

Chapter 2

GOP: A GRAPH-ORIENTED PROGRAMMING MODEL FOR PARALLEL AND DISTRIBUTED SYSTEMS*

Jiannong Cao, Alvin T.S. Chan

Internet and Mobile Computing Lab, Department of Computing

The Hong Kong Polytechnic University, Hung Hom, Kowloon Hong Kong

csjcao@comp.polyu.edu.hk

Yudong Sun

School of Computing Science

University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU UK

yudong.sun@ncl.ac.uk

Abstract The advances of parallel and distributed computing demand high-level programming models that support efficient software development and execution. Graphs can effectively represent the logical structures of distributed systems and applications so as to facilitate the programming of distributed applications and support efficient mapping of programs to hardware architecture. This chapter presents a Graph-Oriented Programming (GOP) model that provides flexible graph constructs and graph-oriented primitives to build a programming paradigm based on graph topology and also provides a formal specification of software architecture for distributed programs. The GOP model creates an abstract programming framework and supports dynamic reconfiguration of distributed computing system to implement adaptive computation and fault-tolerance. Various computing environments have been developed based on GOP for cluster computing, web service, and component-based computation.

Keywords: Graph-oriented programming, distributed computing, software architecture

*This work is partially supported by Hong Kong Polytechnic University under HK PolyU Research Grant H-ZJ80.

1. Introduction

Parallel and distributed systems have been providing a pervasive computing platform, along with the significant advances in network and Internet technologies, for developing and executing various applications. The application areas span from large-scale scientific computing to web service and E-commerce. The programming methodologies for parallel and distributed computing are still an active research area. Due to the high diversity in application areas and system architectures, the modeling of parallel and distributed applications is not a plain deed.

At present, parallel and distributed programming mainly employs the computing paradigms such as message-passing, data-parallel, divide-and-conquer, and master-slave models. These paradigms have the limitations in representing the structuring characteristics of parallel and distributed applications and systems. Therefore, high-level programming model is required, which demand high flexibility to describe distinctive features of different application requirements on the model. The model should be able to describe the logical structure of different applications in identical way so that a uniform programming methodology can be created. It should be scalable to represent computations on clusters and on wide-area systems. It should be adaptive to the dynamic evolution of the computational pattern of an application and the architectural configuration of the underlying system.

A parallel and distributed application is composed of a collection of functional components called *tasks* that can be executed concurrently, possibly on different machines, with necessary interaction and cooperation. Graph is ideally suited to represent such a logical program structure. The nodes represent the tasks and the edges denote the interactions between the tasks. The graph structure is flexible to represent different computation and communication patterns. A graph can be scalable to the size of a program and an underlying system. A graph is pliable to make dynamic modification to reflect the evolution of the computational requirement of a program and the architectural evolution of an underlying system. Different program code can be bound to the nodes to build a MPMD program. Attributes can be assigned to the nodes and edges to represent the features of a program and the performance of system resources. The graphs can be used in an abstract framework to define supportive services in parallel and distributed computing, such as task naming and grouping, communication and coordination, load balancing, and fault tolerance. The graph-oriented programming model can conform to object-oriented programming model and provide a powerful support to the development of graph constructs, primitives, and programs.

This chapter presents a *Graph-Oriented Programming model*, called *GOP*, for developing parallel and distributed programs. The *GOP* model provides a high-level abstraction by which a parallel / distributed program is depicted as a *logical graph*. In the graph, the nodes represent the computational tasks

and the edges represent the communication and synchronization between the tasks. The operations performed by the tasks are defined as *local programs* that are bound to the nodes. Therefore, a distributed program can be specified. The GOP model can also specify a task-to-processor mapping to execute the program. GOP specifies the graph constructs for user to build a program. It provides a library of primitives to be called by local programs for varied operations based on the graph structure.

The GOP model has also provided a formal specification method for the software architecture of parallel and distributed software. It is a versatile model for various computing environments such as cluster computing, web-based applications, and component-based computation.

The rest of the chapter is organized as follows. Section 2 discusses the related work. Section 3 specifies the GOP model and its features. Section 4 introduces the computing frameworks based on the GOP model. Section 5 gives the conclusions and future work.

2. Related Work

Graph-based programming has been a prosperous approach of parallel computing for decade. Graphical programming languages, libraries, and environments have been developed as visual programming tools to ease parallel programming and assist the software development on parallel systems.

