GeDiKe: An attempt to reduce the complexity of distributed programming

Bita Gorji-Ara and Mohammad Hossein Reshadi

University of Tehran
Electrical and Computer Engineering Department
14399 Tehran, IRAN
Phone: 98-21-800-9215; Fax 98-21-877-8690
Bita@cad.ece.ut.ac.ir, Reshadi@cad.ece.ut.ac.ir

Abstract

Although distributed processing, in principle, provides speed up and access to high performance resources, it will be little used if people could not easily exploit it for their applications. In this way, a lot of network application developers attempted to facilitate development of specific distributed applications using different mechanisms. On the other hand, there are many common modules in different distributed applications that of course, their implementation may differ accordingly. Iintending to reduce the complexity of converting a centralized application to a distributed one, we have introduced a generic distributed kernel (GeDiKe) in this paper. Users can save their design in XML format defined by themselves and then customize and initialize GeDiKe to execute their algorithms. The results of execution will be summarized in other XML files and can be processed by any application.

GeDiKe has a modular architecture. Each module is a flexible and replaceable unit and has a definite interface through which other modules can interact.

The architecture utilizes the benefits of XML, Extensible Markup Language, in modeling the tasks. Processing XML is much easier than any other conventional language used for modeling distributed applications.

1. Introduction

Many researchers and application developers need to revise their application to a distributed version to achieve better performance. However, developing a distributed application imposes extra works compared with centralized applications. To facilitate distributed programming, many application-specific tool developers attempted to propose mechanisms for automating this process [1]. Of course, full automation has not been achieved yet. Introducing modeling languages such as HPF and HPC++ along with powerful compilers are successful examples of such attempts. These languages automatically detect iterative bodies and provide strong data parallelism [2],[3]. In some other attempts, available independent tools are composed to provide required resources and to construct a customized distributed application.

In many applications such as CAD applications, there are still case-by-case solutions and methodologies for generating a distributed application [4],[5]. This involves developers with unfamiliar concepts such as task decomposition, resource management, load balancing, network protocols and so on. In this paper, we propose a mechanism for facilitating and accelerating the generation of such applications.

Distributed applications have many common steps that can be designed and implemented once and used various times. This idea leads to propose a Generic Distributed Kernel (GeDiKe) that should have three main characteristics:

- It should be composed of replaceable and flexible modules. Having definite interface and hidden implementation, the modules can be marketable units.
- The engine should use a strong modeling language through which users can describe the structure and data of their design.
- The engine should provide a way that user can customize the functionality.

In section 2 we introduce a mechanism for modeling the tasks in a distributed engine utilizing XML and will show its strengths and benefits for this approach. In section 3 architecture of GeDiKe will be introduced. In section 4 we summarizes the advantages and weaknesses of the architecture and finally in section 5 conclusion and future works will be presented.

2. The Modeling Language

The entry point of any application-specific tool is a compiler which receives a design described in a modeling language. Various languages have been used in different experiences. For example, in NetSolve [6] the modeling language is Fortran, C, Java or MATLAB. In SCIRun

[7], a visual programming language is used to compose SCIRun libraries or programmer-supplied modules in to a simulation application. In Nimrod [8], an experiment definition language is used to organize multiple invocations of a supplied application program [1]. Although these applications isolate the programmers from decisions regarding the resources used to solve the problem, most of application-support systems, including our system, need user interaction for task decomposition, debugging and evaluating performance of the system. In next section, we explore the mechanism that programmers can use to interact with kernel to customize their applications.

GeDiKe uses XML [9] (Extensible Markup Language), a strong language with many benefits for our approach, for modeling user's design and communication format.

