

PROGRAMMING DISTRIBUTED SYSTEMS BASED ON GRAPHS

JIANNONG CAO, LICHUCHA FERNANDO

Department of Computer Science, University of Adelaide

Adelaide, S.A. 5001, Australia

E-mail: jiannong@cs.adelaide.edu.au

and

KANG ZHANG

Department of Computing, Macquarie University

Sydney, NSW. 2109, Australia

E-mail: kang@mpec.mq.edu.au

ABSTRACT

Graphs form an important data structure in computer science, and many interesting computational problems can be expressed in terms of graphs. Graphs have also been widely used in modeling and designing parallel computations and distributed computing system control functions. However, most existing programming languages do not provide graphs as data or control structures. This paper proposes an integrated approach for programming distributed systems based on programmer-specified logical graphs. A distributed implementation of graphs provides a set of linguistic constructs that extend the capability of existing programming languages by directly supporting distributed graph operations. Programmers are relieved from the burden of coding low-level system functions but instead can concentrate on the logic of their distributed programs. More importantly, the programmer is given enough flexibility to exploit the semantics of the graph construct to deal with different aspects of distributed programming such as communication, synchronization and reconfiguration.

1. Introduction

Graphs form an important data structure in computer science, and many interesting computation problems can be expressed in terms of graphs. Graphs have also been widely used in modeling and designing parallel computations and distributed computing system control functions. As distributed computers become popular, parallel and distributed algorithms have been proposed for solving computation-intensive problems in order to improve processing speed. Many of these algorithms can be expressed as a collection of parallel functional modules whose relationships can be defined by a directed graph. On the other hand, graphs have also been widely used in modeling and designing distributed control functions. In a distributed system, processors are physically separated but linked by a communication network. Distributed graph algorithms are used to implement several basic facilities in the system such as message routing, interprocess communication and synchronization. Graphs can also be used in other contexts of distributed computing. A distributed program may need to deal with some or all of these aspects during execution. A desirable integrated approach would allow the programmer to deal with different aspects of distributed programming in a unified way.

This paper proposes an integrated approach for programming distributed systems based on programmer-specified logical graphs. A distributed implementation of graphs provides a set of linguistic constructs that extend the capability of existing programming languages by directly supporting distributed graph operations. Because programmers can manipulate the distributed system based on the logical graph, they are relieved from the burden of coding low-level system functions but instead can concentrate on the logic of their distributed programs. More importantly, programmers are given enough flexibility to exploit the semantics of the graph construct to deal with different aspects of distributed programming in an integrated way. In different contexts of a distributed program a graph can carry different meanings and graph operations can

be used to implement different functions. In this sense graphs are intensions and graph operations depend on the values of graphs at different program contexts ⁹.

The rest of the paper is organized as follows. Section 2 briefly discuss the roll of graphs in distributed programming. Section 3 proposes a graph-based language-level construct and shows how the construct can be used in different contexts of distributed programming. Section 4 discusses the issues of distributed implementation of graphs. In Section 5 we describe a prototype implementation of the proposed graph construct. Section 6 concludes the paper with our future work.

2. Modeling Distributed Programs based on Graphs

There are three aspects of a distributed program, namely *data*, *processing*, and *control*. Distribution can occur at any of the three dimensions and graphs can be used to describe all these distributions (see Figure 1). Consequently, many problems to be solved in parallel and distributed systems can be modeled by graphs and/or their variants. There is a large class of parallel computations that can be expressed in terms of graphs. For example, a computation can be expressed as a data-dependency graph whose vertices are function modules and whose edges denote data dependencies between function modules. Efforts have been made to allow the programmer to *implicitly* specify parallelism in the application program by using a graph-based programming model, which can be either a language such as DGL ¹², or a graphical interface such as HeNCE ³. The procedures represented by the vertices of the graph are written in a conventional, sequential language such as C or Fortran. The run-time system manages the scheduling and parallel execution of function modules on multiple processors.