CODE [1, 15] is a graphical parallel programming language in which a user can create a parallel program by drawing a *dataflow* graph that shows the communication structure of the program. The graph consists of nodes to represent computations (or shared variables) and arcs to represent the data flow. HeNCE [3] is a graphical language for creating and running parallel programs over a heterogeneous collection of computers. Differing from CODE, the graph in HeNCE shows the *control flow* of a program [16]. PYRROS [26, 27] is a compile-time scheduling and code generation tool for parallel program on distributed-memory architecture. It provides a task graph language for creating the task graph for a program, editing the associated C code, specifying the weights of computation and communication operations, and the maximum number of available processors. VPE [17] is a visual parallel programming environment that provides a simple GUI for creating message-passing programs and supports automatic compilation, execution, and animation of the programs.

Some graphical programming tools provide pre-defined computational tasks or program structures for specific applications. VDCE [23, 24] is a software development environment that provides task libraries for building large-scale applications on the network of heterogeneous computers at geographically distributed sites. CAPSE environment [12] provides tools for performance prediction in the development of parallel programs based on the graphical creation and editing of the scalable workload characterizations of MIMD algorithms.

Some graphical programming environments aim to support code reuse. Tracs [2] is a graphical programming environment that promotes a modular approach for application development across heterogeneous machines on a local network. P-RIO [13] is a modular parallel programming environment that provides an object-based software construction methodology with modularity and code reuse.

Finally, there are visual programming environments that adopt object-oriented technology. Prograph [20, 22] is an object-oriented visual programming language and a development environment on Macintosh platforms. Visper [21] is a distributed object-oriented environment that supports visual programming development, object (process) groups, and agent-based system management.

The projects discussed above are graph-based programming languages and environments in which graph is simply used as a visual representation of a program structure. The code, especially the inter-node communication and synchronization, is programmed in the convention of procedural languages (e.g., C, FORTRAN) and message passing libraries (e.g., MPI, PVM, SOAP). A major shortcoming of these existing works is that the design model is not mapped to the implementation model, thus there is a big gap between the design and implementation support. Differently, our GOP model harnesses *graph-oriented* concept. A graph is not only used as the representation of program structure but also specifies the operations of the program based on the topology of a graph. For example, the communication primitives can be defined using the relative references of source and destination nodes in the graph, such as precedent and successor, parent and children, root and leaves, instead of using node IDs. The graph-oriented approach creates a flexible high-level programming model which allows a program adaptive to the parameters of the program and the underlying system such as problem size and number of processors. As an abstract representation of software, the GOP model also provides a framework for the specification of software architecture. It can also support dynamic re-configuration to match the evolution of computational requirements and system resources. With an object-orientation extension, this can be achieved through reflection.

3. GOP Model

GOP (Graph-Oriented Programming) is a graphical programming model for parallel and distributed programming. It specifies the graph constructs and primitives to develop different programs. With the graph-oriented concept, GOP provides a high-level abstraction of program structure that is appropriate to be defined and implemented in object-oriented method and other programming paradigms.

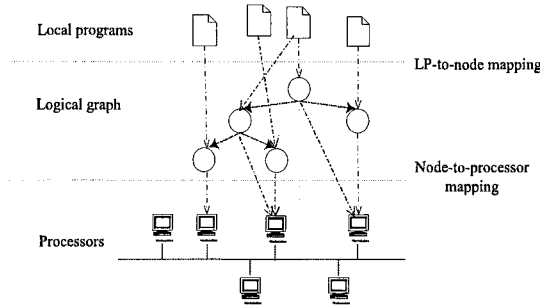


Figure 2.1. The GOP model

3.1 Graph Specification

In GOP, a program is defined as a *logical graph*, $G(N, E)$, where N is a set of *nodes* and E is a set of *edges*. The nodes represent the computational tasks of the program. Each edge links a pair of nodes, denoting the relationship between them. The relationship can be execution precedence, data dependency, and communication requirement. The graph represents the logical structure of the program. The edges can be *directed* to denote unidirectional data flow or control flow. Otherwise, an edge without direction allows bi-directional interaction between the nodes.

Associated with the nodes is a collection of local programs (LPs). Each node is bound with an LP that provides the program code to be executed by the node. The communication operations specified on the edges are implemented in the LPs by calling correspondent message-passing primitives. The nodes of a graph can be allocated to multiple processors and be executed concurrently. The GOP model can specify the node-to-processor mapping. In general, the GOP model consists of the following components.

(1) *Logical graph*: a logical graph (directed or undirected) defines the structure of a program in which the nodes represent the computational tasks and the edges specify the relationships between the nodes.

(2) *Local programs*: a collection of local programs (LPs) specifies the operations performed by the nodes.

(3) *LP-to-node mapping*: the mapping binds the LPs to the nodes.