The first standard version of XML was introduced in 1998 by World Wide Web Consortium (W3C). Like HTML, it is based on SGML (Standard Generic Markup Language), a strong language designed for storing very large structured documents. Tags and attributes in XML describe the structure and meaning of the data and in fact, XML is both data and document. XML grammar relies on regular expressions and hence its processing is very easy and fast. XML also offers the following advantages:

- XML can transfer structured data. In a design, an object may contain some objects that they, in turn, may contain other ones. Saving discrete objects and retrieving their relations at load time can be a very slow task. Figure 1 shows how XML can accelerate this task. In this example, object *A* contains objects *B* and *C* and object *B* contains objects *D* and *E*.

```
<A>
    <B>
        <D>
        </D>
        <E>
        </E>
    </B>
    <C>
    </C>
</A>
```

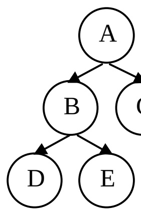

Figure 1- Representing a tree in XML

- In a distributed environment, a transmitted message may contain user data as well as system level information. Utilizing XML, complex headers for defining the meaning of different parts of the message is not a necessity any more.
- Designers can use their own set of tags, which are meaningful for themselves, and they are not limited to predefined keywords.
- Learning XML is not burdensome on the programmers, because XML is very simple and similar to HTML.
- Capabilities such as comments and character data sections in XML, facilitates it to use any other language. Of course using these languages needs an extra effort for compilation or interpretation of the model. This is like using script languages in HTML.
- The meaning of data in an XML file is defined by tag names rather than their positions. This means that modifications to the existing structures, for releasing new versions, needs very little effort while downward compatibility is simply achieved.
- Although, being generated to be used by machine, XML is completely readable for human beings and this simplifies debugging and testing the system.
- As a standard, XML is fully supported by the World Wide Web Consortium (W3C) and this support has led to many available free and documented tools with which developers can view, convert, load, save and check the validity of XML files.

Strength and benefits of XML has made it a suitable format for generic usages. In our engine, we have used it for describing designs and their input and output format as well as for communication purposes.

To clarify the concept of modeling the design with XML, here we introduce a model for input and output format of a simple distributed environment for fault simulation.

2.1. Example: Fault Simulation

In fault simulation, the goal is to find a minimum set of test vectors that can detect almost all faults in a digital circuit. To accomplish this, for each test vector, the good circuit is evaluated and then undetected faults are applied to the circuit (fault injection) and the circuit is evaluated again. A test vector can detect a fault if the output of the good circuit and the faulty circuit are different. If so, the detected fault is removed from the fault list. Parallelism of this application is simple and easy. For example, circuit evaluations can be done on different processors and detected faults can be reported to all others.

The input format should contain the following information:

- Description of digital circuit
- Reduced list of faults
- Test vectors

Figure 2 shows a simple digital circuit and Figure 3 shows one possible structure for input.

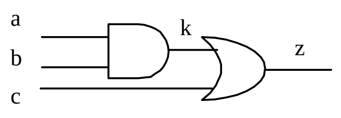

Figure 2- Simple digital circuit

```
<faultsimulator>
    <cir>
        <or>
            <and>
                <i>a</i>
                <i>b</i>
                <o>k</o>
            </and>
            <i>c</i>
            <o>z</o>
        </or>
    </cir>
    <FL>
        <sa0>k</sa0>
        <sa0>z</sa0>
        <sa0>c</sa0>
        <sa1>a</sa1>
        <sa1>b</sa1>
        <sa1>z</sa1>
    </FL>
    <vectors order= "a,b,c">
        <v value="0,0,0"/>
        <v value="0,0,1"/>
        <v value="0,1,1"/>
        <v value="1,1,1"/>
    </vectors>
</faultsimulator>
```

Figure 3- Input format of fault simulator

The fault simulator in this model has tree major parts: the digital circuit, the fault list and the test vectors that are represented by <cir>, <FL> and <vector> tags respectively. The circuit is described in a tree from output to inputs. Gate tags, i.e. OR and AND, contain other gates or wires as input and wires as output. This mechanism of representing a circuit accelerates it's loading compared with a net-list representation, because it needs minimum lookups for wires and nets.