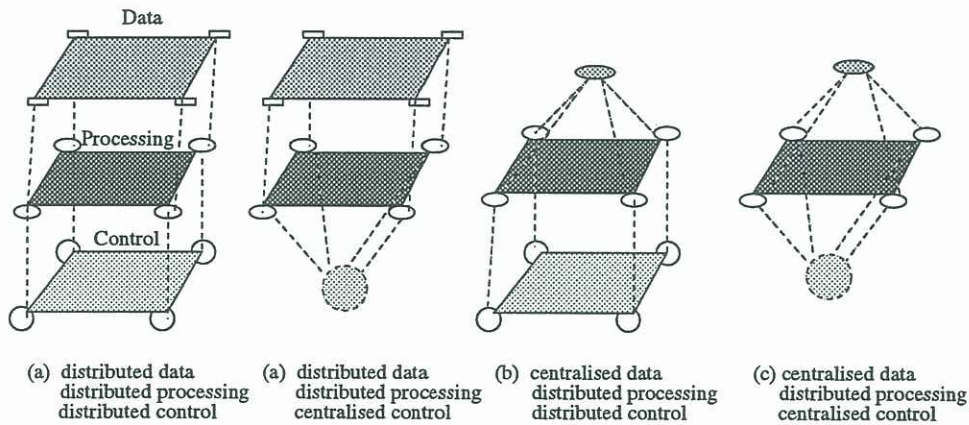


Fig. 1. The three aspects of a distributed program.

The above facilities aim at simplifying the writing of parallel/distributed application programs where the programmer needs not to be concerned with managing concurrency. On the other hand, a great number of distributed functions are originally programmed as a set of decentralized and loosely coupled processes, where parallelism and distribution need to be explicitly dealt with. Examples of these functions include distributed system control functions such as message routing in a computer network, interprocess communications, synchronization, task scheduling, and system reconfiguration. The underlying logical structure of the processes of the system control function can often be modeled by a graph connecting these processes. Consequently, solutions to these control problems are often, partially or entirely, expressed as distributed graph algorithms in terms of the underlying control graphs. ^{8,6,14,15,16,17,18}

Although existing approaches to supporting distributed programming provide the programmer with the ability to construct programs from multiple processes that cooperate to achieve some common goals, few of them allow the programmer to have control over the distributed program structuring and to program the system based on the structure. Most existing distributed programming tools do not provide graphs as

data or control structures. To our knowledge, there is no support of graph operations at the programming language level. The programmer must manually translate graph-based algorithms to implemented programs using whatever constructs provided by the language, explicitly writing the required graph operations. We claim that when programming distributed control functions whose algorithms are expressed on graphs, interactions and coordinations of processes in the algorithm can be expressed in terms of the logical graph, rather than the low-level communication and synchronization primitives. The main purpose of this paper is to propose a general-purpose graph-based programming construct that allows programmers to write distributed programs based on a programmer-specified logical graph. The proposed construct provides a common base for programming distributed systems, where the distributed syntax differs as little as possible from the sequential one but is associated with a natural distribution semantics.

3. Graphs as a Language-Level Construct

We define a distributed program as a collection of local processes that may execute on several processors. Each process performs operations on available data at various points in the program and communicates with other processes. We assume that communication between processes is solely via message passing.

We focus on distributed programs of coordinated processes, where a distributed program is built by gluing together active pieces⁵. The “glue” must allow these independent activities to communicate and synchronize with each other. A graph model provides this kind of glue.

Here, we propose to have graphs as a language-level data and control construct, consisting of a collection of linguistic primitives that directly supporting distributed graph operations required in different contexts. A *graph* $G(V, E)$ is a finite set V whose members are called *vertices* and a finite set E whose members are called *edges*. An edge is an ordered pair of vertices in V . In our context, vertices of a graph are processes and the relationship between the processes defines the edges of the graph. In different contexts of distributed programming, processes and their inter-relationships carry different meanings. Such contexts are defined by several parameters organized into several dimensions (see Section 3.3).