(4) *Node-to-processor mapping* (optional): the mapping allocates the nodes to the processors to execution. When a program does not explicitly specify this mapping, the runtime system will use a default mapping strategy.

Figure 2.1 manifests the construction of the GOP model. In the GOP-based programming, a programmer first constructs a logical graph to describe the abstract structure of a program and writes associated logical programs. The LPs can be written in any languages supported by the implementation system. The LPs also invoke the pre-defined primitives to perform the communication and

coordination operations specified by the edges. The LPs are bound to the nodes by a LP-to-node mapping. The node-to-processor mapping is an optional component that can be specified before submitting the program to execution. The constructs of a GOP-based program can be specified as following:

(1) Logical graph

Let LGraph be the class of logical graph. *Graph-name* is the identifier of a logical graph. *Node_no* is the ID of a node.

$\langle \text{Logical-graph} \rangle ::= \text{LGraph } \text{Graph-name} = \{ \{ \langle \text{set-of-nodes} \rangle \}, \{ \langle \text{list-of-edges} \rangle \} \}$.

$\langle \text{set-of-nodes} \rangle ::= \langle \text{range-of-nodes} \rangle | \langle \text{node-list} \rangle$.

$\langle \text{range-of-nodes} \rangle ::= \langle \text{node_no} \rangle .. \langle \text{node_no} \rangle$.

$\langle \text{node-list} \rangle ::= \langle \text{node_list} \rangle, \langle \text{node_no} \rangle | \langle \text{node_no} \rangle$.

$\langle \text{list-of-edges} \rangle ::= \langle \text{list-of-edges} \rangle, \{ \text{node_no}, \text{node_no} \} | \{ \text{node_no}, \text{node_no} \} | \epsilon$.

$\langle \text{node_no} \rangle ::= \{ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \}$.

(2) Local programs

$\langle \text{Local-program} \rangle ::= \text{code of a programming language and a library of primitives}$.

(3) LP-to-node mapping

Let LMap be the class of LP-to-node mapping. *Lmap-name* is the identifier of an LP-to-node mapping. *LP_id* is the name of a local program. An LP-to-node mapping is defined as a set of $\{ \text{node_no}, \text{LP_id} \}$ pairs that binds the node *node_no* with the local program *LP_id*.

$\langle \text{LP-to-node-mapping} \rangle ::= \text{LMap } \text{Lmap-name} = \{ \langle \text{node-lp-pair} \rangle \}$.

$\langle \text{node-lp-pair} \rangle ::= \langle \text{node-lp-pair} \rangle, \{ \langle \text{node_no} \rangle, \langle \text{LP_id} \rangle \} | \{ \langle \text{node_no} \rangle, \langle \text{LP_id} \rangle \} | \epsilon$.

$\langle \text{LP_id} \rangle ::= \{ \text{a-z} | \text{A-Z} | _ | 0-9 \}$.

(4) Node-to-processor mapping

Let PMap be the class of node-to-processor mapping. *Nmap-name* is the identifier of a node-to-processor mapping. *Processor_id* is the ID of a processor. A node-to-processor mapping is specified as a set of $\{ \text{node_no}, \text{processor_id} \}$ pairs by which *node_no* is mapped to *processor_id*. The node-to-processor mapping is an optional construct. If omitted, a default mapping will be used.

$\langle \text{Node-to-processor-mapping} \rangle ::= \text{PMap } \text{Nmap-name} = \{ \langle \text{node-processor-pair} \rangle \}$.

$\langle \text{node-processor-pair} \rangle ::= \langle \text{node-processor-pair} \rangle, \{ \langle \text{node_no} \rangle, \langle \text{processor_id} \rangle \} | \{ \langle \text{node_no} \rangle, \langle \text{processor_id} \rangle \} | \epsilon$.

$\langle \text{processor_id} \rangle ::= \text{system-dependent id}$.

The GOP model is independent from any language and platform. It can be implemented on different hardware and software platforms such as clusters and distributed systems on top of PVM, MPI, and CORBA. The local programs can

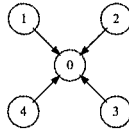


Figure 2.2. Logical graph of a master-slave algorithm

be written in different languages (such as Java, C, and C++), depending on the implementation.

The GOP model builds a high-level framework for software architecture [10, 11] which supports the architectural definition of parallel and distributed software. The GOP modularizes a program as a graph. With regard to the non-intuitive structures of parallel and distributed programs, GOP enables programmer to visually describe the abstract structure and specify the semantic context of a program such as data dependency, task precedence, and synchronization among concurrent tasks. The GOP model establishes a foundation to the architectural design of parallel and distributed programs, which is helpful to improve the efficiency of software development. The model can enhance the understandability of the complex logical structures of parallel and distributed programs.