In the fault list (<FL>) section the <sa0> and <sa1> tags show the stock at zero and stock at one on wires respectively.

The "*order*" attribute in test vectors (<vectors>) section shows the order of primitive inputs in which values are represented.

The output file (Figure 4) is the list of detected faults and the corresponding test vectors that detects them. In this section, the concept of "*order*" and "*value*" is the same as test vectors section.

Note that, there are many alternatives for modeling the data and developers should design the best model to cover their requirements.

```
<dfs order="a,b,c">
    <df value="0,0,0">
        <sa1>a</sa1>
        <sa1>c</sa1>
    </df>
    <df value="0,0,1">
        <sa0>c</sa0>
        <sa0>z</sa0>
    </df>
    <df value="1,0,1">
        <sa1>b</sa1>
    </df>
</dfs>
```

Figure 4- Output format of fault simulator

Because GeDiKe is designed for fast and easy conversion of centralized applications, the XML input files are supposed to be generated by automatically from the data structure that developers use to run their algorithms. There are successful experiences in representation of a structure in memory in a XML file [10].

3. Architecture of the GeDiKe

GeDiKe is a kernel that is composed of flexible and replaceable modules. Each module can be a separate dynamic link library (dll) [11] that hides the details of implementation and has a well-known and documented interface. Besides, this creates a new market for modules and helps the developers to focus their efforts on a small portion of a big design and reuse other available modules.

Other technologies such as COM [12] objects can be used instead of dynamic link library but the overhead that these technologies impose to the performance should be considered.

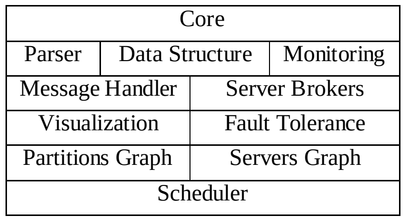

Figure 5- Modules of a Client Element

Figure 5 and Figure 6 show a simplified block diagram of dll modules in both client and server elements. In each element, every dll connects as a plug-in to a core and then performs the necessary operations. Obviously, these dlls can be replaced by other appropriate ones in the configuration stage of the system.

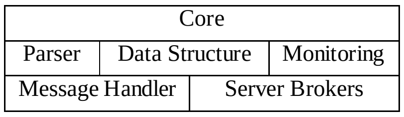

Figure 6 - Modules of a Server Element

In the following sections, we describe the basic modules of these elements.

3.1. Libraries

An increasingly important strategy for transparent inclusion of parallelism is to encapsulate the parallelism and network concepts in component libraries. There are two types of libraries used in GeDiKe:

3.1.1. Data Structure library:

Modules connected to the core work and communicate through predefined abstract data structure. This abstract data structure must be customized by developers to cover application's requirements. To do this, a standard library is prepared that contains basic classes such as TNode, TDataStructure, TGraph, TEvent, TQueue and so on. Developers can inherit their own classes from these basic classes and add new properties to them. The new classes must implement the partitioning algorithm. This new data structure will be plugged to the core by DataStructure.dll.

3.1.2. Functional library:

The implementations of basic distribution and parallelism algorithms are hidden in various modules, which will be described in the following section.

3.2. Basic Modules

3.2.1. Parser.dll

One of the most sophisticated parts of application specific tools are their compilers[13], because of complexity of their languages. XML has been designed using regular expressions and optimized for fast and easy loading.

In GeDiKe, developers can define their own set of tags in XML using parser.dll. Parser reads XML tags and notifies DataStructure.dll to do appropriate settings. Parser may offer other facilities like validity checking of input format and error handling. The Parser.dll can be an extension to SAX [14], which is a standard event driven XML parser.