3.1. The Graph Construct

The language-level graph construct consists of

- a conceptual graph (directed or un-directed), whose vertices are associated with local processes (LPs),
- a processes-to-vertices mapping, which allows the programmer to assign LPs to specific vertices, and
- an optional vertices-to-processors mapping: which allows the programmer to explicitly specify the mapping of the logical graph to an underlying network of processors. When the mapping specification is omitted, the underlying execution system transparently performs the mapping.

Figure 2 depicts the basic components of a graph construct associated with an distributed program.

The programmer first defines variables of the graph construct in a program and then creates an instance of the construct. The following steps are needed:

- Step 1: define the conceptual graph describing the logical relationship between LPs, instantiate the construct and associate a name with the instance.
- Step 2: define the mapping of the LPs to the graph’s vertices and, if necessary, the mapping of the graph vertices graph to the underlying network nodes.
- Step 3: bind the mappings to the graph.

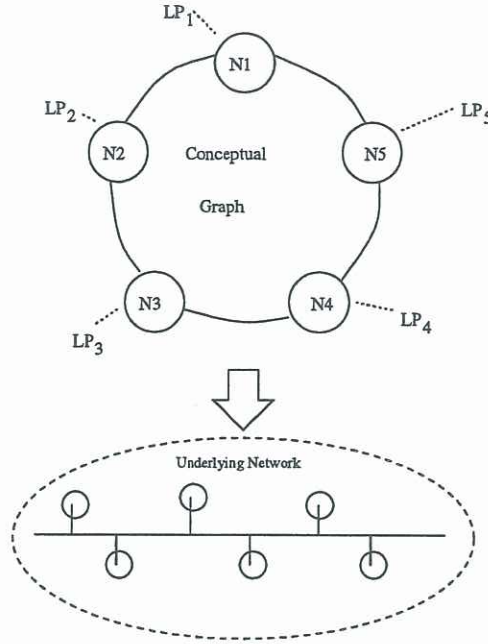


Fig. 2. Creation of a graph construct instance.

The result of above processing is the creation of the representation of the defined graph construct instance and the information required for implementing operations on the instance.

3.2. User-level Programming Primitives

Once the local context for the graph instance is created, interaction and coordination of LP's can be implemented by invoking operations defined on the specified graph. These operations can be categorized into several classes

- *Communication and Synchronization.* These operations provide various forms of communication primitives which can be used to pass messages from one vertex to one or more other vertices in the graph. These primitives can be used by the LPs associated with the vertices to communicate with each other and to synchronize their operations without knowing the low-level details such as name resolution, addressing and routing.
- *Query.* These operations provide information about the graph, such as the number of vertices in a graph and whether an edge exists between two vertices. These informations are useful for many system control and configuration functions.
- *Subgraph generation.* These operations derive subgraphs such as a shortest path between two vertices and spanning trees of a graph. Many distributed algorithms rely on the construction of some forms of subgraphs of the underlying control graph.
- *Graph update.* These operations provide the programmer with the capability to dynamically insert into and/or delete from a graph edges and vertices. These primitives are very useful for dynamic control functions such as dynamic reconfiguration.

3.3. Dimensionality

Note that in a distributed program, multiple graph construct instances may be created for different purposes at different times. Also, mappings can be dynamically changed during the program execution and

LPs can be bound to multiple conceptual graphs. As a matter of fact, there are several dimensions in creating graph instances and performing graph operations.

Each graph instance is logically a 3-tuple (Functionality, Time, Version), representing an instance of a 3-dimensional world as shown in Figure 3. The three dimensions are

- *Functionality*. This dimension reflects different aspects of distributed programming such as computation, communication, coordination and reconfiguration. The graph represents either data organizations, logical structuring for process coordination, or system configuration.
 - For distributed computations on graphs, the graph vertices carry data and the processes bound to the vertices manipulate the data. The edges of the graph represent data dependency relationships;
 - For distributed control functions, the graph vertices carry state information and the processes bound to the vertices communicate with each other of the information and make decisions. The edges of the graph represent the communication paths.
 - For dynamic distributed system configuration, the graph vertices denote system components and the processes bound to the vertices are the agents for the components. The edges of the graph represent the connections between the components.
- *Time*: during the execution of a distributed program, the computation may be represented by different graph instances at different moments in time.
- *Version*: graphs may have different versions indicating computations under different input conditions. A graph from an early version may be partially reused and incrementally updated for subsequent computations.

