.

VisualGOP supports a high-level program development, where the process structure is described using the GOP model. It provides integrated graphical tools for designing, mapping, compiling and executing parallel/distributed programs. With the aid of the Graph Editing Panel, the programmer designs the logical graph. The Coding Editing Panel allows the programmer to write LPs using different programming languages, and provides a set of high-level programming primitives for constructing parallel/distributed programs. It can also interact with the visual components and directly manipulates the textual source code visually. The Mapping Panel displays the information of visual components, and helps the programmer to specify the mapping of nodes to processors and the mapping of LPs to nodes visually. Furthermore, VisualGOP supports the automatic node-to-processor mapping and the XML-based graph representation.

Several future enhancements are planned. For example, a programmer should be able to copy and duplicate graph objects in the Graph Panel. When editing LPs, a well-defined graph grammar can be incorporated to provide verifiable graph structure and a systematic correspondence between the source code and the logical graph. Finally, VisualGOP could be extended and enhanced to be a monitoring and debugging tool for applications in a distributed network environment.

ACKNOWLEDGEMENTS

The authors would like to thank anonymous reviewers for their constructive comments and suggestions, which have helped to significantly improve the paper. This work is partially supported by the Hong Kong Polytechnic University under the Grant H-ZJ80.

REFERENCES

1. Browne JC, Hyder S, Dongarra JJ, Moore K, Newton P. Visual programming and debugging for parallel computing. *IEEE Parallel and Distributed Technology* 1995; **3**(1):75–83.
2. Wirtz G. Graph-based software construction for parallel message-passing programs. *Information and Software Technology* 1994; **37**(7):405–412.
3. Cao J, Fernando L, Zhang K. Programming distributed system based on graphs. *Intensional Programming I*, Orgun MA, Ashcroft EA (eds.). World Scientific: Singapore, 1996.

Copyright © 2005 John Wiley & Sons, Ltd.

4. Cao J, Fang L, Xie L, Chen DX, Zhang K. A formalism for graph-oriented distributed programming. *Software Visualization—From Theory to Practice*, Zhang K (ed.). Kluwer Academic: Dordrecht, 2003; 77–109.
5. Cao J, Liu Y, Xie L, Mao B, Zhang K. Portable runtime support for graph-oriented parallel and distributed programming. *Journal of Systems and Software* 2004; **72**:389–399.
6. Cao J, Fernando L, Zhang K. DIG: A graph-based construct for programming distributed systems. *Proceedings of the 2nd International Conference on High Performance Computing*, New Delhi, India, December 1995. McGraw-Hill, 1995.
7. Hui C, Chanson S. Allocating task interaction graphs to processors in heterogeneous networks. *IEEE Transactions on Parallel and Distributed Systems* 1997; **8**(9):908–925.
8. Senar M, Ripoll A, Cortes A, Luque E. Clustering and reassignment-based mapping strategy for message-passing architectures. *IPPS/SPDP 1998*, Orlando, FL, 30 March–3 April 1998. IEEE Computer Society Press: Los Alamitos, CA, 1998; 415–421.
9. Chan F, Cao J, Sun Y. Graph scaling: A technique for automating program construction and deployment in ClusterGOP. *The Fifth International Workshop on Advanced Parallel Processing Technologies (APPT03)* (*Lecture Notes in Computer Science*, vol. 2834). Springer: Berlin, 2003; 254–264.
10. W3C, Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/REC-xml.html [October 2000].
11. Herman I, Marshall MS. GraphXML—an XML-Based graph description format. *Proceedings of Graph Drawing 2000* (*Lecture Notes in Computer Science*, vol. 1984), Marks J (ed.). Springer: Berlin, 2001; 52–62.
12. Holt R, Winter A, Schürr A. GXL: Toward a standard exchange format. *Proceedings of the 7th Working Conference on Reverse Engineering*, Brisbane, Australia. IEEE Computer Society Press: Los Alamitos, CA, November 2000; 162–171.
13. Chan F, Cao J, Sun Y. High-level abstractions for message-passing parallel programming. *Parallel Computing* 2003; **29**(11–12):1589–1621.
14. Cao J, Ma X, Chan ATS, Lu J. WebGOP: A framework for architecting and programming dynamic distributed Web applications. *Proceedings of 2002 International Conference on Parallel Processing (ICPP'02)*, Vancouver, British Columbia, Canada, August 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002.
15. Browne JC, Dongarra JJ, Hyder SI, Moore K, Newton P. Visual programming and parallel computing. *Technical Report UT-CS-94-229*, University of Texas at Austin, 1994.
16. Cai W, Tan H-K, Turner SJ. Visual programming for parallel processing. *Software Visualization*, Eades P, Zhang K (eds.). World Scientific: Singapore, 1996; 119–140.
17. Kacsuk P, Dzsa G, Fadgyas T. A graphical programming environment for message passing programs. *Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97)*, Boston, MA, 1997; 210–219.
18. Newton P, Dongarra JJ. Overview of VPE: A visual environment for message passing. *Technical Report UT-CS-94-261*, Department of Computer Sciences, University of Tennessee, 1995.
19. Ng K, Kramer J, Magee J, Dulay N. A visual approach to distributed programming. *Tools and Environments for Parallel and Distributed Systems*, Zaky A, Lewis T (eds.). Kluwer Academic: Dordrecht, 1996; 7–31.
20. Shen VY, Richter C, Graf ML, Brumfield JA. VERDI: A visual environment for designing distributed systems. *Journal of Parallel and Distributed Computing* 1990; **8**(6):128–137.
21. Topcuglu H, Hariri S, Furmanski W, Valente J, Ra I, Kim D, Kim Y, Bing X, Ye B. The software architecture of a virtual distributed computing environment. *Proceedings of the High-Performance Distributed Computing Conference*. IEEE Computer Society Press: Los Alamitos, CA, 1997; 40–49.
22. Wolski R, Anglano C, Schopf J, Berman F. Developing heterogeneous application using zoom and HeNCE. *Proceedings of the Heterogeneous Workshop, IPPS 95*. IEEE Computer Society Press: Los Alamitos, CA, 1995.
23. Beguelin AL *et al.* HeNCE: Graphical development tools for network-based concurrent computing. *SHPCC-92 Proceedings of Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992. IEEE Computer Society Press: Los Alamitos, CA, 1992; 129–136.
24. Magee J, Dulay N, Kramer J. A constructive development environment for parallel and distributed programs. *Proceedings of the International Workshop on Configurable Distributed Systems*, Pittsburgh, PA, March 1994. IEEE Computer Society Press: Los Alamitos, CA, 1994.
25. Newton P. A graphical retargetable parallel programming environment and its efficient implementation. *PhD Thesis*, Department of Computer Science, University of Texas at Austin, 1993.
26. Beguelin AL, Nutt G. Visual parallel programming and determinacy: A language specification, an analysis technique, and a programming tool. *Journal of Parallel and Distributed Computing* 1990; **22**:235–250.
27. Stankovic N, Zhang K. Visual programming for message-passing systems. *International Journal of Software Engineering and Knowledge Engineering* 1999; **9**(4):397–423.
28. Stankovic N, Zhang K. A distributed parallel programming framework. *IEEE Transactions on Software Engineering* 2002; **28**(5):478–493.
29. Evangelist M, Shen VY, Forman I, Graf ML. Using raddle to design distributed systems. *Proceedings of the 10th International Conference on Software Engineering*. IEEE Computer Society Press: Los Alamitos, CA, 1988; 102–111.

Copyright © 2005 John Wiley & Sons, Ltd.