3.2.2. DataStructure.dll

Although DOM (Document Object Model) is known to be the standard memory interface for XML, it is not suitable for our purpose because of the following reasons:

- DOM is not very customizable because it uses a single node data structure for all types of XML constructs.
- Loading a big XML file in DOM and then converting it to the desired data structure has a lot of memory overhead.
- DOM stores the whole information in text format and is optimized for representing an XML file in the memory rather than a hardware model. Therefore, it is not time and space efficient for some designs.

Figure 7 shows a sample memory representation for the contents of Figure 3. It is obvious that this is not the only way of representing the information. However, the nodes in the graph are customized classes inherited from basic abstract TNode class. This class only provides abstract primitive methods that every node in a graph must support. By inheriting from basic classes, customized classes should implement such abstract methods and add application specific properties to them.

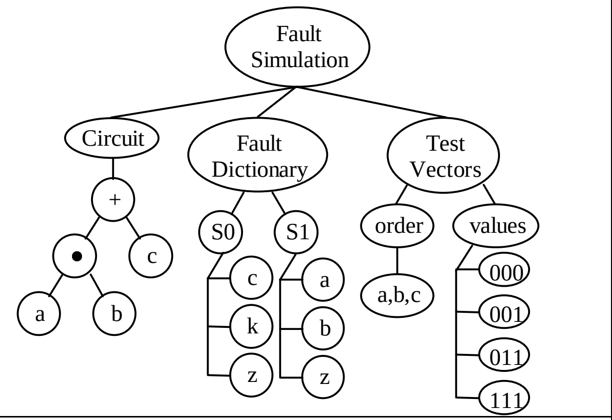

Figure 7- A sample memory model for Figure 3

Note that the data structure can be designed as if it was used for a single node application. For example, the data in Figure 3 can be processed by a centralized application and the server side can still use this application but receives a reduced data file with the same structure.

3.2.3. PartitionsGraph.dll

This dll creates a graph representing the final partitions to be distributed on the servers. Edges in this graph show the dependency of partitions.

3.2.4. ServersGraph.dll

This dll describes the delay of connections and the processing power of available servers in a graph.

3.2.5. Scheduler.dll

Using results of PartitionsGraph.dll and ServersGraph.dll, the Scheduler.dll assigns partitions to servers.

3.2.6. ServerBrokers.dll

This unit associates an ID for each server and connects to the available servers. It also provides multicast features. In this way, there is no restriction on the network protocol used for connections and it can be determined by the developers by replacing this module.

3.2.7. MessageHandler.dll

This unit is in fact the user-defined manager of the environment in which the developers determine the algorithm by which external messages are received and processed. Besides communication of units, security issues, such as coding and decoding the data, are also considered in this unit.

3.3. Client/Server Core

Core is the only fixed part of the engine and the whole application is built by configuring the core through connecting two sets of units to it. The first set is those units that developers *must* provide their own customized version. This set includes MessageHandler and DataStructure units. The second group is those units that developers *may* replace or use the available ones. This set includes ServerBroker, ServersGraph, Scheduler and PartitionsGraph units but developing them needs more advanced knowledge about the environment and the network.

After configuration, the core performs partitioning, scheduling, running and termination of the task according to the messages it receives. It also adds signature and system level tags and information to the in-coming and out-going messages and data files.

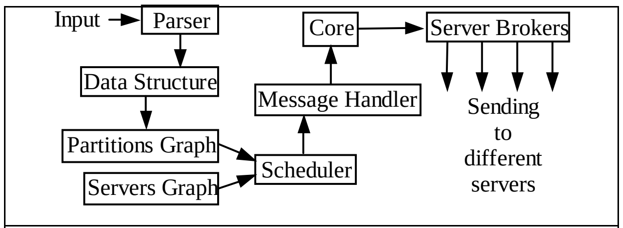

Figure 8- Design Preparation for Distribution

3.4. Flow of data before distribution

Figure 8 shows the process of preparing a design description for distribution. In this process, the Parser detects XML tags and notifies DataStructure unit to create appropriate nodes. The result is delivered to PartitionsGraph in which the structure is partitioned and stored. Meanwhile the ServersGraph prepares, statically or dynamically, a graph of existing server nodes and their connections and sets the measures of processing power and communications delays. Receiving this graph and the partitions graph, the scheduler unit performs scheduling algorithms and assigns server IDs to the partitions. Customized and system tags are added to the partitions in MessageHandler and core unit respectively. Finally, the ServerBrokers unit receives the partitions and put them in the buffer of appropriate server broker.