As the support for software architecture, the GOP model is convenient to define the abstract structure of a family of software system. Using GOP, for example, we can depict the generic structure of a master-slave algorithm. Figure 2.2 shows a logical graph of master-slave computation with one master and four slaves. Node 0 is the master that runs the local program *Master*. It is the coordinator of entire computation. Other nodes are slaves running the local program *Slave* to perform certain computation. They are under the control of the master.

With the GOP graph constructs, the master-slave program can be defined as:

- (1) Logical graph: $LGraph\ msgraph = \{\{0..4\}, \{\{0,1\}, \{0,2\}, \{0,3\}, \{0,4\}\}\};$
- (2) LP-to-node mapping: $LMap\ lmsmap = \{\{0, "Master"\}, \{1, "Slave"\}, \{2, "Slave"\}, \{3, "Slave"\}\};$
- (3) Node-to-processor mapping: $PMap\ pmsmap = \{\{0, "host0"\}, \{1, "host1"\}, \{2, "host2"\}, \{3, "host3"\}\};$ where *host0* to *host3* are the names of the hosts to run the program.

The master-slave program can implement any computation specified by the local programs. For example, the programs can implement client-server computing such as file server and web server. The programs can also accomplish parallel computing where the master acts as the task allocator and the slaves run the tasks in parallel. Hence, the GOP model provides an abstract representation for a family of programs that have an identical structure.

3.2 Graph-oriented Operations

In the GOP model, a library of graph-oriented programming primitives can be defined based upon the semantics of graph. The LPs can call these primitives to implement various operations in the graph-oriented programming. Users can also define custom primitives in the similar graph-oriented semantics. The graph-oriented primitives can be basically classified into four categories:

- *Communication and synchronization*

These primitives support communication operations for passing messages from one node to others, such as unicast, multicast, and broadcast. The LPs call the primitives to fulfill the communications associated with the edges and to synchronize the operations of the nodes.

- *Subgraph derivation*

The primitives aim to derive subgraphs, such as the shortest path and spanning tree, of a graph. Many distributed algorithms include the construction of some form of subgraph deriving from an original graph to obtain optimal solution. The primitives are useful to these algorithms.

- *Query*

The primitives examine the attributes of a graph such as the number of nodes, the current binding of an LP to a node and whether an edge exists between two nodes. The query provides a basis for the operations of system control and reconfiguration.

- *Graph update*

The primitives support dynamic reconfiguration of a graph. For example, there are primitives to insert or delete nodes and edges in a graph. Also, the LP-to-node mapping and node-to-processor mapping can be dynamically altered at runtime. The dynamic reconfiguration of a program graph reflects the adaptation of distributed computing in response to varying computational requirements and available resources.

In the graph-oriented primitives, the references to nodes and edges are generally based on *relative naming*. The primitives do not explicitly indicate the nodes and edges involved. Instead, the relative positions of the nodes and edges are specified such as *precedents*, *successors* and *neighbors*. For example, in Figure 2.2, the “Precedents” of the master are all slave nodes and the “Successor” of all slaves is the master node. In a GOP-based program, new references can be derived by carrying out *set operations* on existing references. For example, the *neighbors* of a node are the union of its precedents and successors.

A collection of message passing primitives including *unicast*, *multicast* and *anycast* can be called in the LPs for inter-node communication. In Figure 2.2, the master can multicast a message to the slaves by the multicast primitive `Msend()` using the reference “Precedents”:

```
Msend(Precedents, message);
```

Each slave node can receive the message by the *anycast* primitive `Arecv()` that receives a message from any of the nodes specified by the reference “Successor”, which refers to the master in this example:

```
Arecv(Successor, message);
```

The interaction between the master and the slaves can be accomplished by a composite of communication primitives. For example, the master can send a message to query all slaves and wait for the responses using `Msend()` and `Mrecv()` primitives:

```
Msend(Precedents, query_command);
Mrecv(Successors, result_buffer);
```

On the other side, each slave receives the query and sends the result back to the master as:

```
Arecv (Successor, command);
if (command == query) {
    process the query;
    Asend (Successor, result);
}
```

The class of logical graph *LGraph* specified in Section 3.1 provides a general specification of a graph structure. It can be used to describe any type of graphs. However, an algorithm is usually designed on a specific graph topology, e.g., tree, hypercube, or mesh. The algorithm needs to operate with specific semantics and constraints on a particular graph topology. To support the topology-specific operations, the GOP model allows programmers to derive new graph type, such as a tree or a star, from the basic *LGraph* along with new primitives based on topology-specific semantics and operations. For a tree, for example, there is one root and each node has one ancestor and one or more descendants. The primitives for tree-specific operations can be defined with the references as *parent*, *children*, *siblings*, *root*, and *leaves*.