All together, these dimensions define the contexts of a graph-based distributed program. A program can dynamically switch to different contexts at different times, as illustrated by Figure 4.

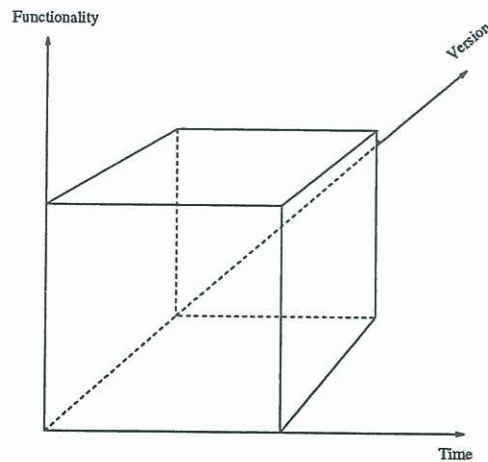


Fig. 3. The dimensionality of the graph model.

Since the graphs carry different meanings for different program contexts, the semantics of the graph operations will depend on the context defined by the programmer. A graph operation can be implemented in different ways but provide the same interface to the programmer. The graph construct can provide marshaling of message parameters in the primitives to allow programmers to describe and implement different aspects of distributed system programming. The intensional approach to parallel programming described in ⁹ may provide valuable techniques to realize these desirable features.

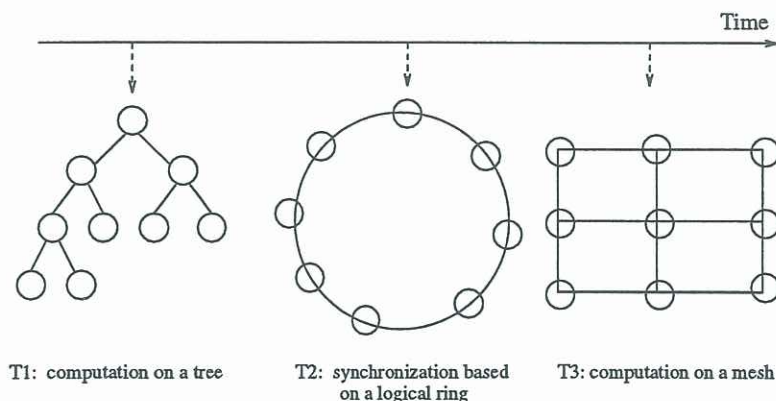


Fig. 4. Dynamic switching of contexts.

4. Implementation issues

The proposed graph construct can be implemented at either the operating system level or the programming language level. It can also be implemented as library routines incorporated in any sequential languages, such as C, Ada, and Modular-2. The addition of these graph-based communication and synchronization primitives yields a high-level, logical graph-based programming platform.

There are several important issues regarding the implementation of the proposed graph construct. The first issue is to manage the mapping of graphs to underlying network processors. If the user specifies the mapping, the problem becomes straightforward. Otherwise the execution system needs to explore task scheduling techniques in order to make efficient use of system resources and/or to speed up the computations.

The second issue is the representation of a graph in a distributed environment. A graph can be either directed or un-directed, and can be represented in three different ways: centralized, partitioned and replicated. Implementations of graph operations will vary for different forms of representations.

The third issue is the distributed implementation of graph operations. Results of existing work on distributed graph algorithms can be used to help the implementation^{1,2,7,10,13}.

The fourth issue is the management and retrieval of graphs generated and used in different contexts and at different conceptual levels. This is typically handled by using tagging schemes found in traditional implementations of dataflow languages.

5. A Prototype Implementation

We have implemented a prototype, called DIG (Distributed Implementation of Graphs), of the proposed graph construct⁴. The initial implementation concentrates on a class of distributed programs concerned with the cooperation and synchronization of local processes to realize a common goal. Examples of such distributed programs are distributed system control functions, such as synchronization, task scheduling, and server programs.