4. Advantages and Weaknesses of the Architecture

Advantages of the proposed idea are as follows:

- Different teams can work on different parts of the tool independently.
- Different implementations of units will intensify the engine gradually.
- Reusing available modules can accelerate the development of a distributed tool.
- Distributed algorithms can be tested by this engine easily.
- Modifying one unit does not force recompilation of the whole system.
- Developers can think of centralized applications and utilize the engine to make a distributed tool.

Weaknesses of the proposed architecture are as follows:

- Because of using XML as a communication format, transmitted messages may be larger packages compared with other formats. Of course, it is closely related to user's design.
- GeDiKe relys on tool developers for task decomposition, and handling some events.

5. Conclusion and future work

In this paper, we proposed a distributed engine that accelerates and facilitates the development of distributed tools. This engine consists of different replaceable units and fixed cores to which the units are connected. Developers still can think of centralized applications and develop each conceptually different part of the design in a different module. By following some easy and standard rules, these parts can be used by the engine to make up a distributed application.

Based on what was proposed, the followings can be done:

- Implementing suitable partitioning and scheduling algorithms for well-known hardware algorithms.
- Implementing different versions of modules for different platforms.
- Supporting different network protocols.

6. References

[1] Lan Foster, Carl Kesselman, *The Grid: Blueprint for a new computing Infrastructure*, ch. 7, Morgan Haufman publishers, 1999.

[2] S. Hiranandani, K. Kennedy, and C. -W. Tseng. Compiling Fortran D for MIMD distributed –memory machines. Communications of the ACJ, 1992.

[3] H. Zima and B. Chapman. Compiling for distributed-memory systems. Proc. IEEE, 1993.

[4] C.P.Ravikumar, Vikas Jain, Anurag Dod, “Faster Fault Simulation through Distributed Computing”, IEEE Conference, 1996

[5] George Xirogiannis, “Granularity Control for Distributed Execution of Logic Programs, IEEE Conference, 1998

[6] H. Casanova and J. Dongarra. NerSolve: A network server for solving computational science problems. Tech. Report, University of Tennessee, November 1995.

[7] S, Parker. D. Weinstein and C. Johnson. The SCIRun computational steering software system. Modern Software Tools Scientific Computing, pages 1-44. Boston” Birkhauser Press, 1997.

[8] D. Abramson, R. Sosic, J. Giddy and B. Hall . Nimrod: A tool for performing parameterized simulation using distributed workstations. In Proc. Fourth IEEE Symp. on High Performance Distributed Computing. Los Alamitos, CA IEEE Computer Society Press. 1995

[9] C.F. Goldfarb, P. Prescod, *The XML Handbook*, ch. 53, second edition, Prentice Hall, 2000.

[10] M. H. Reshadi, B. Gorji-Ara, and Z. Navabi, “HDML: Compiled VHDL in XML”, IEEE VIUF, 2000

[11] David J.Kruglinski, *Inside visual C++*, ch. 21, Microsoft press, 1996.

[12] David J.Kruglinski, *Inside visual C++*, ch. 23(COM), Microsoft press, 1996.

[13] Lan Foster, Carl Kesselman, *The Grid: Blueprint for a new computing Infrastructure*, ch. 8, Morgan Haufman publishers, 1999.

[14] C.F. Goldfarb, P. Prescod, *The XML Handbook*, ch. 48, second edition, Prentice Hall, 2000.