An example of graph derivation is a new graph type created for the master-slave computation shown in Figure 2.2. The new type of graph is called *Star* that is derived from the class of *LGraph*:

```
<Star-graph> ::= Star Star-name '=' '{' <Logical-graph>, '{' <center-
node>'}', '{' <set-of-leaves>'}' '}' .
```

```
<center-node> ::= <node_no> . // master node
```

```
<set-of-leaves> ::= <node-list> . // all slave nodes
```

where *<Logical-graph>*, *<node_no>*, and *<node-list>* are specified in Section 3.1. This new type of graph constrains the topology as a star that consists

of one center node and a number of leaf nodes. Every leaf is connected to the center node.

With the type of star, the primitives can be redefined to directly implement intuitive, star-specific operations. To perform the query operations as above, the master at the center can make use of the star-specific primitives where the reference to the slaves is omitted because it is implied in the primitives:

```
SendToLeaves(query_command);
RecvFromLeaves(result_buffer);
```

The slave nodes receive the query command and send the result back to the master using the primitives as:

```
RecvFromCenter(command);
SendToCenter(result);
```

3.3 Dynamic Reconfiguration

As a framework for software architecture, the GOP model also manifests the dimensions in which the software can evolve. Distributed applications demand the capability of dynamic reconfiguration to adapt to the evolving computational requirements and execution environment. GOP facilitates the dynamic reconfiguration of a software system with the support of graph-oriented operations.

Traditionally, the configuration of a parallel / distributed software contains: (a) a set of software components; (b) the interconnections between the components that specify the interactions between the components; (c) the mapping of the components to target hosts [5]. The configuration is closely related to the concept of software architecture. Dynamically reconfigurable software system is sometimes called a system with *dynamic architecture* [18].

Dynamic reconfiguration is needed in many circumstances of distributed computing. In distributed applications, the workload is often dynamically generated in individual components. The applications require dynamic transformation in its structure to handle the change of workload distribution. New nodes can be added to a graph to share the increasing workload. A node can also be removed from a graph when the node fails to work.

The characteristics of the GOP model determine the feasibility for dynamic reconfiguration. In GOP, local programs (LPs) are separated from a graph topology. The LPs may not share any information except the structure of a logical graph. The LPs need no direct reference to each other for interaction. The LPs bound to individual nodes communicate with one another through relative naming. This feature allows the modification to a graph meanwhile keeping the compatibility of the LPs on the altered graph. The GOP model specifies the query primitives to examine the structure of a graph by which the proper reconfiguration can be determined. Furthermore, the scope of valid reconfiguration can be constrained within a specific graph topology as the valid primitives of graph update are specified based on the graph topology. The

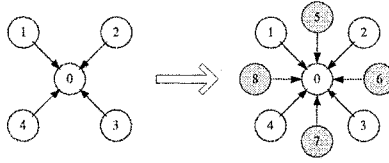


Figure 2.3. Add new leaves to the graph

reconfiguration of a graph can be defined at a high level in terms of logical graph, making it easier to understand and manage.

With an object-oriented extension, the GOP model can support dynamic reconfiguration through inheritance and reflection. As an example, consider the dynamic reconfiguration of the master-slave graph shown in Figure 2.2. The master can dynamically add new slaves to or remove existing slaves from the graph according to the runtime workload. Figure 2.3 shows that four slaves are added to the graph. A new slave can be added to the graph using the primitives of graph update as following:

```

Node newLeaf = new Node(node_no); // create new leaf
Edge newEdge = new Edge(newLeaf, center_node); // create new
edge
AddNode(graph-name, newLeaf); // add new leaf to graph
AddEdge(graph-name, newEdge); // add new edge to graph

```

The dynamic reconfiguration in Figure 2.3 preserves the star topology. Nevertheless, dynamic reconfiguration can also create a new graph topology that is different from the original one. The GOP model implements such a dynamic reconfiguration by deriving a new graph type that extends the original one. The primitives of the new graph type can be invoked to transform the original graph to the new topology. Let us use the graph derivation to reconfigure the graph on the right side of Figure 2.3. Assume that the master is finally overloaded as more and more slave nodes are added to the system, a new master node will be added into the graph. To achieve this goal, a new type of graph called *StarMC* is defined which extends the star topology to permit the coexistence of more than one center. Figure 2.4 shows such a reconfiguration that adds an additional master to the original star. In the multi-center star, each slave is linked to each of the masters. This reconfiguration can be implemented by the following primitives:

```

Node newCenter = new Node(node_no); // create a new center
AddNode (graph-name, newCenter); // add new center to graph
for(each leaf in the graph) {
    Edge newEdge = new Edge(leaf, newCenter); // create a new
edge between each leaf and the new center.
    AddEdge = (graph-name, newEdge); // add the edge to the graph
}