The prototype is implemented on a local area network of Sun workstations. The graph construct, as an extension to C, is provided to the programmer by a library of C routines. A *distributed* representation of a logical, directed graph (a partitioned adjacent matrix) is used. Currently, programs can only be created statically, with one process being assumed for every processor in the network. System functions are written to allocate LPs on remote processors. The communication protocols uses UNIX sockets to pass messages between processors. Multiple DIG-based programs can co-exist and share socket addresses. Basic primitives of communication, query, subgraph and graph update operations have been implemented. Graphs can be dynamically modified but no support is provided for reconfiguration of LPs.

To experiment with the DIG construct, we have designed simple example distributed programs, including

a global-sum program on hypercube and a distributed mutual exclusion program based on trees ⁴. Here, let us consider an example showing how DIG is used to implement a distributed matrix multiplication algorithm ¹¹. The problem is to implement the multiplication of two matrices A and B of order N on four processors. The matrices A and B are decomposed into odd and even rows and odd and even columns, respectively, which are inputs to the processors. Each of the four processors accept input vector components from west and north and calculate an element of the result matrix (see Figure 5). After an initial calculation four partial outputs are produced at approximately the same rate as the input is consumed. Consequently a high degree of parallelism is achieved.

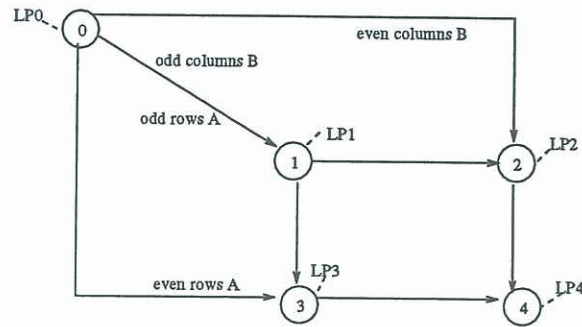


Fig. 5. Logical graph for matrix multiplication

The following text declares the matrix graph, as an instantiation of the DGraph template. The presentation here is based on the C language.

```
DGraph matrix = {{0..4},{0,1}, {0,2}, {0,3}, {1,2},{1,3},{2,4},{3,4}};
```

The declaration associates the DGraph instance with a name *matrix*. The first component on the right hand side of the declaration specifies the vertices of the graph. The second component consists of a set of edges descriptions, each defines an edge between two adjacent vertices in the graph.

The following declaration specifies a map, named *M1*, between the graph vertices and the local programs. In this example, there are two types of LPs: the *Input* LP, which decomposes the matrices and distributes the data, and the *Calculator* LPs, which calculate the elements of the resulting matrix.

```
LV-MAP M1 = {{0,"Input"}, {1, "Calculator"}, {2, "Calculator"},
              {3, "Calculator"}, {4, "Calculator"}};
```

The programmer can specify a mapping *M2* of the graph vertices onto the network processors. This mapping is optional and, if it is missing, a *default* mapping will be performed.

Given the graph declaration, the LP-to-vertex map, and the vertex-to-processor map, the function `CreateDIG(matrix, M1,"MatrixMulti")` can be called to combine the graph declaration and the maps to produce an instance, called *MatrixMulti*, of the DIG construct.

Instances of the DIG construct are created statically by the **Main Program**. The main program is responsible of initiating and terminating, if necessary, the execution of the distributed program. The main program can also be used for monitoring the execution of LPs. In this example, the *Input* LP can act as the *Main* program.

Each LP starts by creating a local operating context for operations of the specified DIG instance. This is done by calling the function

```
SetUpLocalDIG (DGraph-name,DIG-instance-name,myvid)
```

which creates the distributed representation of the graph and the information required by the protocols for graph operations. Each LP has a unique identifier, *myvid*, associated with the operating context.

The code for the *Input* LP and the *Calculator* LPs can be written by using the DIG primitives such as `SendToVertex`, `SendToChildren`, and `ReceiveFromParent`. All the communication primitives are defined in terms of the logical graph and the programmer needs not to know the details of naming and message

passing.