```

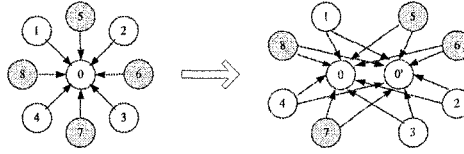


Figure 2.4. Add a new center to a star

The behavior of the master-slave program with multiple centers should be adjusted accordingly. In *StarMC*, some primitives should be redefined. For example, the communication primitive `SendToCenter()` should be redefined for the leaves to send a message to one of the centers to balance the workload between the masters. The primitive of adding a new leaf should be redefined to link the new leaf to each of the centers. On the other hand, the original local programs are still usable on the new topology because the graph-oriented operations will be automatically adjusted based on the adaptive implementation of the primitives in the *StarMC*.

4. Applications of GOP

We have developed different programming environments to implement the GOP model for cluster computing, web-based computation, and component-based computation.

4.1 ClusterGOP

ClusterGOP [8] is a software environment to implement the GOP model for cluster computing. It provides a visual user interface and a framework for developing parallel and distributed programs on cluster systems. The user interface is a visual programming environment. When building an application, a programmer starts to draw a logical graph in the graph design editor. Then, the programmer uses the text editor to write the LP code and binds to the nodes of the logical graph. After that, the whole application is ready for compilation and execution. To run the program, the graph nodes will be mapped to the processors in the cluster according to the user-specified node-to-processor mapping or a default mapping strategy.

The ClusterGOP environment provides a library of primitives for communication, synchronization, query, and graph update. The LPs can call these primitives. The communication and synchronization primitives closely conform to the MPI (Message Passing Interface) standard [14] and are implemented with the MPI functions.

When compiling the application, the ClusterGOP environment converts the logical graph into XML (Extensible Markup Language) format [25]. Then, ClusterGOP transfers the necessary data (including the XML graph, LPs, and ClusterGOP library) to target machines and compiles the application. After

that, ClusterGOP starts to execute the application. Each target machine contains two runtimes to run the programs. One is the ClusterGOP runtime, which is a background process to support the query, update and consistency of a graph structure. When updating a graph, the runtime will block other machines from updating the graph and synchronize the graph update made on different machines. The other is the MPI runtime, which implements the ClusterGOP communication and synchronization primitives. ClusterGOP uses the MPI library as the low-level implementation of inter-process communication.

4.2 WebGOP

WebGOP [6] is a framework for constructing web-based distributed applications. WebGOP uses object-oriented method to support software architecture. It specifies the architecture of a distributed computing system with the graph objects that are separated from the programming of functional components. The object-oriented method also benefits the reusability of the functional components and the software architecture. With the graph objects and the built-in graph-update facilities, dynamic reconfiguration of an application can be implemented.

The WebGOP framework defines the graph constructs as the foundation to build the GOP-based web applications. The framework consists of a WebGOP runtime, a monitoring and management module, and a security protection module. The core of the WebGOP runtime is the distributed representation and management of a graph on which a set of graph-oriented message passing primitives and a set of basic graph update primitives are provided as the APIs for programming web applications. The WebGOP runtime translates the graph nodes and nodes groups into web addresses. Several graph derivation operations such as the shortest path, minimum spanning tree are built in the runtime.

The monitoring and management model is responsible for helping user to manage, debug and monitor the applications. The module also acts as the front-end user interface for application loading, deployment, activation, and dynamic reconfiguration.

Reliable and flexible security protection is essential to the web applications. As an open system running over different administrative domains, WebGOP should control user access to the resources by discerning the permissions. The security protection module is provided for this purpose based on cryptic communication and digital signature identification.

A prototype of WebGOP is developed in a heterogeneous network environment consisting of a Sun Ultra Enterprise server 10000, several Sun Solaris workstations and Microsoft Windows 2000 workstations. The prototype is implemented with Java and Apache SOAP (Simple Object Access Protocol) [4]. Since SOAP has become *de facto* standard protocol for web services, WebGOP uses it to realize the compatibility in the web environment. The prototype pro-

vides a communication middleware to support the interactions between the components of distributed web applications and a graph-oriented framework for architectural modeling and programming. Sample applications have been developed on the WebGOP framework to evaluate the performance of the prototype. For example, a master-slave system is constructed for web service and the dynamic reconfigurations shown in Figure 2.3 and Figure 2.4 are implemented.

4.3 ComponentGOP

ComponentGOP [7] provides a new approach to support the creation and reconfiguration of distributed component-based software (CBS). Distributed software can be viewed as a collection of building blocks (called *components*) that interact with each other through message passing. ComponentGOP implements the GOP model in component-based computing based on a generic middleware such as CORBA [9]. ComponentGOP creates a graph-oriented framework for modeling and constructing component-based software. The architectural design of CBS can be simplified to a higher level with graph abstraction and pre-defined graph patterns. It also acts as the communication middleware for distributed components.

The ComponentGOP framework consists of a configuration manager module, a consistency maintenance module, and a runtime module. The ComponentGOP runtime includes the APIs that provides a set of graph-oriented primitives for programming component-based software and the ComponentGOP LIB for compiling and executing the applications. The kernel of the runtime is the distributed representation and management of graphs. Based on the kernel, a set of graph-oriented primitives are created for message passing, graph update, and query. The configuration manager module acts as the front-end user interface for component loading, deployment, activation, and dynamic reconfiguration management. When a graph needs dynamic reconfiguration, software consistency should be maintained among distributed components. The consistency maintenance module is used for this purpose.

A prototype of ComponentGOP has been implemented on top of CORBA using Java and VisiBroker (a CORBA compliant platform) in a heterogeneous network environment with Sun Solaris workstations and Windows 2000 workstations. In the prototype, the ComponentGOP LIB is divided into two sublibs: GOPORB.LIB and GOP.LIB. The GOPORB.LIB mainly implements the communication-related primitives that are mapped to the CORBA method invocations. The implementation of GOPORB.LIB is based on CORBA DII (Dynamic Invocation Interface), DSI (Dynamic Skeleton Interface) and CORBA compiled client stubs and server skeletons. The GOP.LIB implements other primitives such as graph update and query. ComponentGOP encapsulates the complicated CORBA programming details in abstract, graph-oriented components so that the component-based programming can be simplified.

5. Conclusions

In this chapter, we have presented the GOP model for graph-oriented programming. The GOP model provides a new framework for constructing the logical structure of a distributed program in terms of graph. The operations of the program are specified on the graph structure. The model presents a high-level abstraction of program logic to support the formal specification of software architecture. Distributed programs built on the GOP model are suited to undertake dynamic reconfiguration at runtime in response to the evolution of computational requirements and underlying systems. New computing frameworks and middleware can be developed based on GOP to implement graph-oriented programming in different parallel and distributed environments. We have implemented the graph-oriented computing environments for cluster computing, web-based computation, and component-based computation.

We will design a formal specification of the graph-oriented model. We believe that the graph grammar based formalism is a suitable approach for the formal specification. For the prototype implementation, we will enrich the graph-oriented programming tools including graph editor, task scheduler, and program visualization. We are investigating the integration of the GOP-based environments such as ClusterGOP, WebGOP, and ComponentGOP under a unified interface that can be used for application programming in different systems.

References

- [1] D. Banerjee and J. Browne, "Complete Parallelization of Computations: Integration of Data Partitioning and Functional Parallelism for Dynamic Data Structures", *Proc. 10th IEEE Int'l Parallel Processing Symp. (IPPS'96)*, Honolulu, Hawaii, April 15-19, 1996, pp.354-361.
- [2] A. Bartoli, P. Corsini, G. Dini and C. Prete, "Graphical Design of Distributed Applications Through Reusable Components", *IEEE Concurrency*, Vol. 3, No. 1, Spring 1995, pp.37-50.
- [3] A. Beguelin, J. Dongarra, A. Geist, R. Manchek and K. Moore, "HeNCE: A Heterogeneous Network Computing Environment", *Scientific Programming*, Vol. 3, No. 1, 1994, pp.49-60.
- [4] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte and D. Winer, "Simple Object Access Protocol (SOAP) 1.1", W3C Note, May 8, 2000, available at <http://www.w3.org/TR/SOAP/>
- [5] J. Cao, A. Chan, C. Lee and K. Yu, "A Dynamic Reconfiguration Manager for Graph-Oriented Distributed Programs", *Proc. 1997 Int'l Conf. on Parallel and Distributed Systems (ICPADS'97)*, Seoul, Korea, Dec 1997, pp.216-221.
- [6] J. Cao, X. Ma, A. Chan and J. Lu, "WEBGOP: A Framework for Architecting and Programming Dynamic Distributed Web Applications", *Proc. 2002 Int'l Conf. on Parallel Processing (ICPP'02)*, Vancouver, Canada, August 2002, pp.266-275.
- [7] J. Cao, M. Cao and A. Chan, "Architecture Level Support for Dynamic Reconfiguration and Fault Tolerance in Component-Based Distributed Software", *Proc. 2002 Int'l Conf. on Parallel and Distributed Systems (ICPADS'02)*, Dec. 2002, Taiwan, pp.251-256.
- [8] F. Chan, J. Cao and Y. Sun, "High-level Abstractions for Message-passing Parallel Programming", *Parallel Computing*, Vol.29, No.11-12, 2003, pp.1589-1621.