6. Conclusions

We have proposed a high-level, graph-based approach to programming distributed systems. The definition of a graph-based distributed program consists of a conceptual graph and collection of local functions invoked by messages traversing the graph. Using the construct, the programmer can write distributed programs in a way very much similar to that of writing sequential programs without knowing the details of message passing, task mapping and graph operations. The proposed approach can be applied to a wide range of distributed programming problems and can be implemented as a set of language primitives in distributed programming languages or as an extension to conventional, sequential languages.

Our immediate future work includes optimization of distributed implementations of graph operations and studying the performance of graph-based distributed programs. We should also look into graph operations involving multiple graphs, which will be useful for integration of different system functions or computations to form a larger program.

7. References

1. M. Ahamad and A.J. Bernstein,, "Multicast Communication in UNIX 4.2BSD", *Proc. 5th IEEE International Conference on Distributed Computing Systems*, 1985, pp.80-87.
2. B. Awerbuch, "A New Distributed Depth-First-Search Algorithm", *Information Processing Letters*, 20(1985) pp147-150.
3. A. Beguelin and J.J. Dongarra, "Graphical Development Tools for Network-Based Concurrent Supercomputing", *Proc. Supercomputing 91*, Albuquerque, 1991, pp435-444.
4. J. Cao, L. Fernando, and K. Zhang, "DIG: A Graph-based Construct for Programming Distributed Systems", *Submitted for publication*, March, 1995.
5. N. Carriero and D. Gelernter, "How to Write Parallel Programs: a First Course", *MIT Press, Cambridge, Mass*, 1990.
6. S. Chandrasekaran and S. Venkatesan, "A Message-Optimal Algorithm for Distributed Termination Detection", *Journal of Parallel and Distributed Computing*, 8(1990), pp245-252.
7. E.J.H. Chang, "Echo Algorithms: Depth Parallel Operations on General Graphs", *IEEE Trans Software Engineering*, Vol. SE-8, No.4, July 1982, pp391-401.
8. M.S. Chen and K.G. Shin, "Depth-First Search Approach for Fault-Tolerant Routing in Hypercube Multicomputers", *IEEE Trans. Parallel and Distributed Systems*, Vol.1, No.2, April 1990, pp152-159.
9. W. Du, "An Intensional Approach to Parallel Programming", *IEEE Parallel & Distributed Technology*, August, 1993, pp22-32.
10. R.G. Gallager, P.A. Humblet, and P.M. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees", *ACM Trans. Programming Languages and Systems*, Vol.5, No.1, January 1983, pp66-77.
11. C.A.R. Hoare, *Communicating Sequential Processes*, *Prentice-Hall International*, 1985.
12. R. Jagannathan, A.R. Downing, W.T. Zaumen, and R.K.S. Lee, "Dataflow-based Methodology for Coarse-Grain Multiprocessing on a Network of Workstations", *Proc. 1989 International Conference on Parallel Processing*, pp II-209-216.
13. S. Katz and O. Shmueli, "Cooperative Distributed Algorithms for Dynamic Cycle Prevention", *IEEE Trans Software Engineering*, Vol. SE-13, No.5, May 1987, pp540-552.
14. J. Kramer, J. Margg, and K. Ng, "Graphical Configuration Programming", *IEEE Computer*, October 1989.
15. H. Garcia-Molina and A. Spauster, "Ordered and Reliable Multicast Communication", *ACM Trans. Computer Systems*, Vol.9, No.3, August, 1991, pp242-271.
16. K. Raymond, "A Tree-Based Algorithm for Distributed Mutual Exclusion", *ACM Trans. Computer*

-
- Systems*, Vol.7, No.1, February 1989, pp61-77.
17. M. Singhal, "Deadlock Detection in Distributed Systems", *IEEE Computer*, November 1989, pp37-48.
 18. J. Xu and K. Hwang, "Heuristic Methods for Dynamic Load Balancing in a Message-Passing Multicomputer", *Journal of Parallel and Distributed Computing*, 18(1993) pp1-13.