- [9] CORBA, <http://www.corba.org/>
- [10] D. Garlan and M. Shaw, "An Introduction to Software Architecture", in *Advances in Software Engineering and Knowledge Engineering*, Vol. II, World Scientific Publishing, 1993.
- [11] D. Garlan and D. Perry, "Software Architecture: Practice, Potential, and Pitfalls", *Proc. 16th Int'l Conf. on Software Engineering*, Sorrento, Italy, May 16-21, 1994, pp.363-364.
- [12] B. Gruber, G. Haring, J. Volkert and D. Kranzlmüller, "Parallel Programming with CAPSE - A Case Study", *Proc. 4th EUROMICRO Workshop on Parallel and Distributed Processing (PDP'96)*, Braga, Portugal, Jan. 1996, pp.130-137.
- [13] O. Loques and J. Leite, "P-RIO: A Modular Parallel-Programming Environment", *IEEE Concurrency*, Vol. 6, No. 1, Jan-Mar 1998, pp.47-57.
- [14] The Message Passing Interface (MPI) Standard, <http://www-unix.mcs.anl.gov/mpi/>
- [15] P. Newton and J. Browne, "The CODE 2.0 Graphical Parallel Programming Language", *Proc. ACM Int'l Conf. on Supercomputing (Supercomputing'92)*, Washington D.C., July 1992, pp.167-177.
- [16] P. Newton, "Visual Programming and Parallel Computing", *Workshop on Environments and Tools for Parallel Scientific Computing*, Walland, TN, May 26-27, 1994.
- [17] P. Newton and J. Dongarra, "Overview of VPE: A Visual Environment for Message-Passing", *Proc. 4th Heterogeneous Computing Workshop*, Santa Barbara, CA, April 25, 1995.
- [18] P. Oreizy and R.Taylor, "On the Role of Software Architecture in Runtime System Re-configuration", *IEE Proceedings-Software Engineering*, Vol. 145, No. 5, October 1998, pp.137-145.
- [19] PVM, <http://www.csm.ornl.gov/pvm/>
- [20] T. Smedley and P. Cox, "Visual Languages for the Design and Development of Structured Objects", *Journal of Visual Languages and Computing*, Vol. 8, No. 1, 1997, pp.57-84.
- [21] N. Stankovic and K. Zhang, "A Distributed Parallel Programming Framework", *IEEE Transactions on Software Engineering*, Vol. 28, No. 5, May, 2002, pp.478-493.
- [22] S. Steinman and K. Carver, *Visual Programming with Prograph CPX*, Manning Publications, 1995.
- [23] H. Topcuoglu, S. Hariri, W. Furmanski, J. Valente, I. Ra, D. Kim, Y. Kim, X. Bing and B. Ye, "The Software Architecture of a Virtual Distributed Computing Environment", *Proc. 6th Int'l Symp. on High Performance Distributed Computing (HPDC'97)*, Portland, OR, Aug. 5-8, 1997, pp.40-49.
- [24] H. Topcuoglu, S. Hariri, D. Kim, Y. Kim, X. Bing, B. Ye, I. Ra and J. Valente, "The Design and Evaluation of a Virtual Distributed Computing Environment", *Cluster Computing*, Vol. 1, No. 1, 1998, pp.81-93.
- [25] XML, <http://www.xml.org/>
- [26] T. Yang and A. Gerasoulis, "PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors", *Proc. ACM Int'l Conf. on Supercomputing (Supercomputing'92)*, Washington D.C., July 1992, pp.428-437.
- [27] T. Yang and A. Gerasoulis, "A Parallel Programming Tool for Scheduling on Distributed Memory Multiprocessors", *Proc. the Scalable High Performance Computing Conference*, Williamsburg, Virginia, April 26-29, 1992, pp.350-357.

New Horizons of Parallel and Distributed Computing

Guo, M.; Yang, L.T. (Eds.)

2005, XIV, 333 p. 136 illus., Hardcover

ISBN: 978-0-387-24434-1